
# Scriptum zur Vorlesung
# Digitale Systeme

gehalten von Prof. Dr. Manfred Schimmler

Lehrstuhl für Technische Informatik

Institut für Informatik

Technische Fakultät

Universität Kiel

Inhaltsverzeichnis

# 1. Zahlendarstellung in Computern

Empfohlene Literatur: Klar, R.: Digitale Rechenautomaten, de Gruyter ISBN 3 11 0041944

## *1.1. Darstellung natürlicher Zahlen im B-adischen Zahlensystem*

Im normalen Leben werden Zahlen in der Regel im Dezimalsystem dargestellt. Die natürliche Zahl 201 steht für die Summe

$$\mathbf{2}*10^2 + \mathbf{0}*10^1 + \mathbf{1}*10^0$$

Die Ziffern geben also die Koeffizienten eines Polynoms in 10 an, wobei jede Stelle für eine Potenz des Basiswerts 10 steht.

Das Dezimalsystem ist nicht besonders gut geeignet für elektronische Rechenanlagen, weil es technisch sehr schwierig ist, die zehn unterschiedlichen Ziffern in zehn Wertigkeiten einer physikalischen Größe zu codieren, so daß dieser Code

1. eindeutig ist (d.h. daß man immer weiß, welche Ziffer gemeint ist) und
2. nicht durch Störeinflüsse wie Übersprechen auf Leitungen verfälscht werden kann.

Daher bedient man sich in Computern in der Regel des dualen (binären) oder 2-adischen Zahlensystems. Hier werden die natürlichen Zahlen als Polynome in 2 betrachtet, wobei als Koeffizienten nur die Ziffern 0 und 1 erforderlich sind. Die Dezimalzahl 201 hat dann die Repräsentation 11001001, die zu verstehen ist als

$$\mathbf{1}*2^7 + \mathbf{1}*2^6 + \mathbf{0}*2^5 + \mathbf{0}*2^4 + \mathbf{1}*2^3 + \mathbf{0}*2^2 + \mathbf{0}*2^1 + \mathbf{1}*2^0.$$

Die Beschränkung auf zwei Ziffern erlaubt es, die Koeffizienten durch eine Größe zu repräsentieren, von der nur zwei Werte unterschieden werden müssen, z.B. eine Spannung, wobei 0V für die 0 und 3,3V für die 1 steht. Oder durch eine Lampe, bei der eine 1 durch Leuchten der Lampe und eine 0 durch Nicht-Leuchten codiert ist.

Nun gibt es aber nicht nur diese beiden Systeme zur Repräsentation von Zahlen. So wie wir bisher 10 und 2 als Basis des Zahlensystems verwendet haben, kann man jede positive ganze Zahl B≥2 als Basis eines sogenannten B-adischen Zahlensystems wählen. Eine beliebige Zahl n kann dann eindeutig repräsentiert werden als

$$n = \sum_{i=0}^{N-1} b_i B^i = b_0 B^0 + b_1 B^1 + \cdots + b_{N-1} B^{N-1}$$

Dabei sind die Ziffern $b_i \in \{0,1,2,...,B-1\}$. N ist die Anzahl der Stellen für die Darstellung. Da wir in Computern in der Regel ein festes „Wortformat“ zur Verfügung haben, ist damit für N eine obere Grenze gegeben. Wenn wir z.B. von einer „32-Bit-Architektur“ reden, bedeutet, das, dass die Zahlen mit 32 (binären, d.h. B = 2) Stellen dargestellt werden, also N = 32.

Mit B = 10 erhält man das Dezimalsystem, mit B = 2 das Binär- oder Dualsystem.

Zur Vereinfachung ist folgende Konvention in der Schreibweise üblich:

$$n = (\boldsymbol{b_{N-1}\, b_{N-2} \ldots b_1\, b_0})_B$$

wobei führende Ziffern weggelassen werden, wenn sie 0 sind und die Kennzeichnung durch das tiefgestellte B am Ende wegfällt, wenn man weiß, über welche Basis man redet.

**Beispiele:**

Wir hatten die Zahl $(201)_{10}$ bereits als $(11001001)_2$ kennengelernt. Mit der Basis 8 hat sie folgende Repräsentation:

$$\mathbf{0}*8^{N-1} + ... + \mathbf{0}*8^4 + \mathbf{0}*8^3 + \mathbf{3}*8^2 + \mathbf{1}*8^1 + \mathbf{1}*8^0 = (311)_8.$$

Das 8-adische Zahlensystem wird auch als Oktalsystem bezeichnet.

Wählen wir als Basis die 16, so begeben wir uns ins Hexadezimalsystem. Hier wird aus der Dezimalzahl 201

$$\mathbf{0}*16^{N-1} + ... + \mathbf{0}*16^3 + \mathbf{0}*16^2 + \mathbf{12}*16^1 + \mathbf{9}*16^0 = (C9)_{16}$$

Da wir für die Zahlen „zehn“, „elf“, „zwölf“, „dreizehn“, „vierzehn“ und „fünfzehn“ keine Ziffern zur Verfügung haben, behelfen wir uns mit den Großbuchstaben A bis F, wobei

A für 10

B für 11

C für 12

D für 13

E für 14

F für 15

steht. Die Zahl $(45054)_{10}$ beispielsweise hat die hexadezimale Repräsentation AFFE.

**Merkmale eines B-adischen Zahlensystems:**

- Es gibt B Ziffern 0, 1, ..., (B-1).
- Jede Stelle hat ein Gewicht einer Potenz von B.
- Das Gewicht der Stelle i ist das B-Fache des Gewichts der Stelle i-1.

**Satz:**

Die N-stellige B-adische Darstellung ermöglicht es, jede ganze Zahl aus $\{0,1,...,B^N-1\}$ auf genau eine Weise darzustellen.

**Beweis:**

„Jede Zahl kann dargestellt werden“ wird bewiesen mit vollständiger Induktion nach N.

Induktionsanfang: Sei N=1. Die Zahlen 0, 1, ..., B-1 sind genau durch die B-adischen Ziffern als 1-stellige Zahlen darstellbar.

Sei die Aussage des Satzes richtig für N = k.

Dann können die Zahlen 0, 1, ..., $B^k-1$ dargestellt werden als k-stellige B-adische Zahlen. Die k+1-stelligen Zahlen, die mit einer 0 beginnen, decken also diesen Bereich ab. Die k+1-stellige Zahl mit einer Ziffer z am Anfang gefolgt von Nullen hat den Wert $zB^k$.

Sei m eine Zahl aus $\{0, 1, ..., B^{k+1}-1\}$. Sei $m = q * B^k + r$, mit ganzzahligen und positiven Werten q und r, wobei $r<B^k$ sein soll. Dann ist q darstellbar als die Ziffer q gefolgt von k Nullen und r ist nach Induktionsvoraussetzung darstellbar als eine k+1-stellige Zahl, die mit einer 0 beginnt. Das Ersetzen der ersten 0 von r durch q liefert die Darstellung von m.

Zu zeigen bleibt, daß die Darstellung eindeutig ist. Nun hat die Menge $\{0, 1, ..., B^N-1\}$ genau $B^N$ Elemente. Andererseits stehen $B^N$ N-stellige Zeichenreihen mit den Ziffern {0, 1, ..., B-1}

zur Verfügung. Da jede Zahl dargestellt werden kann ist also keine Zeichenreihe übrig, um eine Zahl doppelt darzustellen. Also ist die Darstellung eindeutig. □

**Beispiele für B-adische Zahlen (B = 2, 3, 8, 10, 16):**

| **Binär 2-adisch** | **Ternär 3-adisch** | **Oktal 8-adisch** | **Dezimal 10-adisch** | **Hexadezimal 16-adisch** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |
| 10 | 2 | 2 | 2 | 2 |
| 11 | 10 | 3 | 3 | 3 |
| 100 | 11 | 4 | 4 | 4 |
| 101 | 12 | 5 | 5 | 5 |
| 110 | 20 | 6 | 6 | 6 |
| 111 | 21 | 7 | 7 | 7 |
| 1000 | 22 | 10 | 8 | 8 |
| 1001 | 100 | 11 | 9 | 9 |
| 1010 | 101 | 12 | 10 | A |
| 1011 | 102 | 13 | 11 | B |
| 1100 | 110 | 14 | 12 | C |
| 1101 | 111 | 15 | 13 | D |
| 1110 | 112 | 16 | 14 | E |
| 1111 | 120 | 17 | 15 | F |
| 10000 | 121 | 20 | 16 | 10 |

### 1.2. Rechnen im B-adischen Zahlensystem

In jedem B-adischen Zahlensystem können wir ähnlich wie im Dezimalsystem rechnen. Man muß aber dabei aufpassen daß man nicht (implizit) Ziffern benutzt, die in dem jeweiligen System nicht vorhanden sind.

**Beispiele:**

Im Oktalsystem sind die Zahlen 7351 und 642 zu addieren:

```
          7351
           642
Übertrag  11100
         10213
```

Im Dualsystem sollen 1101000101 und 10011010111 addiert werden.

```
            1101000101
           10011010111
Übertrag  111110001110
          100000011100
```

Im Hexadezimalsystem sollen CAD von 1234 subtrahiert werden:

```
          1 2 3 4
            CAD
Übertrag  1 1 1 0
            5 8 7
```

Multiplikation der Zahl $21D5_{16}$ mit der Zahl $6_{16}$:

```
  21 D5  *  6
  CAFE
```

Denn 6*5 = 1E (E schreib hin, 1 im Sinn), 6*D = 4E (+1 macht F, F schreib hin, 4 im Sinn), 6*1 = 6 (+4 macht A, A schreib hin, 0 im Sinn) und 6*2 = C.

### 1.3. Darstellung negativer Zahlen

Wir sind gewohnt, Zahlen durch Betrag und Vorzeichen anzugeben. In Rechenanlagen hat diese Technik zwei wesentliche Nachteile:

1. Die Subtraktion muß durch eine eigenständige Einheit ausgeführt werden.
2. Die Entscheidung, ob ein Ergebnis größer, gleich oder kleiner als 0 ist, ist für den Rechner schwierig.

Eine wesentlich elegantere Lösung ist die Darstellung von negativen Zahlen im Komplement. Bei der B-adischen Darstellung unterscheidet man das B-Komplement und das (B-1)-Komplement.

### 1.3.1. B-Komplement

**Definition:**

Sei n eine natürliche Zahl, dargestellt als ***N-stellige B-adische Zahl***. Das B-Komplement von n ist die Zahl $B^N$-n. Das B-Komplement von n wird interpretiert als -n.

Auf diese Weise werden bei ungradzahligem B alle Zahlen von 000...01 bis zzz...z, wobei z die Ziffer des Wertes (B-1)/2 ist zu positiven Zahlen und alle Zahlen von zzz...(z+1) bis fff...f, wobei f die Ziffer (B-1) ist zu negativen Zahlen. Die 0 ist dargestellt als 000...0.

Bei gradzahligem B sind 000...01 bis (B/2-1)FFF...F positiv und B/2 00...0 bis FFF...F negativ.

**Beispiele:**

Im Dezimalsystem mit Zahlen der Länge 5 können die Zahlen von -50000 bis +49999 unter Benutzung des 10-Komplements dargestellt werden:

Die Zahlen von 0 (00000) bis 49999 haben Ihre normale Interpretation, so wie wir sie gewöhnt sind. Die -1 wird jedoch dargestellt als 99999, die -2 als 99998, die -35971 als 64029 und die -50000 als 50000.

Wir betrachten die 8-stelligen Dualzahlen mit negativen Zahlen im 2-Komplement. Mit diesen können wir den Zahlenbereich von -128 bis +127 darstellen.

$$00000000 = (0)_{10}$$

$$01010101 = (85)_{10}$$

$$01111111 = (127)_{10}$$

$$10000000 = (-128)_{10}$$

$$11100110 = (-26)_{10}$$

$$11111111 = (-1)_{10}$$

| Dezimal | 4-stellige 2-Komplementzahlen |
|---|---|
| -8 | 1000 |
| -7 | 1001 |
| -6 | 1010 |
| -5 | 1011 |
| -4 | 1100 |
| -3 | 1101 |
| -2 | 1110 |
| -1 | 1111 |
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |

Was ist der Vorteil vom B-Komplement?

Innerhalb unseres vorgegebenen Bereichs darstellbarer Zahlen können wir jetzt eine Subtraktion der Zahl $Z_2$ von der Zahl $Z_1$ einfach als Addition von $Z_1$ mit dem B-Komplement von $Z_2$ durchführen.

**Beispiele:**

Alle folgenden Additionen und Subtraktionen werden mit 8-stelligen Dualzahlen durchgeführt, wobei negative Zahlen im 2-Komplement dargestellt werden:

$103_{10} - 70_{10}$ :

$$\begin{array}{lll} & 01100111 & (103)_{10} \\ - & 01000110 & -(70)_{10} \end{array}$$

$$\begin{array}{lll} = & 01100111 & (103)_{10} \\ + & 10111010 & +(-70)_{10} \\ \hline & 1|00100001 & (33)_{10} \end{array}$$

$15_{10}$ - $70_{10}$ :

| | | | |
|---|---|---|---|
| | | 00001111 | $(15)_{10}$ |
| | - | 01000110 | $-(70)_{10}$ |
| = | | 00001111 | $(15)_{10}$ |
| | + | 10111010 | $+(-70)_{10}$ |
| | | 0\|11001001 | $(-55)_{10}$ |

$-15_{10}$ - $(-70_{10})$ :

| | | | |
|---|---|---|---|
| | | 11110001 | $(-15)_{10}$ |
| | - | 10111010 | $-(-70)_{10}$ |
| = | | 11110001 | $(-15)_{10}$ |
| | + | 01000110 | $+(+70)_{10}$ |
| | | 1\|00110111 | $(+55)_{10}$ |

Wie konvertiere ich eine Zahl in ihr B-Komplement?

Alle Ziffern z werden in (B-1-z) umgewandelt und dann wird auf die ganze Zahl 1 addiert.

Begründung:

$$-(b_{N-1}*B^{N-1} + b_{N-2}*B^{N-2} + b_{N-3}*B^{N-3} + ... + b_1*B^1 + b_0*B^0)$$
$$= B^N - (b_{N-1}*B^{N-1} + b_{N-2}*B^{N-2} + b_{N-3}*B^{N-3} + ... + b_1*B^1 + b_0*B^0)$$
$$= B^N - B^{N-1} - b_{N-1}*B^{N-1} + B^{N-1} - B^{N-2} - b_{N-2}*B^{N-2} + B^{N-2} - B^{N-3} - b_{N-3}*B^{N-3} + B^{N-3} - ... - B^1 - b_1*B^1 + B^1 - 1 - b_0*B^0 + 1$$
$$= (B - 1 - b_{N-1})*B^{N-1} + (B - 1 - b_{N-2})*B^{N-2} + (B - 1 - b_{N-3})*B^{N-3} + ... + (B - 1 - b_1)*B^1 + (B^1 - 1 - b_0)*B^0 + 1$$

Dies ist gerade die Darstellung der B-Komplement-Zahl in der oben angegebenen Form: jede Ziffer $\mathbf{b_i}$ ist durch (B -1 - $\mathbf{b_i}$) ersetzt worden und am Schluß wird zu allem 1 addiert.

**Beispiele:**

$-(107)_{10}$ als zweistellige Hexadezimalzahl:

$$(+107)_{10} = (6B)_{16}$$
$$(-107)_{10} = (94)_{16} + (1)_{16} = (95)_{16}$$

$-107_{10}$ als Achtstellige Dualzahl:

$$(+107)_{10} = (01101011)_2$$
$$(-107)_{10} = (10010100)_2 + (1)_2 = (10010101)_2$$

**Vorteile des B-Komplements:**

- Addition und Subtraktion können mit derselben Hardware gemacht werden.
- Konversion ist einfach.
- Positive und negative Zahlen können gleich behandelt werden.
- Das Vorzeichen ist bei geradem B an der ersten Stelle (most significant digit) zu erkennen.

**Nachteil:** Überläufe werden nicht automatisch erkannt. Konvertierung erfordert ein „carry“

**Beispiel:**

$(11)_{10} + (10)_{10}$ im System der fünfstelligen Dualzahlen:

$$\begin{array}{rr} & 01011 \\ + & \underline{01010} \\ = & 0|10101 \end{array}$$

Das Ergebnis wird interpretiert als $-11_{10}$. Das ist ja falsch.

Allerdings kann man solche Überläufe erkennen, wenn man eine weitere (führende) Stelle vor der signifikantesten Stelle des Ergebnisses einführt.

1. Fall: Wenn eine positive und eine negative Zahl addiert werden, kann nie ein Übertrag auftreten.
2. Wenn zwei positive oder zwei negative Zahlen addiert werden, und die zusätzliche führende Stelle nach der Addition (Subtraktion) mit der signifikantesten Stelle übereinstimmt, ist kein Überlauf aufgetreten. Unterscheiden sie sich, ist ein Überlauf aufgetreten. Und zwar wenn die Stellen 01 sind, eine nicht darstellbare positive Zahl und wenn sie 10 sind ein betragsmäßig zu großes negatives Ergebnis.

**Beispiele:**

Das Beispiel von eben: $(11)_{10} + (10)_{10}$ im System der fünfstelligen Dualzahlen:

Sicherungsstelle ↓ (Kopie der signifikantesten Ziffer)

$$\begin{array}{rr} & 001011 \\ + & \underline{001010} \\ = & 0|10101 \end{array}$$

Sicherungsstelle der Summe = 0, MSB (most signifikant Bit) der Summe = 1, also Überlauf: nicht darstellbare positive Zahl als Ergebnis.

$10_{10}$ - $11_{10}$ im System der fünfstelligen Dualzahlen:

Sicherungsstelle ↓ (Kopie der signifikantesten Ziffer)

```
    001010
+   110101
=   1|11111
```

Sicherungsstelle der Summe = 1, MSB (most signifikant Bit) der Summe = 1, also kein Überlauf und kein Unterlauf: darstellbare (negative) Zahl als Ergebnis (-1).

**Satz:**

Genau dann ist bei Addition zweier N-stelliger 2-adischer Zahlen das Ergebnis wieder im (mit N Ziffern) darstellbaren Bereich, wenn nach der Addition die Vorzeichenstelle (Stelle N-1) mit der Sicherungsstelle (Stelle N) übereinstimmt.

**Beweis:**

Zu unterscheiden sind folgende 6 Fälle:

1. $n_1 \geq 0$, $n_2 \geq 0$, $n_1+n_2 \geq 2^{N-1}$
2. $n_1 \geq 0$, $n_2 \geq 0$, $n_1+n_2 < 2^{N-1}$
3. $n_1 < 0$, $n_2 \geq 0$
4. $n_1 \geq 0$, $n_2 < 0$
5. $n_1 < 0$, $n_2 < 0$, $n_1+n_2 \geq -2^{N-1}$
6. $n_1 < 0$, $n_2 < 0$, $n_1+n_2 < -2^{N-1}$

Im Fall 1. beginnen beide Zahlen mit einer 0; das MSB sowie die Sicherungsstelle sind also 0. Da $n_1+n_2 \geq 2^{N-1}$ ist, entsteht bei der Addition an der Stelle N eine 1 als Übertrag:

```
  00XXX.....XXX   n1
  00XXX.....XXX   n2
  01XXX.....XXX
  ↑ Stelle N
```

Also ist die Sicherungsstelle der Summe = 0, die Stelle N = 1, d.h. ein Überlauf ist aufgetreten.

Betrachten wir noch den Fall 3. Hier ist $n_1<0$, $n_2\geq 0$, also

11XXX.....XXX $n_1$

<u>00XXX.....XXX</u> $n_2$

XXX.....XXX

↑ Stelle N

An der Stelle N kann ein Übertrag auftreten oder nicht. Im erste Fall sind die beiden ersten Stellen der Summe = 0, im zweiten sind sie beide = 1. Die Stelle vor der Sicherungsstelle wird nicht betrachtet. In beiden Fällen sind die beiden höchsten Stellen gleich, d.h. es ist kein Überlauf und kein Unterlauf aufgetreten.

Der Leser möge sich selbst die anderen vier Fälle überlegen.

## *1.4. Konvertierung natürlicher Zahlen aus dem B-adischen Zahlensystemen ins B'-adische Zahlensystem mit unterschiedlichen Basen B und B'*

Eine im B-adischen Zahlensystem dargestellte Zahl

$$n = \boldsymbol{b_N\, b_{N-1}\, b_{N-2} \ldots b_1\, b_0} = \boldsymbol{b_N} * B^N + \boldsymbol{b_{N-1}} * B^{N-1} + \ldots + \boldsymbol{b_1} * B^1 + \boldsymbol{b_0} * B^0$$

kann mit dem Hornerschema auch in folgender Weise geschrieben werden.

$$n = ((\ldots((\boldsymbol{b_N} * B + \boldsymbol{b_{N-1}}) * B + \boldsymbol{b_{N-2}}) * B + \ldots) * B + \boldsymbol{b_1}) * B + \boldsymbol{b_0}.$$

Ebenso hat n eine gleichartige Repräsentation im System mit der Basis B':

$$n = ((\ldots((\boldsymbol{b_N'} * B' + \boldsymbol{b_{N-1}'}) * B' + \boldsymbol{b_{N-2}'}) * B' + \ldots) * B' + \boldsymbol{b_1'}) * B' + \boldsymbol{b_0'}.$$

Um nun die $b_i$' aus der ersten Repräsentation zu berechnen, können wir durch wiederholte Division durch B' die Reste ermitteln, die dann genau den Ziffern im B'-adischen System entsprechen. Diese Division muß im B-adischen System durchgeführt werden, da wir ja anfangs nur die B-adische Darstellung von n kennen. Daher brauchen wir die B-adische Darstellung von B' für die Division.

Das Divisionsschema sieht dann so aus:

| | | | | | | |
|---|---|---|---|---|---|---|
| n | : | B' | = | $q_0$ | Rest | $\boldsymbol{b_0'}$ |
| $q_0$ | : | B' | = | $q_1$ | Rest | $\boldsymbol{b_1'}$ |
| $q_1$ | : | B' | = | $q_2$ | Rest | $\boldsymbol{b_2'}$ |
| | | . | . | . | | |
| $q_{N-1}$ | : | B' | = | $q_N$ | Rest | $\boldsymbol{b_N'}$ |

Und das Ergebnis wird durch die Folge der Reste $\boldsymbol{b_N'\, b_{N-1}'\, b_{N-2} \ldots b_0'}$ geliefert.

**Beispiele:**

Die Zahl $556_7$ soll ins 3-adische System (Ternärsystem) umgewandelt werden.

$556_7 : 3_7 = 164_7$ Rest 1
3
25
24
16
15
1

$164_7 : 3_7 = 43_7$ Rest 2
15
14
12
2

$43_7 : 3_7 = 13_7$ Rest 1
3
13
12
1

$13_7 : 3_7 = 3_7$ Rest 1
12
1

$3_7 : 3_7 = 1_7$ Rest 0
3
0

$1_7 : 3_7 = 0_7$ Rest 1

Ergebnis $556_7 = 101121_3$

Die Zahl $556_7$ soll ins Oktalsystem umgewandelt werden.

$556_7 : 11_7 = 50_7$ Rest 6
55
06
00
6

$50_7 : 11_7 = 4_7$ Rest 3
44
3

$4_7 : 11_7 = 0_7$ Rest 4

Ergebnis $556_7 = 436_8$

Eine zweite Möglichkeit der Konvertierung ist die Abarbeitung des Hornerschemas

$$n = ((...((\mathbf{b}_N*B + \mathbf{b}_{N-1})*B + \mathbf{b}_{N-2})*B + ... )*B + \mathbf{b}_1)*B + \mathbf{b}_0$$

von links nach rechts. In diesem Falle müssen alle $\mathbf{b_N}$ sowie die Basis B in der Formel zunächst ins B'-System umgewandelt werden. Sodann kann die Formel wie üblich nach den Rechenregeln für + und * im B'-adischen System von links nach rechts ausgerechnet werden.

**Beispiele:**

Umwandlung der Zahl $110010101_2$ ins Dezimalsystem:

n = (((((((1*2+1)*2+0)*2+0)*2+1)*2+0)*2+1)*2+0)*2+1 = 405

Umwandlung von $365_{10}$ ins Oktalsystem:

n = (3*12+6)*12+5 = (36+6)*12+5 = 44*12+5 = 550+5 = 555

Man beachte: die 10 ist im Oktalsystem als 12 dargestellt.

**Bemerkung:**

Die Konvertierung durch sukzessive Division empfiehlt sich, wenn die Arithmetik in dem Ausgangssystem einfacher ist und die Konvertierung durch Multiplikation und Addition ist einfacher, wenn die Arithmetik im Zielsystem besser beherrscht wird.

## *1.5. Andere Zahlensysteme*

### 1.5.1. Redundante Zahlensysteme

Die Darstellung einer Zahl im B-adischen Zahlensystem ist eindeutig, d.h. jede Zahl kann auf genau eine Weise dargestellt werden. Dies ist in der Regel auch erwünscht, da dies eine Rückkonvertieung ins Dezimalsystem (z.B. für die Ausgabe) erleichtert. Es gibt aber auch Zahlensysteme, die bewußt unterschiedliche Darstellungen derselben Zahl zulassen, um dadurch Vorteile bei der Arithmetik zu haben. Ein Beispiel dafür ist ein System, bei dem - wie beim Dualsystem - die einzelnen Ziffern mit dem Gewicht der Zweierpotenzen in den Wert der Zahlen eingehen, bei dem aber zusätzlich zu den Ziffern 0 und 1 noch die -1 als Ziffer zugelassen wird.

$$n = \mathbf{b}_{N-1}*B^{N-1} + \mathbf{b}_{N-2}*B^{N-2} + \mathbf{b}_{N-3}*B^{N-3} + ... + \mathbf{b}_1*B^1 + \mathbf{b}_0*B^0$$

mit $\mathbf{b_1} \in \{-1,0,1\}$.

Mit dieser Repräsentation kann die Addition von Zahlen so ausgeführt werden, dass ein Übertrag nie über einen längeren Bereich wandern muss.

**Beispiel:**

```
   -1-1 0+1 0-1-1-1 0 0+1 0+1 0-1 0+1+1+1 0 0-1 0 0+1 0+1
+   1-1-1 0-1-1-1-1 0+1+1-1-1+1-1 0 0+1+1+1-1-1-1-1 0 0 0
t   0-2-1+1-1-2-2-2 0+1+2-1 0+1-2 0+1+2+2+1-1-2-1-1+1 0+1
  ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙ ↙
c 0-1 0 0-1-1-1-1 0+1+1 0 0 0-1 0+1+1+1 0-1-1-1 0 0 0 0
r   0 0-1+1+1 0 0 0 0-1 0-1 0+1 0 0-1 0 0+1+1 0+1-1+1 0+1
s 0-1 0-1 0 0-1-1 0+1 0 0-1 0 0 0+1 0+1 0 0 0-1+1-1+1 0+1
```

**Vorteil:** Addition und Multiplikation sind schneller zu lösen als mit B-adischen Systemen. Sehr schnelle Division.

**Nachteil:** Rückkonvertierung schwierig, Zahlen nicht eindeutig dargestellt.

Anwendungen: In Spezialhardware z.B. zur Beschleunigung von Multiplikation und Division.

### 1.5.2. Restklassensysteme

Ein Restklassensystem ist ein Zahlensystem, das durch eine Menge von Moduli bestimmt ist. Die Moduli sind natürliche Zahlen, die paarweise teilerfremd sind. Sei $\{p_1, p_2, ..., p_k\}$ die Menge der Moduli und P das Produkt der $p_i$. Dann sind im zugehörigen Restklassensystem alle Zahlen von 0 bis P-1 eindeutig durch die Reste bei Division durch die $p_i$ charakterisiert.

**Beispiel:**

Menge der Moduli M =(3, 7, 11). Dann ist P = 231. Die Zahl $29_{10}$ wird dargestellt als (2,1,7), denn 29:3 hat den Rest 2, 29:7 hat den Rest 1 und 29:11 hat den Rest 7. Zwischen 0 und 230 gibt es keine andere Zahl mit diesen Resten. Weil entsprechendes für alle Zahlen gilt, ist die Darstellung eindeutig.

In jedem solchen Restklassensystem kann man sehr bequem rechnen, denn die Operationen +,- und * verhalten sich gutartig bezüglich der Reste:

$(x_1,x_2, ... , x_k) + (y_1,y_2, ..., y_k) = ((x_1+y_1) \bmod p_1, (x_2+y_2) \bmod p_2, ... , (x_k+y_k) \bmod p_k)$

$(x_1,x_2, ... , x_k) - (y_1,y_2, ..., y_k) = ((x_1-y_1) \bmod p_1, (x_2-y_2) \bmod p_2, ... , (x_k-y_k) \bmod p_k)$

$(x_1,x_2, ... , x_k) * (y_1,y_2, ..., y_k) = ((x_1*y_1) \bmod p_1, (x_2*y_2) \bmod p_2, ... , (x_k*y_k) \bmod p_k)$

**Beispiel:**

Wir betrachten das System aus dem letzten Beispiel. Zu berechnen sind $16_{10} + 13_{10}$, $16_{10} - 13_{10}$ und $16_{10} * 13_{10}$. $16_{10} = (1, 2, 5)$ und $13_{10} = (1, 6, 2)$.

$(1, 2, 5) + (1, 6, 2) = (2, 1, 7) = 29_{10}$ .

$(1, 2, 5) - (1, 6, 2) = (0, 3, 3) = 3_{10}$ .

$(1, 2, 5) * (1, 6, 2) = (1, 5, 10) = 208_{10}$ .

**Vorteile:** Addition und Multiplikation können ohne Überträge ausgeführt werden.

**Nachteil:** Division und Rückkonvertierung ist schwierig.

Anwendungen in der Kryptographie (Verschlüsselungslehre).

## *1.6. Brüche in Festkommadarstellung*

Bisher haben wir uns nur mit ganzen Zahlen beschäftigt. Aber alle Ergebnisse, insbesondere die über die B-adische Zahlendarstellung und über die Komplementbildung können auch auf gebrochene Zahlen angewendet werden. Wir ergänzen dafür die Definition der B-adischen Darstellung in folgender Weise:

$$n = \sum_{i=-M+1}^{N-1} b_i B^i$$

Zahlendarstellungen dieser Art nennt man Festkommadarstellungen, weil fest steht, daß das Komma nach der N-ten Ziffer kommt. Für die Addition und Subtraktion bleibt alles bisher Gelernte gültig.

Bei der Multiplikation von ganzen Zahlen gilt: Das Produkt zweier N-stelligen ergibt eine 2N-stellige Zahl. Da die Größe der darstellbaren Zahlen in Computern in der Regel vorgegeben ist (typisch 16, 32 oder 64 Stellen) wählen wir als Ergebnis nur wieder die niederwertigsten N Stellen. Wenn eine der höherwertigen Stellen eine Ziffer ≠ 0 enthält, ist ein Überlauf aufgetreten, der über einen gesonderten Mechanismus abzufangen ist.

Für Festkommazahlen wählt man als Bezugspunkt die Stelle, an der das Komma steht. Man wählt also vom doppeltlangen Ergebnis wiederum N Stellen vor und M Stellen nach dem Komma. Alle anderen Stellen werden verworfen. Das kann genauso zu Überläufen führen wie bei der ganzzahligen Multiplikation. Zusätzlich können aber auch Nachkommastellen ausgelöscht werden, die zu weit „hinter“ dem Komma stehen, wodurch ein Teil der Genauigkeit verloren gehen kann.

**Konversion von Festkommazahlen**

Die Konversion von Festkommazahlen in einem B-adischen System in ein B’-adisches System kann wieder durch Auflösen des Horner-Schemas geschehen.

$$n = (b_{-1}b_{-2}\ldots b_{-M})_B = ((\ldots(b_{-M}*B^{-1}+b_{-M+1})*B^{-1}+b_{-M+2})*B^{-1}+\ldots)*B^{-1}+b_{-1})*B^{-1}$$

Wir betrachten jetzt nur den gebrochenen Teil, denn wir wissen bereits wie wir den ganzzahligen Teil konvertieren können. Wenn man im B-adischen System leichter rechnen kann als im B’-adischen, empfiehlt sich folgendes Verfahren, das man „Wiederholte Multiplikation mit abschneiden“ nennt:

Man multipliziert die zu konvertierende Zahl mit der Basis B’. Die Stelle, die dabei vor das Komma gerät, ist die erste Ziffer der B’-adischen Darstellung. Diese subtrahiert man vom Ergebnis und bekommt wieder einen B-adischen Bruch mit einer 0 vor dem Komma. Mit diesem verfährt man genauso, usw.

**Beispiel:**

0,75625 soll ins Binärsystem gewandelt werden:

| | | |
|---|---|---|
| 0,75625 * 2 = 1,5125 | ⇒ | erste Ziffer ist 1. Diese abziehen ergibt 0,5125 |
| 0,51250 * 2 = 1,025 | ⇒ | nächste Ziffer ist 1. Diese abziehen ergibt 0,025 |
| 0,02500 * 2 = 0,05 | ⇒ | nächste Ziffer ist 0. Diese abziehen ergibt 0,05 |
| 0,05 * 2 = 0,1 | ⇒ | nächste Ziffer ist 0. Diese abziehen ergibt 0,1 |
| 0,1 * 2 = 0,2 | ⇒ | nächste Ziffer ist 0. Diese abziehen ergibt 0,2 |
| 0,2 * 2 = 0,4 | ⇒ | nächste Ziffer ist 0. Diese abziehen ergibt 0,4 |
| 0,4 * 2 = 0,8 | ⇒ | nächste Ziffer ist 0. Diese abziehen ergibt 0,8 |
| 0,8 * 2 = 1,6 | ⇒ | nächste Ziffer ist 1. Diese abziehen ergibt 0,6 |
| 0,6 * 2 = 1,2 | ⇒ | nächste Ziffer ist 1. Diese abziehen ergibt 0,2 |
| 0,2 * 2 = 0,4 | ⇒ | nächste Ziffer ist 0 usw. |

Die Binärdarstellung lautet also 0,11000001100110011….

Die zweite Möglichkeit besteht in der rechnerischen Auflösung des Horner-Schemas. Diese empfiehlt sich, wenn man im Zielsystem rechnen möchte. In diesem Falle konvertiert man die Basis B sowie alle Ziffern in der Quelldarstellung zunächst ins B'-adische System und rechnet dann die oben angegebene Form der Zahl durch die angegebenen Additionen und Multiplikationen im B'-adischen System aus.

**Beispiele:**

1. Die Hexadezimalzahl 0,3D5 soll ins Dezimalsystem gewandelt werden:

$$((5*16^{-1}+13)*16^{-1}+3)*16^{-1} = 0{,}239501953125$$

2. Die Dezimalzahl 100,92 soll ins Binärsystem gewandelt werden:

   100,92 (mit 7 Stellen vor und 5 Stellen nach dem Komma):

   a) 100:2=50 Rest 0
      50:2=25 Rest 0
      25:2=12 Rest 1
      12:2=6 Rest 0
      6:2=3 Rest 0
      3:2=1 Rest 1
      1:2=0 Rest 1
      100=1100100

   b) 0,92:
      0,92*2 = 1,84
      0,84*2 = 1,68
      0,68*2 = 1,36
      0,36*2 = 0,72
      0,72*2 = 1,44
      0,44*2 = 0,88 (nur erforderlich für die Rundung)
      0,92=0,11101+0=0,11101
      100,92=1100100,11101

Wenn B eine Potenz von B' ist, ist es einfach, zwischen B-adisch und B'-adisch zu konvertieren. Sei z.B. B = 16 und B' = 2. Da $16 = 2^4$ ist, bestehen alle Ziffern im B'-adischen System aus vier B'-adischen Ziffern, z.B. 4 = 0100 oder C = 1100. Auf diese Weise kann man einfach Ziffernweise konvertieren, wobei man im B'-adischen System immer Viererblöcke aus Ziffern zusammenfassen muss.

**Übung:** Wie kann man einfach vom Oktalsystem ins Hexadezimalsystem konvertieren?

## *1.7. Gleitkommadarstellung*

Im technischen und wissenschaftlichen Bereich bedient man sich neben der Festkommadarstellung von Brüchen auch der Exponentenschreibweise.

**Beispiel**

$$3456{,}5 = 0{,}34565*10^4 = 0{,}34565E+4$$

Der Vorteil ist, dass man auf kurze und übersichtliche Weise einen sehr großen Zahlenbereich darstellen kann. Diese Darstellungsweise hat auch enorme Vorteile für die Zahlendarstellung in Computern: Hier ist man durch die Hardware auf eine oder wenige Wortgrößen (Anzahl von Bits) festgelegt, in denen ein Operand untergebracht werden muss. Würde nun nur die normale Dualdarstellung verwendet, könnte man bereits die Zahl 10 Milliarden in einem 32-Bit Wort nicht mehr darstellen. Wenn man andererseits auch gebrochene Zahlen in Festkommadarstellung zulassen möchte und jeweils 16 Bit vor und 16 Bit nach dem Komma verwendet, scheitert man bereits an der Zahl 100000. Der Bereich darstellbarer Zahlen ist in diesem Falle $[-2^{15} .. +2^{15}-2^{-16}]$.

In Computern werden binäre Gleitkommaformate verwendet. In einem Register werden drei Teile einer Zahl nacheinander in einer festen Anzahl von Bits gespeichert: Das Vorzeichen V der Zahl, die Mantisse und der Exponent.

| V | Exponent | Mantisse |
|---|---|---|

Der Wert der Gleitkommazahl ist im einfachsten Falle

$$n = Vorzeichen * 0,Mantisse * 2^{Exponent}$$

wobei das Vorzeichen +1 ist, wenn V=0 ist und -1, wenn V=1 ist. Dabei wird die Mantisse als positive Dualzahl und der Exponent als Dualzahl in Zweierkomplementdarstellung interpretiert.

**Beispiel:**

Wir definieren ein fiktives binäres Gleitkommaformat mit 16 Bit: 1 Bit Vorzeichen, 10 Bit Mantisse und 5 Bit Exponent. Die Zahl

1000110110000000 bedeutet $-0{,}011_2*10_2^{11}{}_2 = -0{,}375_{10}*2_{10}^3 = -3$

0111111110000000 bedeutet $+0{,}111_2*10_2^{-1} = +0{,}875_{10}*2_{10}^{-1} = +0{,}4375$

0000011000000000 bedeutet $+0{,}1_2*10_2^1 = 1$

0000000000000000 bedeutet 0

Der darstellbare Zahlenbereich für dieses Format ist bereits

$$-2^{2^4-1} + 2^5 ... + 2^{2^4-1} - 2^5 \approx -2^{15} ... + 2^{15}$$

was mit einem Festkommaformat von 16 Bit nur dann erreicht werden kann, wenn man keine Nachkommastellen zulässt.

### 1.7.1. Arithmetik mit Gleitkommazahlen

Die Multiplikation von Gleitkommazahlen ist einfach, denn

$$m_1 * 2^{e_1} * m_2 * 2^{e_2} = m_1 * m_2 * 2^{e_1+e_2}$$

d. h. die Mantissen können als normale Dualzahlen multipliziert werden und die Exponenten addiert. Das gleiche gilt für die Division. Schwieriger ist dagegen die Addition: Hier muss zunächst dafür gesorgt werden, dass die Exponenten angeglichen werden, denn eine Addition der Mantissen kann nur dann richtig ausgeführt werden, wenn die Exponenten gleich sind:

$$m_1*2^{e_1}+m_2*2^{e_1}=(m_1+m_2)*2^{e_1}$$

Zu diesem Zweck muss zunächst ermittelt werden, welcher Exponent der größere ist und die Exponentendifferenz d wird mittels einer Subtraktion gebildet. Sodann wird die Mantisse der Zahl mit dem kleineren Exponenten um d Stellen nach rechts verschoben, wobei von links Nullen nachgezogen werden. Dies entspricht einer Division der Mantisse durch $2^d$ bei gleichzeitiger Vergrößerung des Exponenten um d. Nun kann die Addition auf den angepassten Mantissen durchgeführt werden, wobei als Exponent des Ergebnisses der größere Exponent der Operanden wird. Durch die Addition kann es passieren, dass im Ergebnis eine Folge von führenden Nullen entsteht. Um aber für nachfolgende Operationen die Genauigkeit nicht einzuschränken, wird die Mantisse des Ergebnisses nun wieder nach links verschoben, bis die erste signifikante Stelle eine 1 ist. Wenn die Verschiebedistanz d' ist, muss schließlich der Exponent noch um d' vermindert werden, damit der Wert des Ergebnisses nicht verändert wird.

**Beispiele:**

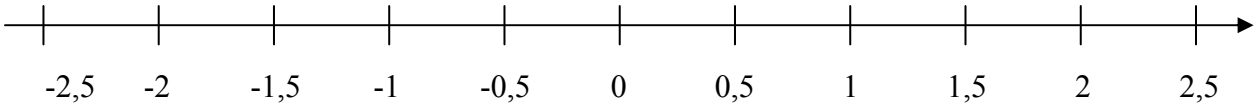
1. In unserem obigen Gleitkommaformat wollen wir 4*0,2 rechnen:
   0000111000000000 * 0111101100110011
   Mantissenmultiplikation: 0,1*0,1100110011=0,01100110011
   Exponentenaddition: 00011+11110=100001, wird interpretiert als 00001
   Normalisierung macht daraus 0,1100110011 * $2^0$.
   Ergebnis: 0000001100110011 ( = 0,799804...)

2. Was ist 1+0,1?
   0000011000000000 + 0111011100110011
   Die Exponentendifferenz ist 00001 - 11101 = 00100 $(4)_{10}$
   Die Mantisse mit dem größeren Exponenten ist 1000000000
   Die andere Mantisse um 4 Stellen verschoben ist 0000110011
   Die Mantissensumme ist 1000110011
   und wir bekommen als Ergebnis 0000011000110011 (=1,099609...)

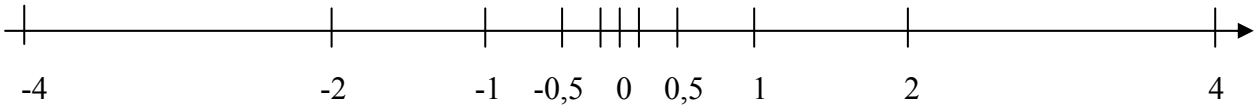
**Bemerkungen:**

Mit Gleitkommazahlen kann man eine größeren Zahlenbereich abdecken als mit Festkommazahlen mit gleich vielen Stellen. Während beim Festkommaformat zwischen je zwei benachbarten Zahlen der gleiche Abstand liegt, variiert dieser bei Gleitkommazahlen. In der Gegend der Null liegen sie sehr dicht, während sie zu den Grenzen des darstellbaren Zahlenbereichs hin immer weiter auseinanderliegen:

Festkommazahlen:




Gleitkommazahlen:




Multiplikation von Gleitkommazahlen ist aufwendiger als von Festkommazahlen, Addition von Gleitkommazahlen ist viel aufwendiger als bei Festkommazahlen.

### 1.7.2. Gebräuchliche Gleitkommaformate

Die IEEE hat für Rechenanlagen drei Gleitkommaformate spezifiziert, die wir in den gängigen Programmiersprachen wiederfinden:

**32 Bit (float, single):**

1 Vorzeichenbit

8 Exponentenbits (MSB first)

23 Mantissenbits (MSB first)

Der Wert w einer in diesem Format dargestellten Zahl berechnet sich als:

$w=(-1)^V*(1,M)*2^{E-127}$, falls $E>0$ und $E<255$

$w=(-1)^V*(0,M)*2^{-126}$, falls $E=0$ und $M\neq0$

$w=(-1)^V*0$, falls $E=0$ und $M=0$

$w=(-1)^V*$Infinity ($\infty$), falls $E=255$ und $M=0$

w=NaN (not a number), falls $E=255$ und $M\neq0$

Darstellbarer Bereich ca. $[-10^{41}...10^{41}]$

**64 Bit (double):**

1 Vorzeichenbit

11 Exponentenbits (MSB first)

52 Mantissenbits (MSB first)

Der Wert w einer in diesem Format dargestellten Zahl berechnet sich als:

$w=(-1)^V*(1,M)*2^{E-1023}$, falls E>0 und E<2047

$w=(-1)^V*(0,M)*2^{-1022}$, falls E=0 und M≠0

$w=(-1)^V*0$, falls E=0 und M=0

$w=(-1)^V*$Infinity (∞), falls E=2047 und M=0

w=NaN (not a number), falls E=2047 und M≠0

Darstellbarer Bereich ca. $[-10^{300}...10^{300}]$

**80 Bit (extended):**

1 Vorzeichenbit

15 Exponentenbits (MSB first)

64 Mantissenbits (MSB first)

Der Wert w einer in diesem Format dargestellten Zahl berechnet sich als:

$w=(-1)^V*(1,M)*2^{E-16383}$, falls E>0 und E<32767

$w=(-1)^v*$Infinity (∞), falls E=32767 und M=0

w=NaN (not a number), falls E=32767 und M≠0

Darstellbarer Bereich ca. $[-10^{5000}...10^{5000}]$

**Dabei ist zu beachten:**

Die Zahlen sind in normalisierter Form, d.h. vor dem Komma steht eine 1. Diese führende 1 wird nicht mit dargestellt (hidden one).

Der Exponent ist mit einem Offset versehen. Bei Single-Zahlen 127, bei Double-Zahlen 1023 und bei Extended-Zahlen 32767. D. h. die dargestellten Exponenten sind um diesen Offset zu vermindern um ihre „normalen" Dualdarstellungen zu bekommen. Man erreicht dadurch, daß keine negativen Exponenten auftauchen können.

**Folgende Sonderfälle werden unterschieden:**

Eine 0 wird dargestellt durch Vorzeichen=0, Mantisse = 000...0, Exponent = 000...0.

Eine ∞ wird dargestellt durch Mantisse = 000...0, Exponent = 111...1.

Eine nicht darstellbare Zahl (NaN, not a number) wird dargestellt als Mantisse ≠ 0 und Exponent = 111...1.

Wenn der Exponent 000...0 ist und die Mantisse ≠ 000...0, so wird die versteckte 1 nicht einkopiert.

### *1.8. Codierung von Zeichen in Binärdarstellung*

Bisher haben wir uns mit der Codierung von Zahlen (z.B. Typ Integer oder Typ float) beschäftigt. Es werden aber auch andere Codierungen benutzt. Die folgende Tabelle zeigt verschiedene Codes für die Ziffern von 0 bis 9, die in Rechenanlagen genutzt werden:

| **Dezimalziffer** | **Dual (8421)** | **Aiken** | **3-Exzess** | **2aus5Code** |
|---|---|---|---|---|
| 0 | 0000 | 0000 | 0011 | 11000 |
| 1 | 0001 | 0001 | 0100 | 00011 |
| 2 | 0010 | 0010 | 0101 | 00101 |
| 3 | 0011 | 0011 | 0110 | 00110 |
| 4 | 0100 | 0100 | 0111 | 01001 |
| 5 | 0101 | 1011 | 1000 | 01010 |
| 6 | 0110 | 1100 | 1001 | 01100 |
| 7 | 0111 | 1101 | 1010 | 10001 |
| 8 | 1000 | 1110 | 1011 | 10010 |
| 9 | 1001 | 1111 | 1100 | 10100 |
| Gewichte | 8421 | 2421 | keine | 74210 |

Auch mit dieser Codierung kann man aber nur numerische Werte verarbeiten. Um nun alle Zeichen, also auch die Buchstaben und Sonderzeichen in Rechenanlagen verarbeiten zu können, haben sich zwei Standards durchgesetzt, EBCDIC und ASCII. Diese sind auf den folgenden Tabellen dargestellt.

| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |
| 0 | 0000 | | | | | | | | | | | | | | | | |
| 1 | 0001 | | | | | | | | | | | | | | | | |
| 2 | 0010 | | | | | | | | | | | | | | | | |
| 3 | 0011 | | | | | | | | | | | | | | | | |
| 4 | 0100 | blank | | | | | | | | | | § | . | < | ( | + | \| |
| 5 | 0101 | & | | | | | | | | | | ! | $ | • | ) | ; | |
| 6 | 0110 | - | / | | | | | | | | | ∧ | , | % | | > | ? |
| 7 | 0111 | | | | | | | | | | | : | # | @ | ´ | * | “ |
| 8 | 1000 | | a | b | c | d | e | f | g | h | i | | | | | | |
| 9 | 1001 | | j | k | l | m | n | o | p | q | r | | | | | | |
| A | 1010 | | | s | t | u | v | w | x | y | z | | | | | | |
| B | 1011 | | | | | | | | | | | | | | | | |
| C | 1100 | | A | B | C | D | E | F | G | H | I | | | | | | |
| D | 1101 | | J | K | L | M | N | O | P | Q | R | | | | | | |
| E | 1110 | | | S | T | U | V | W | X | Y | Z | | | | | | |
| F | 1111 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | | | | |

# EBCDIC**(Extended Binary Coded Decimal Interchange Code)**

# ASCII

## (American Standard Code for Information Interchange)

| | Hex | | | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Hex** | | | | | **000** | **001** | **010** | **011** | **100** | **101** | **110** | **111** |
| **0** | **0** | **0** | **0** | **0** | NUL | DLE | SP | 0 | @ | P | ` | p |
| **1** | **0** | **0** | **0** | **1** | SOH | DC1 | ! | 1 | A | Q | a | q |
| **2** | **0** | **0** | **1** | **0** | STX | DC2 | “ | 2 | B | R | b | r |
| **3** | **0** | **0** | **1** | **1** | ETX | DC3 | # | 3 | C | S | c | s |
| **4** | **0** | **1** | **0** | **0** | EOT | DC4 | $ | 4 | D | T | d | t |
| **5** | **0** | **1** | **0** | **1** | ENQ | NAK | % | 5 | E | U | e | u |
| **6** | **0** | **1** | **1** | **0** | ACK | SYN | & | 6 | F | V | f | v |
| **7** | **0** | **1** | **1** | **1** | BEL | ETB | ´ | 7 | G | W | g | w |
| **8** | **1** | **0** | **0** | **0** | BS | CAN | ( | 8 | H | X | h | x |
| **9** | **1** | **0** | **0** | **1** | SKIP | EM | ) | 9 | I | Y | i | y |
| **A** | **1** | **0** | **1** | **0** | LF | SUB | * | : | J | Z | j | z |
| **B** | **1** | **0** | **1** | **1** | VT | ESC | + | ; | K | [ | k | { |
| **C** | **1** | **1** | **0** | **0** | FF | FS | , | < | L | \ | l | \| |
| **D** | **1** | **1** | **0** | **1** | CR | GS | - | = | M | ] | m | } |
| **E** | **1** | **1** | **1** | **0** | SO | HOME | . | > | N | ∧ | n | ~ |
| **F** | **1** | **1** | **1** | **1** | SI | NL | / | ? | O | _ | o | DEL |

ASCII verwendet nur 7 bits für die eigentliche Codierung. Das 8-te Bit eines Bytes (in dem jedes Zeichen codiert wird) wird als Paritätsstelle zur Fehlererkennung benutzt. Dieses achte Bit wird immer so gesetzt, dass die Anzahl der 1en im Byte gerade ist. Auf diese Weise ist bei einer Übertragung jeder Fehler zu erkennen, bei dem genau ein Bit (oder eine ungerade Anzahl) innerhalb des Bytes verfälscht wurde.

# 2. Boolesche Funktionen und Schaltnetze

## *2.1. Grundlegende Begriffe*

**Definition:** Sei $\mathbb{B} = \{0,1\}$ und n und m natürliche Zahlen. Eine boolesche Funktion f mit n Eingabevariablen und m Ausgabevariablen

$$f: B^n \rightarrow B^m$$

ordnet jedem n-stelligen Vektor $(x_0,x_1,x_2,...,x_{n-1})$ von Eingabevariablen einen m-stelligen Vektor $(y_0, y_1, y_2, ..., y_{m-1})$ von Ausgabevariablen zu.

Wir bezeichnen $X = (x_0, x_1, x_2, ..., x_{n-1})$ als (n-stelliges) Eingabewort und $Y = (y_0, y_1, y_2, ..., y_{m-1})$ als (m-stelliges) Ausgabewort über $\mathbb{B}$ (und verwenden gelegentlich auch die Schreibweise $X = x_0x_1x_2 ...x_{n-1}$ und $Y = y_0y_1y_2...y_{m-1}$).

Ein Schaltnetz ist eine technische Realisierung einer solchen booleschen Funktion. Man kann es sich als einen schwarzen Kasten vorstellen, der n Eingänge und m Ausgänge hat. Diese Eingänge könnten zum Beispiel Schalter sein, die zwei mögliche Stellungen haben und die Ausgänge könnten kleine Lämpchen sein, die an oder aus sein können. Eine andere, unseren Anwendungen sehr schön entsprechende Vorstellung ist ein elektronisches Bauteil, in das n Leitungen hereinführen und m heraus. Ein Eingabewort ist in diesem Falle eine Belegung jeder Eingabeleitung mit einem von zwei unterschiedlichen Spannungspotentialen, z.B. 0 = 0V und 1 = 3,3V. Ein Ausgabewort entspricht einer ebensolchen Belegung für alle m Ausgabeleitungen. Diese Vorstellung hat den Vorteil, dass man Schaltnetze zusammenschalten, d.h. die Ausgaben des ersten mit den Eingaben des zweiten identifizieren kann.

Jede Komponente des Eingabewortes bezeichnen wir als Eingabevariable und jede Komponente des Ausgabewortes als Ausgabevariable.

Beispiele für Schaltnetze werden wir in den folgenden Abschnitten kennenlernen.

## *2.2. Beschreibungsformen für boolesche Funktionen und Schaltnetze*

Sie kennen bei reellwertigen Funktionen unterschiedliche Beschreibungsformen. Neben einer Wertetabelle können Sie eine Funktion als Formel oder z.B. als Graph beschreiben.

Ebenso gibt es für boolsche Funktionen gängige Beschreibungsformen, die wir hier kennenlernen wollen.

### 2.2.1. Wertetabelle

Die Wertetabelle einer booleschen Funktion gibt die Zuordnung der Belegung der Eingabevariablen zu der Belegung der Ausgabevariablen explizit an. Für jede mögliche Belegung der Eingabevariablen ist eine Zeile vorhanden, die sagt: für dieses Eingabewort ist der Funktionswert dieses Ausgabewort. Wegen der (häufig verwendeten) Interpretation der Werte von $\mathbb{B}$ als True (= 1) und False (= 0) wird die Wertetabelle in der Literatur auch oft als Wahrheitstabelle bezeichnet.

**Beispiel:**

| Eingabevariablen | | | Ausgabevariablen | |
|---|---|---|---|---|
| X | | | Y | |
| $x_0$ | $x_1$ | $x_2$ | $y_0$ | $Y_1$ |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Die Wertetabelle hat den Nachteil, daß sie u.U. sehr groß werden kann, z.B. wenn für 10 Eingabevariablen $2^{10} = 1024$ Zeilen erforderlich sind.

### 2.2.2. Formel

Wie bei reellwertigen Funktionen kann eine boolesche Funktion (und somit das entsprechende Schaltnetz) auch als Formel beschrieben werden. Dabei werden die Variablen durch Operatoren miteinander verknüpft. Dies sind nun aber nicht arithmetische Operatoren wie + und *, sondern boolesche (logische) Operatoren wie „und“ (als Zeichen $\wedge$), „oder“ (als Zeichen $\vee$) und „nicht“ (als Zeichen ein Querstrich über der Variablen). Die in der obigen Wahrheitstabelle spezifizierte boolesche Funktion könnte als Formel so geschrieben werden:

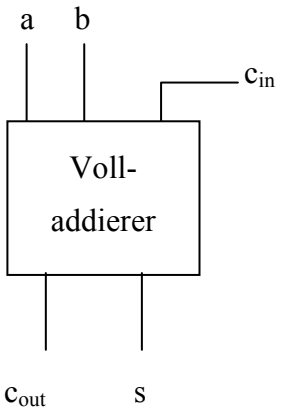
$$y_0 = \overline{x_0}\,\overline{x_1}\,x_2 + \overline{x_0}\,x_1\,\overline{x_2} + x_0\,\overline{x_1}\,\overline{x_2} + x_0\,x_1\,x_2$$

$$y_1 = x_1\,x_2 + x_0\,x_2 + x_0\,x_1$$

Was genau „und“, „oder“ und „nicht“ bedeutet, werden wir in den folgenden Abschnitten lernen.

### 2.2.3. Symbol

Das oben verwendete Beispiel stellt einen Volladdierer dar, ein elementares Schaltnetz, das verwendet werden kann, um binäre Zahlen miteinander zu addieren. Als Schaltsymbol kann er folgendermaßen angegeben werden.

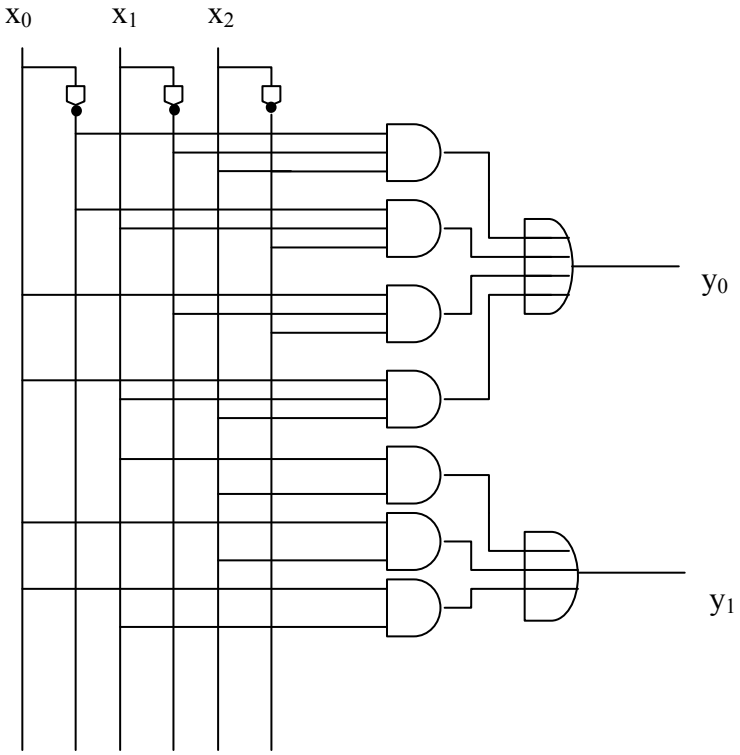



Wobei a, b, $c_{in}$ drei 1-Bit Ein- und s, $c_{out}$ zwei 1-Bit Ausgänge sind.

Neben den Ein- und Ausgabevariablen wird durch das Pluszeichen seine Funktion beschrieben.

### 2.2.4. Netz von Elementarschaltungen

Die Auflösung in Elementarschaltungen, die die technisch realisierbaren atomaren Bausteine eines Schaltnetzes darstellen, nennt man Realisierung.



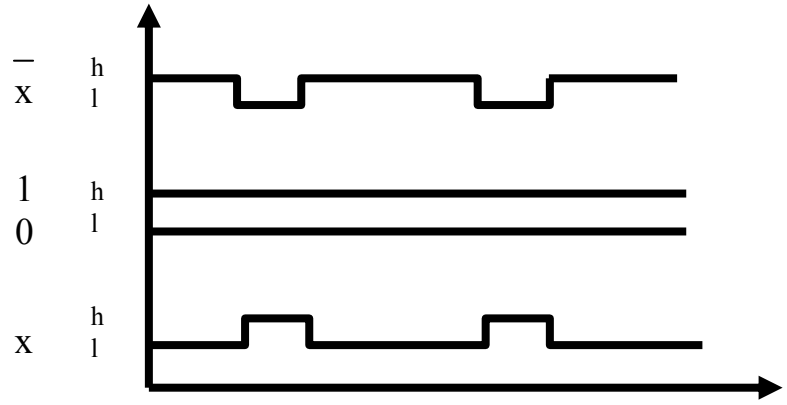

Das Bild zeigt die Realisierung des Volladdierers auf der Basis in der oben angegebenen Formel. Die verwendeten Bauteile sind sogenannte „und“- , „oder“- und „nicht“-Gatter, deren Funktion im Abschnitt 2.5 erklärt wird.

### 2.3. Boolesche Funktionen mit einer Ein- und einer Ausgabevariablen

Es gibt vier verschiedene boolesche Funktionen mit einer Eingabevariablen $x_0$. Das ergibt sich dadurch, dass diese Eingabevariable zwei Werte haben kann und es $2^2 = 4$ Möglichkeiten gibt, diese beiden Werte auf die beiden möglichen Werte der Ausgangsvariablen abzubilden.

| Benennung | $x_0 = 0$ | $x_0 = 1$ | Algebraische Darstellung |
|---|---|---|---|
| Nullfunktion | 0 | 0 | 0 |
| Identität | 0 | 1 | $x_0$ |
| Negation | 1 | 0 | $\overline{x_0}$ |
| Einsfunktion | 1 | 1 | 1 |

Das Ein/Ausgabeverhalten von Booleschen Funktionen kann man anhand von Impulsdiagrammen studieren. Hierbei wird auf der waagerechten Achse die Zeit abgetragen und auf der senkrechten die Werte der in Diskussion stehenden Funktionen. Um die Verläufe der Funktionen unterscheiden zu können, werden häufig mehrere unterschiedliche Funktionen übereinander in dasselbe Diagramm gezeichnet.




### 2.4. Boolesche Funktionen mit zwei Ein- und einer Ausgabevariablen

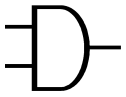
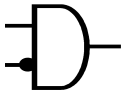
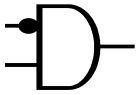
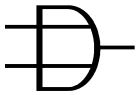
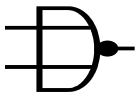
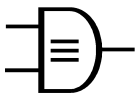
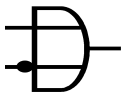
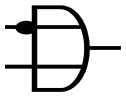
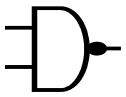
Wenn wir k Eingabevariablen und eine Ausgabevariable haben, gibt es $2^{2^k}$ Möglichkeiten, die $2^k$ möglichen Belegungen der Eingabevariablen auf zwei Werte abzubilden. Es gibt daher 16 verschiedene boolesche Funktionen mit zwei Eingabevariablen und einer Ausgabevariablen. Diese sind in der folgenden Tabelle aufgeführt.

Eingabevariablen: $x_0$,$x_1$

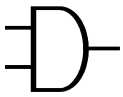
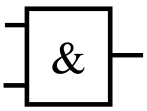
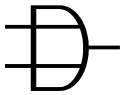
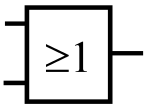
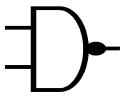
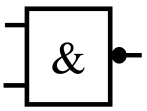
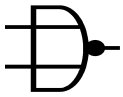
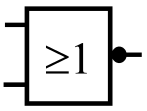
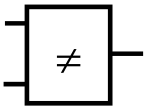
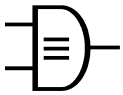
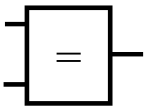
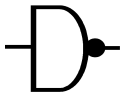
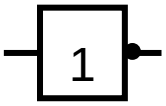
Ausgabevariable: y

Die Schaltsymbole entsprechen der alten (aber noch heute sehr gebräuchlichen) europäischen Standardisierung. Die wichtigsten Symbole sind in der darauf folgenden Tabelle noch einmal dargestellt, zusammen mit den entsprechenden Symbolen nach neuem internationalen Standard.

| **Benennung** | | **Algebraische** | |
|---|---|---|---|

| | | | | | **Darstellung** | |
|---|---|---|---|---|---|---|
| $x_1$ | 0 | 0 | 1 | 1 | | |
| $x_0$ | 0 | 1 | 0 | 1 | | |
| Nullfunktion | 0 | 0 | 0 | 0 | 0 | 0 |
| UND (Konjunktion) | 0 | 0 | 0 | 1 | $x_0 x_1$ |  |
| (UND) | 0 | 0 | 1 | 0 | $\overline{x_0} x_1$ |  |
| (Identität) | 0 | 0 | 1 | 1 | $x_1$ | $X_1$ |
| (UND) | 0 | 1 | 0 | 0 | $x_0 \overline{x_1}$ |  |
| (Identität) | 0 | 1 | 0 | 1 | $x_0$ | $X_0$ |
| XOR (Antivalenz) | 0 | 1 | 1 | 0 | $x_0 \neq x_1$ |  |
| ODER (Disjunktion) | 0 | 1 | 1 | 1 | $x_0 + x_1$ |  |
| NOR (Peirce-Fkt.) | 1 | 0 | 0 | 0 | $\overline{x_0 + x_1}$ |  |
| XNOR (Äquivalenz) | 1 | 0 | 0 | 1 | $x_0 \equiv x_1$ |  |
| (Negation) | 1 | 0 | 1 | 0 | $\overline{x_0}$ |  |
| Implikation | 1 | 0 | 1 | 1 | $\overline{x_0} + x_1$ |  |
| (Negation) | 1 | 1 | 0 | 0 | $\overline{x_1}$ |  |
| (Implikation) | 1 | 1 | 0 | 1 | $x_0 + \overline{x_1}$ |  |
| NAND (Sheffer-Fkt.) | 1 | 1 | 1 | 0 | $\overline{x_0 x_1}$ |  |
| Einsfunktion | 1 | 1 | 1 | 1 | 1 | 1 |

Alte und neue Symbole:

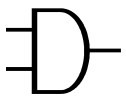
| Bezeichnung: | Alte Symbole: | Neue Symbole: |
|---|---|---|
| UND |  |  |
| ODER |  |  |
| NAND |  |  |
| NOR |  |  |
| XOR |  |  |
| XNOR |  |  |
| NOT |  |  |

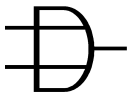
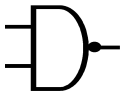
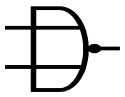
### *2.5. Gatter und Leitungen als Elementarbausteine von Schaltnetzen*

**Definition:**

Ein *Schaltnetz* ist eine technische Realisierung einer booleschen Funktion. Schaltnetze können durch Verbindung von Gattern und Leitungen aufgebaut werden.

Das einfachste Schaltnetz ist eine Leitung. Sie hat einen Eingang und einen Ausgang und sie repräsentiert die Identitätsfunktion, denn der Ausgang ist immer gleich dem Eingang. Mittlerweile kennen wir aber weitere Elementarbausteine, die komplexere Boolesche Funktionen realisieren. Zu diesen zählen das Nicht-Gatter (Inverter, not-gate, Negation) deren Ausgang immer das Inverse des Eingangs ist, das Und-Gatter (and-gate, Konjunktion), dessen Ausgang genau dann 1 ist, wenn alle Eingänge 1 sind und das Oder-Gatter (or-gate, Disjunktion), das eine 1 liefert, wenn mindestens einer der Eingänge 1 ist. Eine Übersicht der wichtigsten Gatter mit zwei Eingängen mit den gebräuchlichsten Schaltsymbolen findet sich in der folgenden Tabelle.

| | |
|---|---|
| AND (UND) |  |

| | |
|---|---|
| OR (ODER) |  |
| NAND (Nicht AND) |  |
| NOR (Nicht ODER) |  |

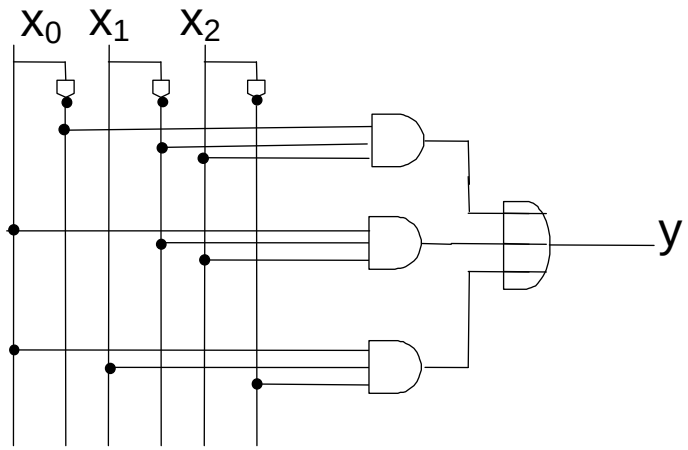
Als Zeichen für „oder“ ist $\vee$ oder das Pluszeichen + gebräuchlich. Für „und“ wird $\wedge$ und das Malzeichen $\cdot$ benutzt. Letzteres wird häufig (wie in der Mathematik) weggelassen, wenn klar ist, dass es sich um eine boolesche Verknüpfung zwischen zwei Variablen handelt.

### 2.5.1. Minterme und kanonische disjunktive Normalform

Boole’sche Funktionen können in der ersten Stufe bestimmte Eingangsvariablen oder deren Inverse mit und-Operationen durch eine Oder-Operation zusammengefügt werden. Eine solche zweistufige Form nennt man „disjunktive Normalform“.

**Definition:** Produktterm, DNF;

Beispiel:




Die folgende Tabelle zeigt eine zu realisierende Funktion mit drei Eingabevariablen. Genau die Kombinationen der Eingabewerte, für die der Ausgang 1 ist, entsprechen einem *Minterm*. Ein Minterm (Vollkonjunktion, Minimaler konjunktiver Term) ist eine konjunktive Verknüpfung, in der alle Eingabevariablen genau einmal entweder in invertierter oder in nichtinvertierter Form auftauchen. Jede Zeile der Wertetabelle entspricht einem Minterm. Die Mintermfunktionen, für die Minterme, deren Wert 1 ist, sind in der Tabelle mit $M_0$ bis $M_2$ bezeichnet.

| **Eingangsvariablen** | | | **Negationen** | | | **Mintermfunktionen** | **Ausgangsvariable** |
|---|---|---|---|---|---|---|---|
| $x_0$ | $x_1$ | $x_2$ | $\overline{x}_0$ | $\overline{x}_1$ | $\overline{x}_2$ | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | $M_0 = \overline{x}_0 \cdot \overline{x}_1 \cdot x_2$ | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | $M_1 = x_0 \cdot \overline{x}_1 \cdot x_2$ | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | $M_2 = x_0 \cdot x_1 \cdot \overline{x}_2$ | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | | 0 |

Mintermfunktionen:

$$M_0 = \overline{x}_0 \cdot \overline{x}_1 \cdot x_2$$

$$M_1 = x_0 \cdot \overline{x}_1 \cdot x_2$$

$$M_2 = x_0 \cdot x_1 \cdot \overline{x}_2$$

**Definition:**

Ein *Minterm* einer booleschen Funktion ist eine Konjunktion (Und-Verknüpfung) aller Eingangsvariablen, wobei jede genau einmal entweder in nicht invertierter oder in invertierter Form auftritt.

Die Ausgangsfunktion kann man nun realisieren, indem man die Minterme, deren Wert 1 ist, durch ein großes Oder-Gatter miteinander verknüpft.

$$y = M_0 + M_1 + M_2 = \overline{x}_0 \overline{x}_1 x_2 + x_0 \overline{x}_1 x_2 + x_0 x_1 \overline{x}_2$$

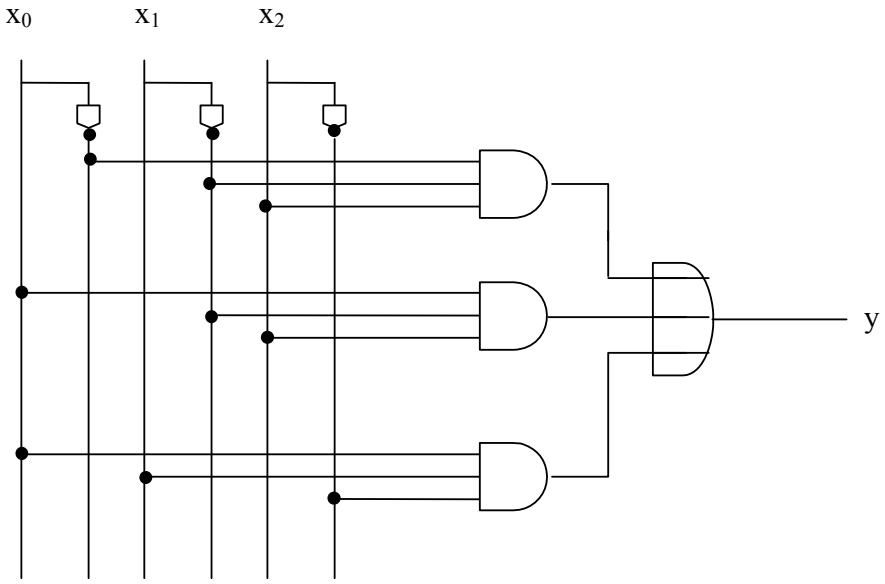
Die Oder-Verknüpfung sorgt dafür, dass jede eins, die durch eine der Mintermfunktionen erzeugt wird, auch an den Ausgang für y gelangt.

Diese Form der Oder-Verknüpfung aller Minterme, für die die Ausgangsfunktion den Wert eins annimmt, wird *disjunktive Normalform* (DNF) genannt.

**Definition:**

Die *kanonische disjunktive Normalform* (KDNF) einer booleschen Funktion ist die Disjunktion (Oder-Verknüpfung) aller Minterme, die für diese Funktion den Wert logisch 1 annehmen.

Das folgende Bild zeigt die schaltungstechnische Realisierung der kanonischen disjunktiven Normalform der Funktion aus dem obigen Beispiel unter Verwendung von Invertern und Und- und Oder-Gattern.




### 2.5.2. Maxterme und kanonische konjunktive Normalform

Neben der kanonischen disjunktiven Normalform, die aus Mintermen aufgebaut ist, gibt es die duale Möglichkeit, eine Schaltfunktion aus *Maxtermen* aufzubauen. Wir verwenden als Beispiel wieder die Tabelle für eine Funktion, deren Ausgang auf eins ist, wenn genau zwei ihrer Eingänge auf eins sind. Die Maxterme werden als Oder-Verknüpfung der invertierten Eingänge gebildet, für die die Ausgangsfunktion 0 ist.

| **Eingangsvariablen** | | | **Negationen** | | | **Maxtermfunktionen** | **Ausgangsvariable** |
|---|---|---|---|---|---|---|---|
| $x_0$ | $x_1$ | $x_2$ | $\overline{x}_0$ | $\overline{x}_1$ | $\overline{x}_2$ | | |
| 0 | 0 | 0 | 1 | 1 | 1 | $M_0 = x_0 + x_1 + x_2$ | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | $M_1 = x_0 + \overline{x}_1 + x_2$ | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | $M_2 = x_0 + \overline{x}_1 + \overline{x}_2$ | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | $M_3 = \overline{x}_0 + x_1 + x_2$ | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | $M_4 = \overline{x}_0 + \overline{x}_1 + \overline{x}_2$ | 0 |

Maxtermfunktionen:

$$M_0 = x_0 + x_1 + x_2$$

$$M_1 = x_0 + \overline{x}_1 + x_2$$

$$M_2 = x_0 + \overline{x}_1 + \overline{x}_2$$

$$M_3 = \overline{x}_0 + x_1 + x_2$$

$$M_4 = \overline{x}_0 + \overline{x}_1 + \overline{x}_2$$

**Definition:**

Ein *Maxterm* einer booleschen Funktion ist eine Disjunktion (Oder-Verknüpfung) aller Eingangsvariablen, wobei jede genau einmal entweder in nicht invertierter oder in invertierter Form auftritt.

Die Ausgangsfunktion kann man nun realisieren, indem man die Maxterme durch ein großes Und-Gatter miteinander verknüpft.

$$y = M_0 \cdot M_1 \cdot M_2 \cdot M_3 \cdot M_4$$

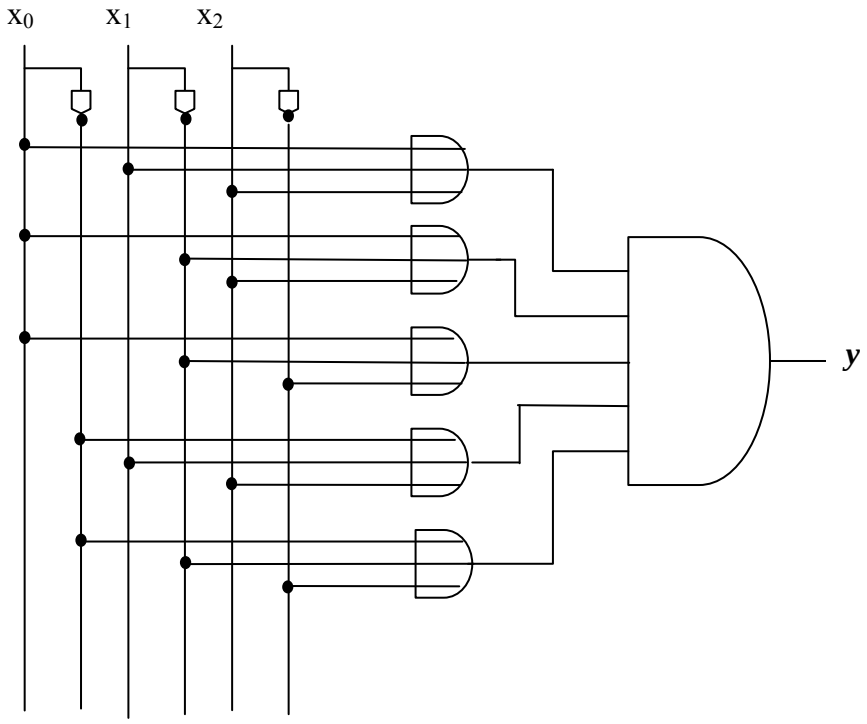
$$= (x_0 + x_1 + x_2)(x_0 + \overline{x_1} + x_2)(x_0 + \overline{x_1} + \overline{x_2})(\overline{x_0} + x_1 + x_2)(\overline{x_0} + \overline{x_1} + \overline{x_2})$$

**Definition:**

Die *kanonische konjunktive Normalform* (KKNF) einer booleschen Funktion ist die Konjunktion (Und-Verknüpfung) aller Maxterme, die für diese Funktion den Wert logisch 0 annehmen.

Die Und-Verknüpfung sorgt dafür, daß jede Null, die durch eine der Maxtermfunktionen erzeugt wird, auch an den Ausgang für y gelangt. Nur wenn keiner der Maxterme eine Null produziert, kann die Ausgangsfunktion Eins werden.

Das folgende Bild zeigt die schaltungstechnische Realisierung der konjunktiven Normalform der Funktion aus dem obigen Beispiel unter Verwendung von Invertern und Und- und Oder-Gattern.



## 2.6. Boolesche Algebra

Kanonische disjunktive und konjunktive Normalform sind im Allgemeinen aufwendige Realisierungen von Schaltfunktionen. Der Begriff *aufwendig* kann konkretisiert werden anhand von Kostenmaßen. Diese sind in der Regel abhängig von der Schaltungstechnologie die zur Realisierung benutzt wird. Zwei Maße gelten jedoch offensichtlich für Realisierungen in allen Schaltkreisfamilien: Die Anzahl der Gatter und die Anzahl der Eingänge pro Gatter. Der Schaltkreisentwerfer sollte daher stets bestrebt sein, seine Schaltung in Hinsicht auf diese beiden Maße zu minimieren.

Als zu Grunde liegende Theorie zu diesem Gebiet hat sich die Boolesche Algebra (Schaltalgebra) zu einem nützlichen Werkzeug entwickelt. Sie stellt einen Satz Regeln und Verfahren zur Verfügung, mit denen eine Schaltung systematisch in Hinsicht auf die Anzahl der Gatter und die Anzahl der Eingänge pro Gatter optimiert werden kann.

Der Booleschen Algebra liegen die folgenden vier Axiome zugrunde, die für alle Booleschen Funktionen mit Variablen x, $x_0$, $x_1$, $x_2$ und Operatoren + (oder) und · (und) gelten:

1. Kommutativgesetze:

$$\forall x_0, x_1 : x_0 + x_1 = x_1 + x_0$$

$$\forall x_0, x_1 : x_0 \cdot x_1 = x_1 \cdot x_0$$

2. Neutrale Elemente:

$$\forall x : 0 + x = x$$

$$\forall x : 1 \cdot x = x$$

3. Distributivgesetze:

$$\forall x_0, x_1, x_2 : (x_0 + x_1)\, x_2 = x_0\, x_2 + x_1\, x_2$$

$$\forall x_0, x_1, x_2 : (x_0\, x_1) + x_2 = (x_0 + x_2)(x_1 + x_2)$$

4. Inverses Element:

$$\forall x \; \exists \overline{x} : \quad \overline{x} + x = 1$$

$$\forall x \; \exists \overline{x} : \quad \overline{x} \cdot x = 0$$

Aus diesen vier Axiomen laßen sich eine Reihe von Aussagen ableiten:

**Satz 1:**

Für jede Eingabevariable $x$ gilt: $x + 1 = 1$

Beweis:

$$x + 1 = 1 \cdot (x + 1) = (x + \overline{x}) \cdot (x + 1) = (\overline{x} + x) \cdot (1 + x) = (\overline{x} \cdot 1) + x = \overline{x} + x = 1$$

**Satz 2:**

Für jede Eingabevariable $x$ gilt: $x \cdot 0 = 0$

**Satz 3:**

Für jede Eingabevariable $x$ gilt: $x + x = x$

Beweis:

$$x + x = (1 \cdot x) + (1 \cdot x) = (1+1) \cdot x = 1 \cdot x = x$$

**Satz 4:**

Für jede Eingabevariable $x$ gilt: $x \cdot x = x$

Beweis:

$$x \cdot x = (0 + x) \cdot (0 + x) = (0 \cdot 0) + x = 0 + x = x$$

Assoziativgesetze:

**Satz 5:**

$$\forall x_0, x_1, x_2 : (x_0\, x_1)\, x_2 = x_0 (x_1\, x_2)$$

Beweis:

$$x_0 + x_0 \cdot x_1 = x_0 \cdot 1 + x_0 \cdot x_1 = x_0 \cdot (\overline{x_1} + x_1) + x_0 \cdot x_1 = (x_0 \cdot \overline{x_1} + x_0 \cdot x_1) + x_0 \cdot x_1 =$$
$$= x_0 \cdot \overline{x_1} + (x_0 \cdot x_1 + x_0 \cdot x_1) = x_0 \cdot \overline{x_1} + x_0 \cdot x_1 = x_0 \cdot (\overline{x_1} + x_1) = x_0 \cdot 1 = x_0$$

**Satz 6:**

$$\forall x_0, x_1, x_2 : (x_0 + x_1) + x_2 = x_0 + (x_1 + x_2)$$

**Satz 7:**

$$\forall x : \overline{\overline{x}} = x$$

$\overline{0} = 1$ und $\overline{1} = 0$

**Satz 8:**

Für jede Eingabevariable $x_0$, $x_1$ gilt:

$$x_0 + (x_0\, x_1) = x_0$$

**Satz 9:**

Für jede Eingabevariable $x_0$, $x_1$ gilt:

$$x_0 (x_0 + x_1) = x_0$$

**Beweis:**

Solange wir diese Sätze nur über B betrachten, kann man die „Beweise“ auch über das Ausprobieren aller Möglichkeiten führen. Ein Beispiel dafür sieht so aus.

| $x_0$ | $x_1$ | $x_0x_1$ | $x_0+(x_0x_1)$ | $x_0+x_1$ | $x_0(x_0+x_1)$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | **0** | 0 | **0** |
| 0 | 1 | 0 | **0** | 1 | **0** |
| 1 | 0 | 0 | **1** | 1 | **1** |
| 1 | 1 | 1 | **1** | 1 | **1** |

Führen Sie die Beweise zu den anderen Sätzen als Übungsaufgabe.

**Satz 10:**

Für jede Eingabevariable x gilt:

$$\overline{\overline{x}} = x$$

**Satz 11:**

Es gilt: $\overline{0} = 1$ und $\overline{1} = 0$.

**Satz 12:** (Erstes De Morgansches Gesetz):

Für jede Eingabevariable $x_0$, $x_1$ gilt: $\overline{x}_0 + \overline{x}_1 = \overline{x_0 \cdot x_1}$

**Satz 13:** (Zweites De Morgansches Gesetz):

Für jede Eingabevariable $x_0$, $x_1$ gilt: $\overline{x}_0 \cdot \overline{x}_1 = \overline{x_0 + x_1}$

**Satz 14:** (Erste Vereinfachungsregel):

Für jede Eingabevariable $x_0$, $x_1$ gilt: $(x_0 + x_1)(x_0 + \overline{x_1}) = x_0$

**Satz 15:** (Zweite Vereinfachungsregel):

Für jede Eingabevariable $x_0$, $x_1$ gilt: $(x_0 x_1) + (x_0 \overline{x_1}) = x_0$

**Übungsaufgabe:** errechnen der KKNF aus der KDNF mit Hilfe der Gesetze von De Morgan.

### 2.6.1. Minimierung von Booleschen Funktionen mit Mitteln der Booleschen Algebra

Durch Anwendung der Vereinfachungsregeln gelingt es, aus der kanonischen disjunktiven Normalform eine äquivalente minimale Darstellung der Ausgabefunktion zu bestimmen. Diese minimale Darstellung heißt Disjunktive Minimalform (DMF).

**Definition:**

Eine aus einer Disjunktion von Produkttermen bestehende Boolesche Funktion f heißt *disjunktive Minimalform*, wenn

- jede äquivalente Darstellung derselben Ausgabefunktion mindestens genauso viele Produktterme besitzt, und wenn

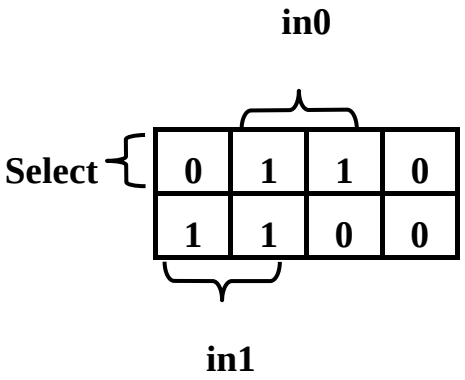
- jede für äquivalente Darstellung derselben Ausgabefunktion mit gleichviel Produkttermen die Anzahl der Eingänge in die Produktterme mindestens genauso groß ist wie die Anzahl der Eingänge in die Produktterme von f.

### 2.6.2. Minimierung von Booleschen Funktionen mit KV-Diagrammen

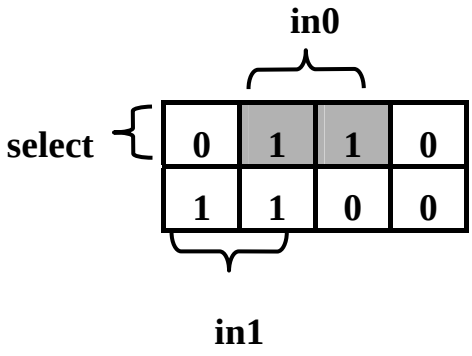
Eine einfache, intuitive Methode zur Minimierung von Funktionen mit wenigen Eingabevariablen liefert die Methode der Karnaugh-Veitch-Diagramme (KV-Diagramme). Die Ränder eines rechteckigen Gitters werden so mit den Variablennamen beschriftet, dass jeweils die Hälfte der Zeilen (Spalten) den nicht invertierten Wert und die andere Hälfte den invertierten Wert der jeweiligen Variablen repräsentiert. Wenn mehrere Variablen am selben oder an gegenüberliegenden Rändern abgetragen werden, müssen auch hier die überlappenden Bereiche und die nicht überlappenden Bereiche gleich groß gewählt werden.

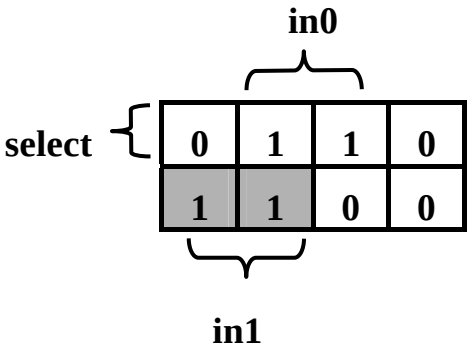
**Beispiel:**

KV-Diagramm für in1, in2, select.




Jedes Feld stellt einen Minterm dar. Im obigen Beispiel steht z.B. die linke Eins in der oberen Zeile für select·in0·in1. Wenn die Ausgangsfunktion für die Eingabekombination, die das Feld repräsentiert, Eins ist, wird eine Eins eingetragen; Ist sie Null, wird eine Null eingetragen. Sodann werden Blöcke von zusammenhängenden Einsen gesucht. Diese werden repräsentiert durch Terme, die nicht alle Eingabevariablen enthalten. Ein Zusammenfassen von zwei Einsen zu einem Block der Größe 2 entspricht einer Anwendung der Vereinfachungsregel. Ein Zusammenfassen von 4 Einsen entspricht zweimaliger Anwendung der Vereinfachungsregel, usw. Die Blöcke müssen immer rechteckige Bereiche aus $2^k$ Einsen sein, für ein ganzzahliges k.

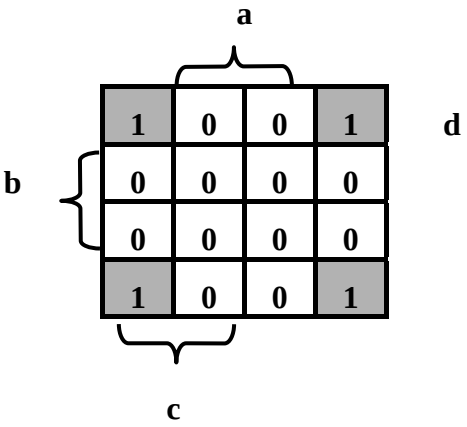





Durch Zusammenfassen der schattierten Blöcke ergibt sich die minimierte Funktion:

$$y = select \cdot in0 + \overline{select} \cdot in1$$

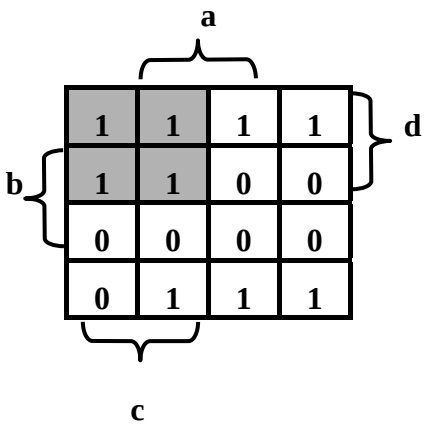
Mit KV-Diagrammen kann man häufig eine Schaltungsminimierung „durch scharfes Hinsehen“ betreiben. Man muss aber vorsichtig sein: Die gegenüberliegenden Ränder des Rechtecks sind als identisch zu betrachten, so dass sich topologisch ein Torus ergibt. Dadurch können auch Blöcke von Einsen zusammengefasst werden, die aus Teilen bestehen, die an gegenüberliegenden Rändern liegen. In einem KV-Diagramm mit vier Eingabevariablen z.B. kann im Extremfall ein Block aus vier Einsen gebildet werden, die in den vier Ecken des Diagramms stehen.

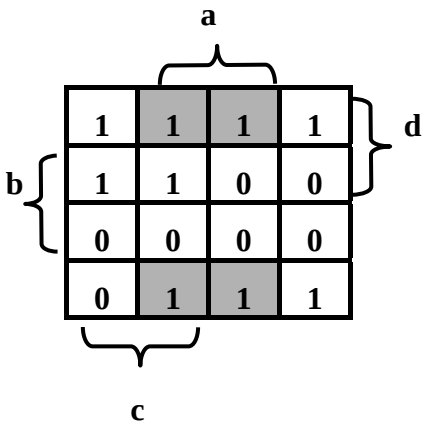


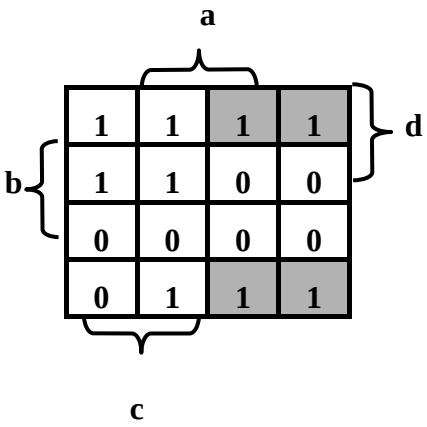

Die Blöcke können sich überlappen. Das Ziel ist, alle Einsen mit einer minimalen Anzahl von Blöcken zu überdecken, wobei die Größe der einzelnen Blöcke maximiert wird.

**Beispiele:**

| a | b | c | d | f |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |



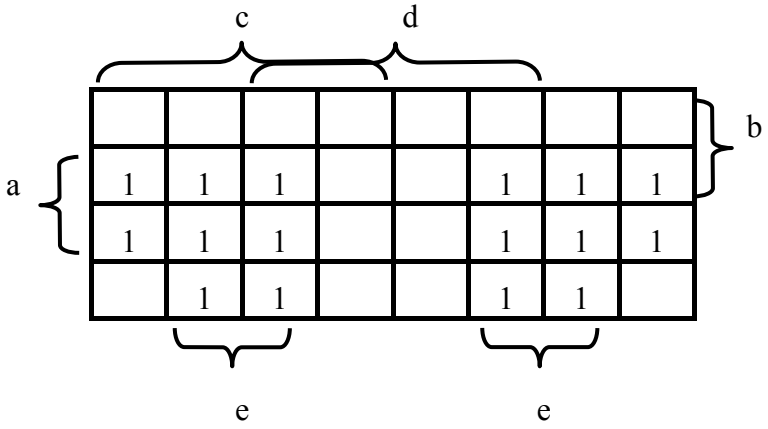





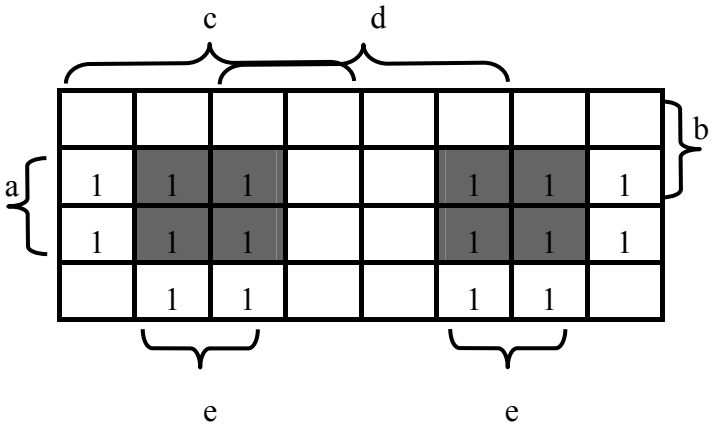

$$f = a\bar{b} + \bar{b}\bar{c} + cd$$

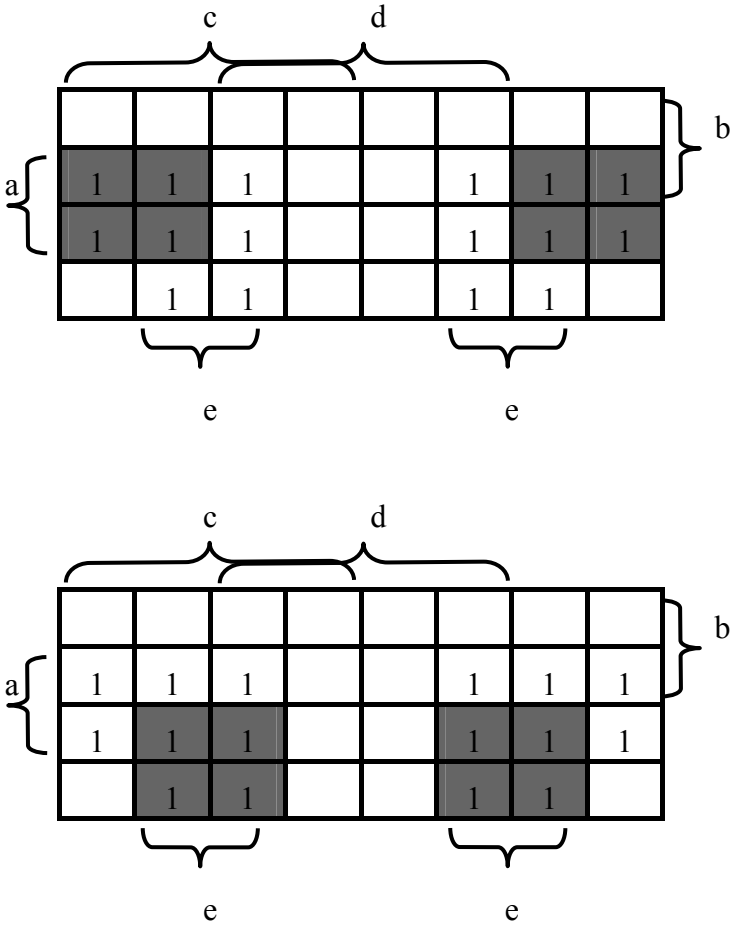
Wenn die Anzahl der Variablen größer als vier wird, kommt ein zusätzliches Problem dazu: man kann die Ränder nicht mehr so beschriften, dass die Variablen auf einem Torus einen zusammenhängenden Bereich abdecken. Daher muss bei jeder Variablen ab der vierten in Kauf genommen werden, dass auch innen im Rechteck liegende Kanten als identisch angesehen werden müssen. Es erfordert hier etwas Übung, die Blöcke zusammenhängender Einsen in einem solchen Schema zu erkennen.

**Beispiel:**




**Lösung:**






$$f = ae + a\bar{d} + \bar{b}e$$

Allen Anfängern sei geraten, bei Funktionen mit mehr als 4 Variablen andere Techniken (z.B. Quine-McCluskey) zusätzlich zu den KV-Diagrammen zu verwenden.

### 2.6.3. Minimierung von Booleschen Funktionen mit dem Verfahren von Quine und McCluskey

KV-Diagramme können nur bis zur Anzahl von 4, höchstens 6 Variablen benutzt werden. Außerdem stellen sie eine Methode des scharfen Hinsehens dar.

Eine systematische Methode zur Konstruktion der DMF ist das Verfahren von Quine und McCluskey. Es besteht darin, aus der KDNF durch wiederholtes Anwenden der zweiten Vereinfachungsregel zu einer Minimalform zu kommen. Dies ist dann die DMF.

Das Verfahren besteht aus der Hintereinanderausführung zweier Algorithmen: Der Algorithmus von McCluskey konstruiert systematisch alle Primterme. Das sind Konjunktionen mit einer minimalen Anzahl von Variablen, für die (ähnlich wie bei Mintermen) die Ausgangsfunktion 1 ist. Die Oder-Verknüpfung aller solcher Primterme ist zwar eine Realisierung der Funktion, aber noch nicht notwendigerweise die optimale. Durch das anschließend ausgeführte Verfahren von Quine werden aus der Gesamtheit der Primterme diejenigen ausgewählt, die die Funktion mit minimalem Aufwand realisieren.

Der Algorithmus von McCluskey:

Für eine Funktion mit n Eingabevariablen werden im ersten Schritt aus den Mintermen (die ja Produkte aus n Faktoren sind) alle möglichen Produktterme mit n-1 Faktoren gebildet, von denen jeder aus zwei Mintermen durch Eliminierung einer Variablen mit der zweiten Vereinfachungsregel hervorgeht. Alle Minterme, die aus einem der so konstruierten Produktterme durch Multiplikation mit einer weiteren Variablen entstehen, werden gestrichen, alle anderen werden als nicht minimierbar markiert. Doppelte Terme werden mit Regel (i) eliminiert.

Im nächsten Schritt wird auf die gleiche Weise mit den so entstandenen neuen Produkttermen verfahren, wobei Produkte mit n-2 Faktoren entstehen. Wieder wird danach überprüft, welche Terme der vorherigen Iteration durch die neu entstandenen Terme „überdeckt" werden, d.h. durch Multiplikation mit nur einer Variablen daraus entstehen. Diese werden gestrichen, der Rest ist bereits minimiert. Doppelte Terme werden wieder eliminiert.

Auf diese Weise wird fortgefahren, bis keine kleineren Terme mehr entstehen. Alle markierten Terme sind Primterme.

Beispiel:

In diesem Beispiel wird mit $\bar{x}$ die invertierte Variable $x$ bezeichnet.

$$abcd + ab\bar{c}d + a\bar{b}cd + \bar{a}bcd + \bar{a}\bar{b}cd + abc\bar{d} + a\bar{b}c\bar{d} + \bar{a}bc\bar{d} + \bar{a}\bar{b}c\bar{d} =$$

I II III IV V VI VII VIII IX

$$abd + acd + bcd + abc + \bar{b}cd + a\bar{b}c + \bar{a}cd + \bar{a}bc + \bar{a}\bar{b}c + ac\bar{d} + bc\bar{d} + \bar{b}c\bar{d} + \bar{s}c\bar{d} =$$

A B C D E F G H I J K L M

$$cd + ac + cd + ac + bc + \bar{b}c + \bar{a}c + c\bar{d} + abd =$$

$$ac + bc + cd + \bar{a}c + \bar{b}c + c\bar{d} + abd =$$

$$c + c + c + abd = c + abd$$

Durch das Verfahren von McCluskey werden u. U. Primterme erzeugt, die redundant sind, weil die Zeilen in der Wertetabelle, die sie überdecken, bereits durch andere Primterme in der Liste der markierten Terme überdeckt werden.

**Beispiel:**

| a | b | c | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

**Beispiel:**

| a | b | c | f |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

KDNF: $f = ab\bar{c} + abc + \bar{a}bc + \bar{a}\bar{b}c$

McCluskey:

$ab + bc + \bar{a}c$

Die Funktion f wird aber bereits durch $ab + \bar{a}c$ realisiert. Der Term $bc$ ist redundant, d.h. er überdeckt nur solche Zeilen in der Wertetabelle, die durch die anderen beiden Terme auch überdeckt werden. Deswegen muss nach der Konstruktion der Primterme mit McCluskey's Algorithmus das Verfahren von Quine angewendet werden, um aus der Liste der Primterme eine Auswahl nicht redundanter Primterme zu treffen.

Das Verfahren von Quine:

Zunächst wird eine Tabelle erstellt, die für jeden Minterm in der DNF eine Zeile und für jeden Primterm, der durch das Verfahren von McCluskey erzeugt worden ist, eine Spalte hat. In das Feld in der Zeile des Minterms M und der Spalte des Primterms P wird eine 1 eingetragen, wenn P M überdeckt, d.h. wenn aus P=1 folgt M=1. Andernfalls wird in dieses Feld eine 0 eingetragen.

Nachdem die Tabelle erstellt ist, werden dominante Zeilen gesucht. Das sind solche, bei denen in der ganzen Zeile nur eine 1 steht. Die Spalten in denen diese Einsen stehen werden markiert (der Primterm ist *nicht* redundant). Dominante Zeilen werden ausgestrichen. Wenn in einer nicht dominanten Zeile eine 1 steht, die auf einer markierten Spalte liegt, so wird die Zeile ebenfalls ausgestrichen.

Mit den nicht markierten Spalten und nicht ausgestrichenen Zeilen fährt man auf gleiche Weise fort. Wenn keine 1 mehr auf einer unmarkierten Spalte und einer ungestrichenen Zeile übrigbleibt, stellt die Oder-Verknüpfung der Primterme der markierten Spalten eine DMF der Funktion dar.

**Beispiel:**

Wir betrachten wieder die Funktion mit der KDNF $f = ab\bar{c} + abc + \bar{a}bc + \bar{a}\bar{b}c$

Mit dem Verfahren von McCluskey hatten wir die Primterme $ab + bc + \bar{a}c$ ermittelt.

Die Quine Tabelle (Primterm/Minterm-Tabelle) sieht so aus:

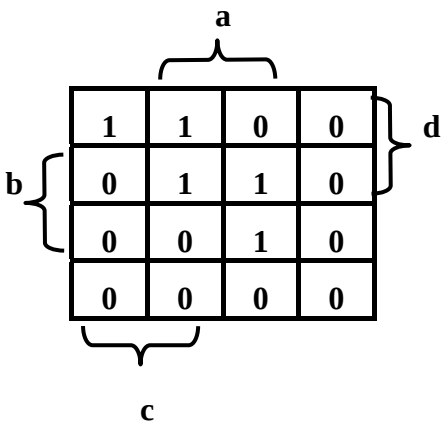
| | $ab$ | $bc$ | $\bar{a}c$ |
|---|---|---|---|
| $ab\bar{c}$ | 1 | | |
| $abc$ | 1 | 1 | |
| $\bar{a}bc$ | | 1 | 1 |
| $\bar{a}\bar{b}c$ | | | 1 |

| | $ab$ | $bc$ | $\bar{a}c$ |
|---|---|---|---|
| $ab\bar{c}$ | 1 | | |
| $abc$ | 1 | 1 | |
| $\bar{a}bc$ | | 1 | 1 |
| $\bar{a}\bar{b}c$ | | | 1 |

Die erste und letzte Zeile sind dominant. Dadurch werden die erste und dritte Spalte markiert. In diesen stehen Einsen in der zweiten und dritten Zeile, die somit auch gestrichen werden. Es bleibt keine 1 übrig. Das Ergebnis ist also $ab + \bar{a}c$.

Es kann vorkommen, dass keine dominante Zeile vorhanden ist. In diesem Falle kann eine beliebige Spalte mit einer maximalen Anzahl von Einsen gestrichen werden.

**Beispiel:**




KDNF: $f = \bar{a}\bar{b}cd + a\bar{b}cd + abcd + ab\bar{c}d + ab\bar{c}\bar{d}$

McCluskey: $f = \bar{b}cd + acd + abd + ab\bar{c}$

Quine:

| | $\bar{b}cd$ | $acd$ | $abd$ | $ab\bar{c}$ |
|---|---|---|---|---|
| $\bar{a}\bar{b}cd$ | 1 | | | |
| $a\bar{b}cd$ | 1 | 1 | | |
| $abcd$ | | 1 | 1 | |
| $ab\bar{c}d$ | | | 1 | 1 |
| $ab\bar{c}\bar{d}$ | | | | 1 |

Die erste und letzte Zeile sind dominant. Daraufhin werden die erste und vierte Spalte markiert. Dabei werden die zweite und vierte Zeile gestrichen.

| | $\bar{b}cd$ | $acd$ | $abd$ | $ab\bar{c}$ |
|---|---|---|---|---|
| $\bar{a}\bar{b}cd$ | 1 | | | |
| $a\bar{b}cd$ | 1 | 1 | | |
| $abcd$ | | 1 | 1 | |
| $ab\bar{c}d$ | | | 1 | 1 |
| $ab\bar{c}\bar{d}$ | | | | 1 |

Nun ist keine dominante Zeile mehr vorhanden, es kann eine beliebige (die zweite oder die dritte) Spalte markiert werden. Die dritte Zeile wird gestrichen.

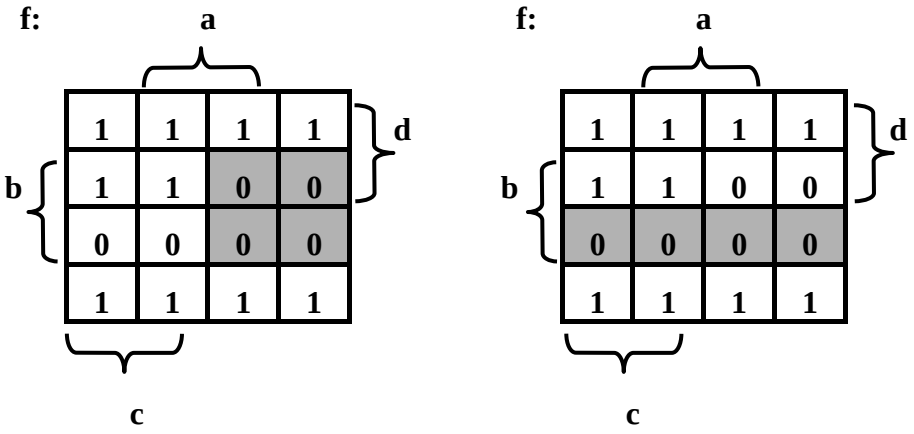
### 2.6.4. Nutzung von KV-Diagrammen zur Minimierung über Maxterme.

Die im letzten Abschnitt diskutierte Art der Minimierung führt zu einer disjunktiven Minimalform. Man kann KV-Diagramme aber auch Nutzen, um eine konjunktive Minimalform zu konstruieren.

Anstelle der Einsen fasst man maximale Blöcke aus Nullen zusammen. Jede Null steht für einen Maxterm. Zusammenfassen von zwei kleinen Blöcken von Nullen zu einem doppelt so großen Block entspricht einer Anwendung der ersten Vereinfachungsregel.

Die Terme der konjunktiven Minimalform werden nun gebildet als Oder-Verknüpfung der **invertierten** Variablen, die die 0-Blöcke im KV-Diagramm überdecken.

**Beispiel:**




$$f = (c + \bar{b}) \cdot (\bar{b} + d)$$

### 2.6.5. Darstellung boolscher Funktionen mit eingeschränkten Gattertypen

**Satz:**

Jede boolesche Funktion kann unter ausschließlicher Verwendung von Invertern, Und-Gattern und Oder-Gattern realisiert werden.

**Beweis:**

Wie schon früher bewiesen wurde, kann man jede boolesche Funktion in einer KDNF darstellen. Dabei besteht eine KDNF aus Und- und Oder-Verknüpfungen und Negationen.

Jede boolesche Funktion kann realisiert werden unter ausschließlicher Verwendung von

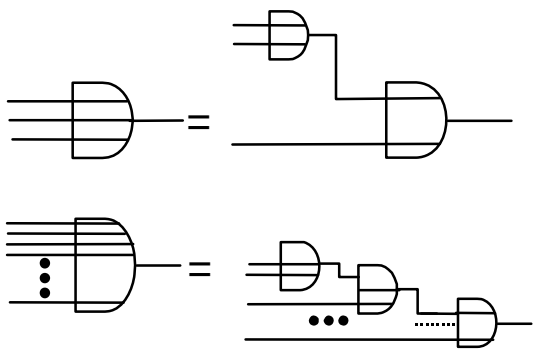
(i) Invertern, Und-Gattern mit zwei Eingängen und Oder-Gattern mit zwei Eingängen.

Beweis:

a) Oder-Gatter: Wiederholte Anwendung der folgenden Gleichung:

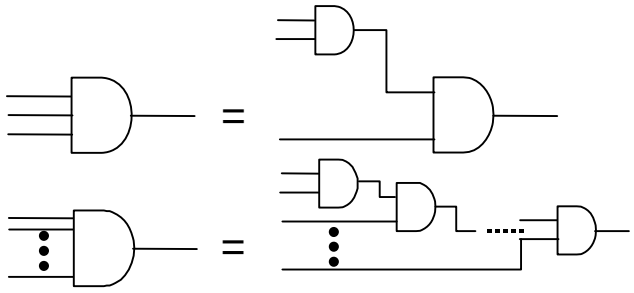
$(a_1+a_2+a_3+a_4...a_{k-1}+a_k)= a_1+(a_2+a_3+a_4+...+a_{k-1}+a_k)$

Bild:



b) Und-Gatter: Wiederholte Anwendung der folgenden Gleichung:
$(a_1a_2a_3a_4...a_{k-1}a_k)=a_1(a_2a_3a_4...a_{k-1}a_k)$

Bild:



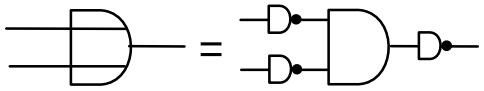
(ii) Invertern und Und-Gattern mit zwei Eingängen.

Beweis: jeder Oder-Gattern kann man mit Hilfe von Und-Gattern und Inverter darstellen.

Nach dem Gesetz von De Morgan gilt:

$$a+b=\overline{\overline{a+b}}=\overline{\bar{a}\,\bar{b}}$$

Bild:

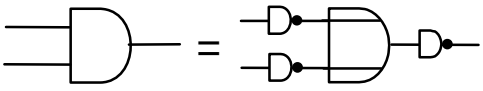


(iii) Invertern und Oder-Gattern mit zwei Eingängen.

Beweis: Jedes Und-Gatter kann man mit Hilfe von Oder-Gattern und Invertern darstellen.

$$ab=\overline{\overline{ab}}=\overline{\bar{a}+\bar{b}}$$

Bild:



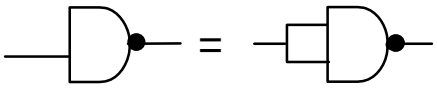
(iv) Nand-Gattern mit zwei Eingängen.

Beweis:

a). Negation:

$a=aa$, also $\bar{a}=\overline{aa}$

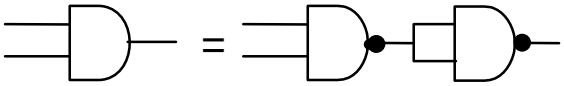
Bild:



b). Und:

$ab = \overline{\overline{ab}} = \overline{\overline{ab}\,\overline{ab}}$

Bild:



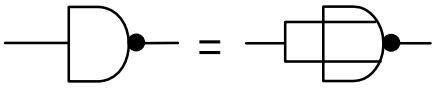
(v) Nor-Gattern mit zwei Eingängen.

Beweis:

a). Negation:

$a = a + a$ also $\overline{a} = \overline{a + a}$

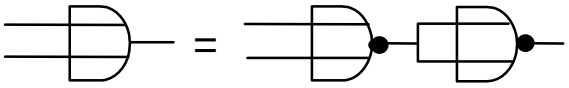
Bild:



b). Oder:

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

Bild:



Warum interessiert man sich für so etwas? Weil „Nand“ und „Nor“ in einer Realisierung in CMOS-Technologie billiger sind als „And“ und „Or“. Das sehen wir genau im nächsten Abschnitt:

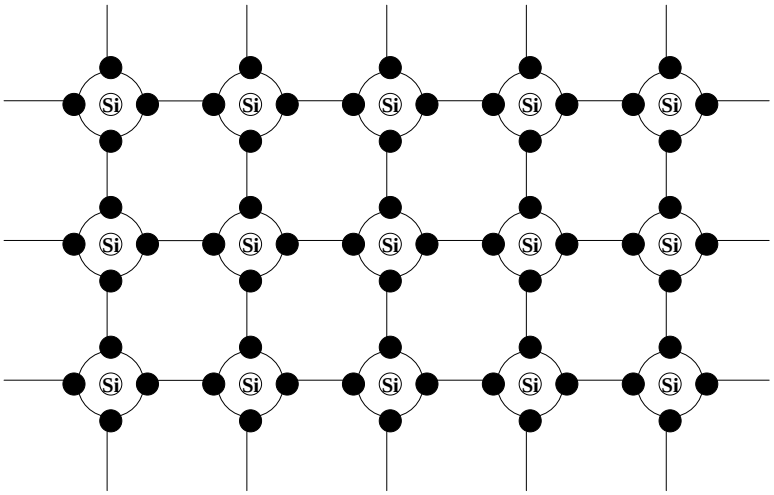
### 2.6.6. Realisierung von Booleschen Funktionen in CMOS-Technologie

**Halbleiterdioden**

Halbleiterdioden sind Bauelemente, die die Leitfähigkeitseigenschaften eines **pn-Übergangs** nutzen. Sie werden meist aus Silizium hergestellt. Ein pn-Übergang ist der Übergang von positiv dotiertem (p-) Silizium zu negativ dotiertem (n-) Silizium. Dotierung ist das gezielte Einfügen von Fremdatomen in die Kristallstruktur des Siliziums. Silizium ist vierwertig, und bildet in reinster Form eine Kristallstruktur, bei der je zwei Elektronen zweier benachbarter Atome eine Bindung eingehen. Man kann sich das als ein regelmäßiges rechteckiges Gitter

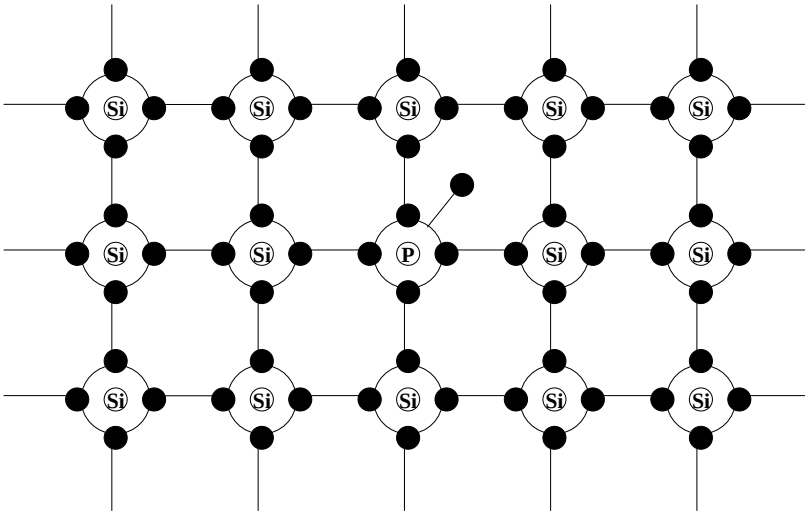
vorstellen, wie es auf dem nächsten Bild gezeigt ist. Es enthält so keine freien Elektronen und ist daher auch kaum leitfähig.

Bild: Silizium-Kristallgitter




Wenn man in diesem Gitter nun ein Atom durch ein 5-wertiges Fremdatom ersetzt, z.B. Phosphor, so gibt es zusätzlich zu dem vollständigen Gitter ein freies Elektron, das keine Bindung eingehen kann.

Bild: Negativ dotiertes Silizium

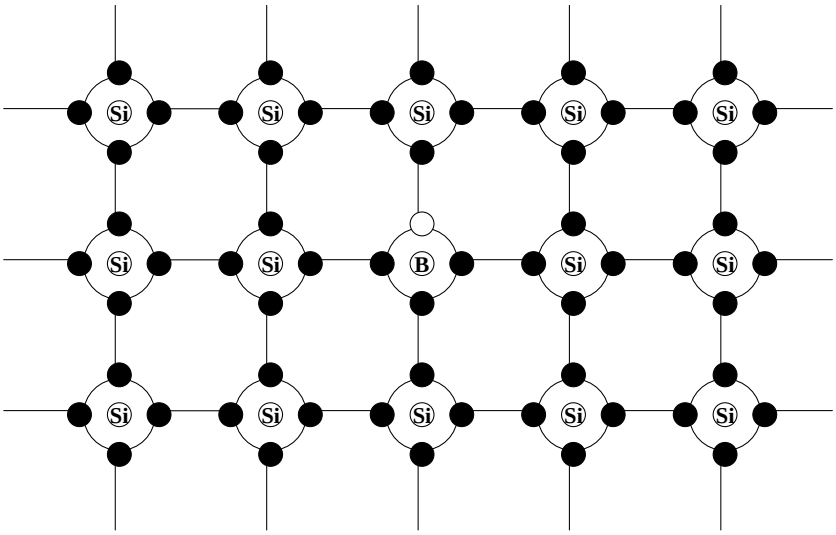



Dieses freie Elektron kann jetzt zur Leitung von elektrischem Strom benutzt werden. Durch **Dotierung von Silizium mit Phosphor** erzeugt man somit aus dem zunächst schlecht leitenden Silizium einen Leiter. Aus diesem Grund bezeichnet man Silizium als **Halbleiter**. Die Dotierung mit Phosphor generiert freie Elektronen im Silizium. Da diese negativ geladen sind, spricht man von einer **n-Dotierung**, gelegentlich auch von einem **n-Halbleiter**.

Verwendet man anstelle von Phosphor ein dreiwertiges Element, z.B. Bor, so kommt man zu einer **p-Dotierung**. Ein entsprechendes Kristallgitter ist im folgenden Bild zu sehen. Es gibt

allerdings einen qualitativen Unterschied. Im p-dotierten Material fehlen Elektronen im Kristallgitter. Das fehlende Elektron eines Atoms kann durch ein Elektron eines Nachbaratoms ersetzt werden, wenn dieses aus seiner Paarbindung herausgelöst wird. Es entsteht dort eine **positive Ladung**, ein **Loch** oder **Defektelektron**. Das Fremdatom (Bor), das jetzt ein Elektron aufgenommen hat, wird negativ geladen, bleibt aber ortsfest.

Bild: Positiv dotiertes Silizium, Mobilität der „Löcher“

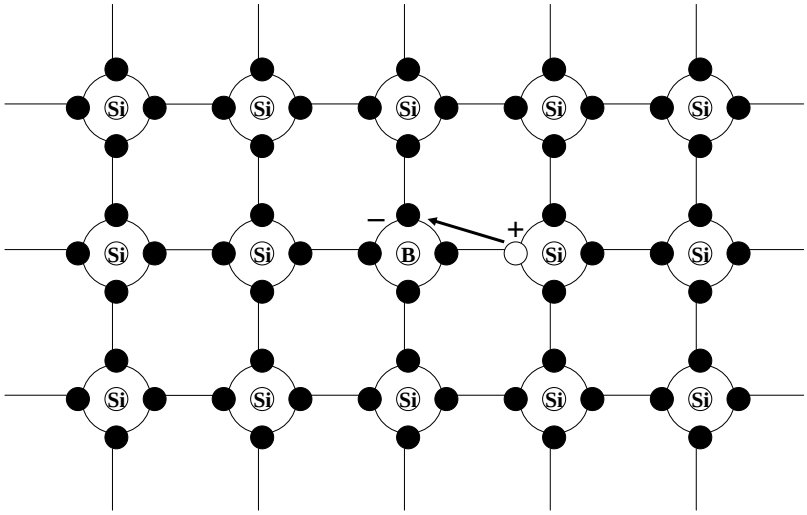



Durch diesen Vorgang des Ersetzens eines fehlenden Elektrons durch ein Nachbarelektron **wandert das Loch** nun im Kristallgitter. Es kann also ebenfalls genutzt werden, um elektrischen Strom zu transportieren. Allerdings ist die Beweglichkeit der Löcher im p-dotierten Halbleiter nicht so groß wie die Beweglichkeit der freien Elektronen im n-dotierten Halbleiter, weil die Elektronen im p-dotierten Material ja zuerst aus ihrer bestehenden Bindung herausgelöst werden müssen. Als Daumenregel kann man sich merken, dass n-dotiertes Silizium etwa eine dreimal so hohe Leitfähigkeit hat wie p-dotiertes.

Wenn man nun eine p-Dotierung direkt an eine n-Dotierung angrenzen lässt, entsteht ein **pn-Übergang**. Wegen der freien Elektronen in der n-Zone und der (frei beweglichen) Löcher in der p-Zone entsteht an der Grenze eine spezielle Reaktion: Freie Elektronen diffundieren in die p-Zone und Löcher in die n-Zone, wo sie rekombinieren. Dadurch verringert sich die Zahl der freien Ladungsträger in der Grenzschicht. Die ladungsträgerfreie Grenzschicht wird zu einer hochohmigen sogenannten Sperrschicht.

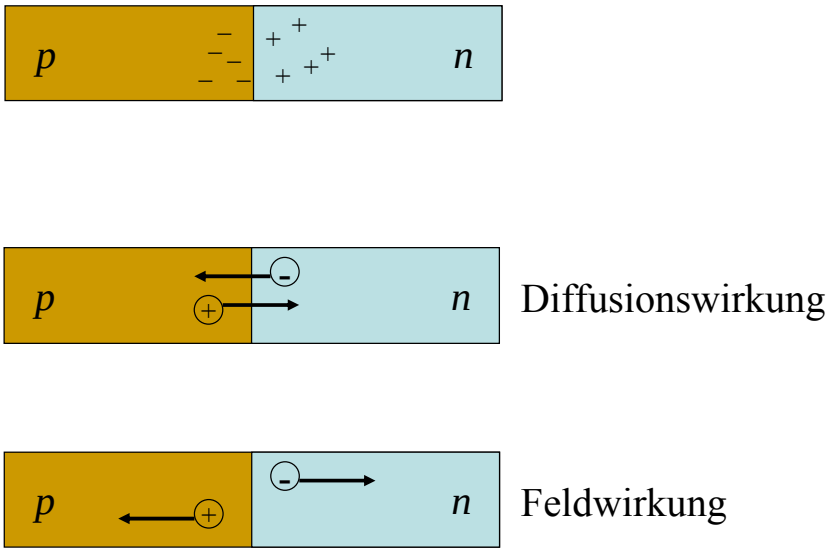
Durch die Diffusion der Elektronen in die Sperrschicht bleiben aber ortsfeste, positive Ionen (**sogenannte Raumladungen**) zurück, und durch Rekombination der Löcher mit den Elektronen entsehen in der p-Zone ortsfeste negative Ionen.

Bild: pn-Übergang




Zwischen der positiven Raumladung und der negativen Raumladung entsteht ein elektrisches Feld. Auf freie Ladungsträger innerhalb der Raumladungszone wirkt die Diffusion und in entgegengesetzter Richtung die elektrische Feldkraft.

Bild: Kräfte auf Ladungsträger am pn-Übergang

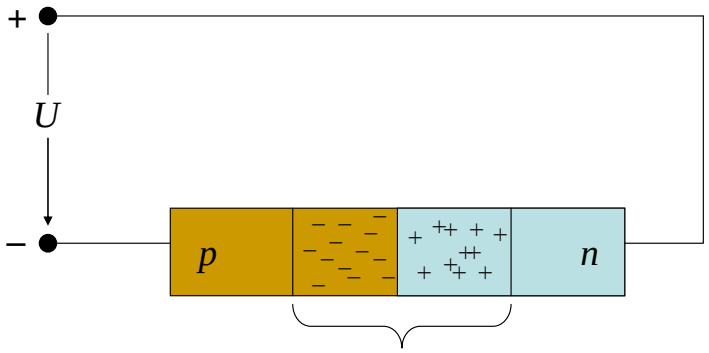



Es stellt sich ein dynamisches Gleichgewicht am pn-Übergang ein, wenn die Feldwirkung und die Diffusionswirkung gleich groß ist. Dann besteht zwischen der positiven Raumladung in der n-Zone und der negativen Raumladung in der p-Zone eine feste Spannung, die Diffusionsspannung $U_D$. Diese beträgt bei Silizium etwa 0,75 V.

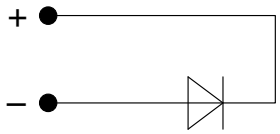
Wenn man an eine Diode eine Gleichspannung anlegt, wird sie – je nach Polung der Gleichspannung – **leitend oder sperrend**. Wenn der Minuspol der Spannungsquelle an die p-Zone und der Pluspol and die n-Zone der Diode gelegt wird, steigt die Spannung in der Raumladungszone auf $U_D+U$. Die Feldstärke wird größer und die ladungsträgerfreie

Raumladungszone wird breiter. Sie wird zu einer hochohmigen Sperrschicht; man sagt, die Diode ist in Sperrrichtung gepolt.

Bild: Diode in Sperrrichtung

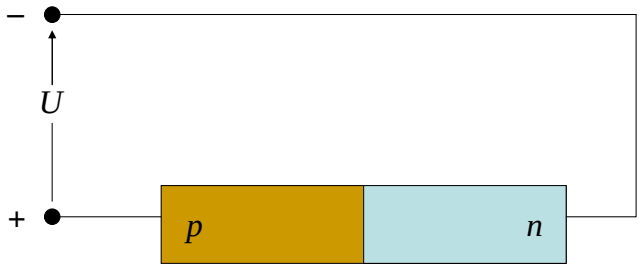



Raumladungszone ohne
freie Elektronen oder Löcher

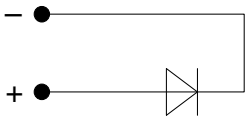



Wenn umgekehrt der Minuspol der Spannungsquelle an die n-Zone und der Pluspol and die p-Zone der Diode gelegt wird, sinkt die Spannung in der Raumladungszone auf $U_D$-$U$. Die ladungsträgerfreie Raumladungszone wird schmaler und verschwindet ganz, wenn $U>U_D$ ist. (Der Wert von $U$, für den das gilt, wird auch **Schwellspannung** genannt). Dadurch ist die Diode leitend, weil jetzt auf dem ganzen Weg von + nach – genügend freie Ladungsträger sind.

Bild: Diode in Durchlassrichtung




Keine Raumladungszone, freie
Elektronen und Löcher vorhanden



Durch Kombination von zwei pn-Übergängen kann man Bipolartransistoren aufbauen. Da diese aber in der digitalen Schaltungstechnik gegenwärtig keine große Bedeutung mehr haben, werden sie hier nicht behandelt. Stattdessen konzentrieren wir uns auf Schaltelemente, die heute und sicher auch noch in Zukunft die bedeutendste Rolle im Aufbau von Digitalschaltungen haben, die MOS-Transistoren.

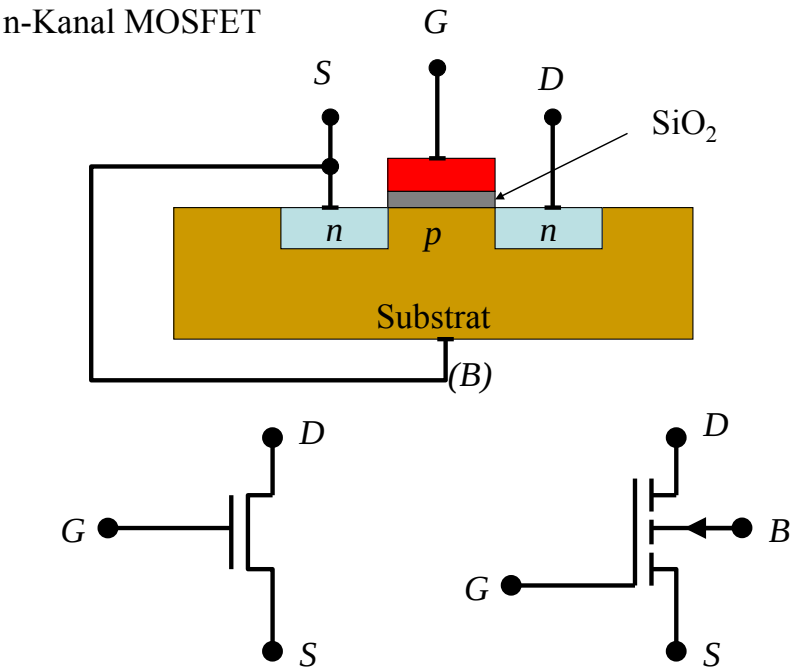
**MOS-Transistoren**

Die Abkürzung **MOSFET** steht für **Metal-Oxide-Semiconductor Field Effect Transistor**. Metal-Oxide-Semiconductor bezeichnet die Schichtenfolge, durch die er ursprünglich aufgebaut wurde (heute nimmt man anstelle von Metall in der Regel polykristallines Silizium). Field Effect Transistor bedeutet: Der Transistor-Effekt wird erzielt durch Erzeugen eines elektrischen Feldes durch Anlegen einer Spannung an die Steuerelektrode.

Die drei Anschlüsse eines FETs werden mit **D (Drain), S (Source)** und **G (Gate)** bezeichnet. Das Gate ist die **Steuerelektrode**, auf die man eine Spannung legt, um dadurch einen Verbindung zwischen Drain und Source zu schalten.

Man unterscheidet MOSFETs nach der Art der Dotierung des Halbleitermaterials, in dem (bei geeigneter Beschaltung) der leitende Kanal zwischen Drain und Source entsteht. Wir beginnen mit dem **selbstsperrenden n-Kanal MOSFET**, **(enhancement mode n-channel MOSFET)** dessen Aufbau im folgenden Bild zu sehen ist.

Bild: n-Kanal-MOSFET, Aufbau der Schichten

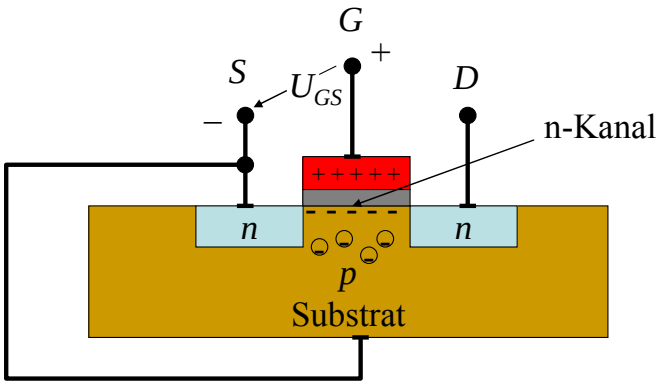



In den als **Substrat** bezeichneten p-Halbleiter sind zwei hochdotierte n-Zonen als Source und Drain eindiffundiert. Sie sind mit dem Source- bzw. Drainanschluss verbunden. Auf das Substrat ist zwischen diesen beiden Zonen eine Isolierschicht aus Siliziumdioxyd aufgebracht. Darüber befindet sich das Gate, das somit isoliert ist gegenüber Source, Drain und Substrat. Da die Oxydschicht allerdings sehr dünn ist, bildet das Gate mit dem Substrat

einen Kondensator. Die Zonenfolge Source-Substrat-Drain ist eine npn-Anordnung. Weil der Abstand zwischen den beiden n-Zonen zu groß ist, bildet sich aber kein bipolarer Transistor.

Wird nun an das Gate eine gegenüber dem Substrat positive Spannung angelegt, so werden die Löcher als bewegliche Ladungsträger vom Gate weg in das Substrat abgestoßen. Es entsteht an der Randschicht zum Oxyd hin eine negative Raumladungszone. Wenn das dadurch gebildete elektrische Feld so groß ist, dass die freien Elektronen nicht mehr in das Substrat hineindiffundieren, so bildet sich am Rand der Raumladungszone eine leitende Schicht aus freien Ladungsträgern (Elektronen). Diese wird **n-Kanal** genannt.

Bild: Kanalbildung im n-Kanal-MOSFET




Die Spannung, ab der sich ein leitender Kanal bildet wird **Schwellspanung** (Threshold) $\boldsymbol{U_{th}}$ genannt (im englischen $V_{th}$).

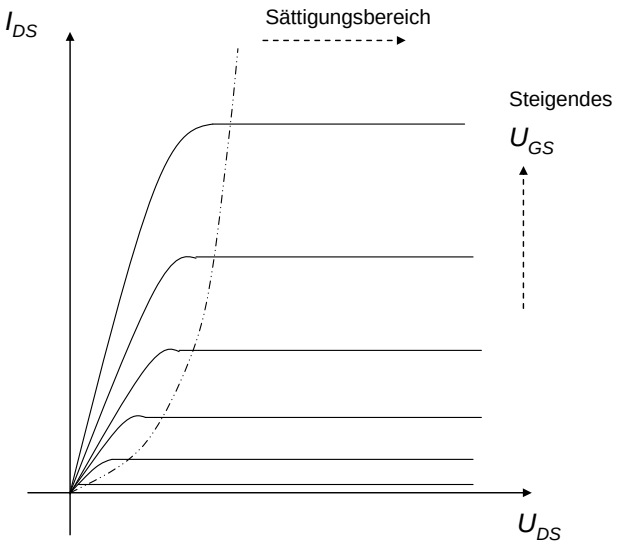
Hat sich ein solcher leitender Kanal gebildet, dann kann ein Drainstrom fließen (von Drain nach Source), wenn $U_{DS} > 0V$ ist. Die Abhängigkeit des Drainstroms $I_D$ von $U_{DS}$ und $U_{GS}$ wird als **Kennlinienfeld** dargestellt (nächstes Bild).

Der Substratanschluss wird mit dem Sourceanschluss verbunden und auf das Spannungspotential 0V gelegt . Dies wird als Bezugspotenzial (Bulk, B) genutzt (das im alten deutschen Schaltzeichen auch explizit eingetragen wird).

Im Sperrbereich ist $U_{DS} > 0V$ und $U_{GS} < U_{th}$ . Es kann sich kein leitender Kanal aufbauen. Da der Drain-Substrat-Übergang eine in Sperrrichtung beschaltete Diode darstellt, fließt kein Strom $I_D$.

Im Arbeitsbereich $U_{GS} > U_{th}$ bildet sich ein leitender n-Kanal. Durch diesen fließen die Elektronen aus der n-Zone als Drainstrom aufgrund der Spannung zwischen Drain und Source.

Bild: Kennlinienfeld eines n-Kanal-MOSFET




Solange $U_{DS} < U_{GS} - U_{th}$ ist, steigt der Drainstrom $I_D$ etwa proportional zur Drainspannung $U_{DS}$. Dies ist der **lineare Bereich der Kennlinie (oder Widerstandsbereich)**. Wird aber $U_{DS} > U_{GS} - U_{th}$ , so wird die Raumladungszone am Drain-Substrat-Übergang größer (weil er eine Diode in Sperrrichtung darstellt) und der leitende Kanal wird „abgeschnürt“. Der Drainstrom $I_D$ geht in den so genannten **Sättigungsbereich** (und steigt nicht nennenswert weiter, auch wenn $U_{DS}$ wächst). Trotz der Abschnürung fließt aber weiterhin ein Strom, da der Kanal bis zu einem bestimmten Abstand von der Drain besteht, und die Elektronen von dort aus durch das elektrische Feld der Drain-Source-Spannung zur Drain hingezogen werden.

Der **p-Kanal-MOSFET** arbeitet analog. Hier wird allerdings das Gate negativ gegenüber Source und Substrat angesteuert. Dadurch wird am Rand der Isolationsschicht ein leitender **p-Kanal** aus Löchern gebildet, über den der Drainstrom fließen kann.

Bild: p-Kanal-MOSFET

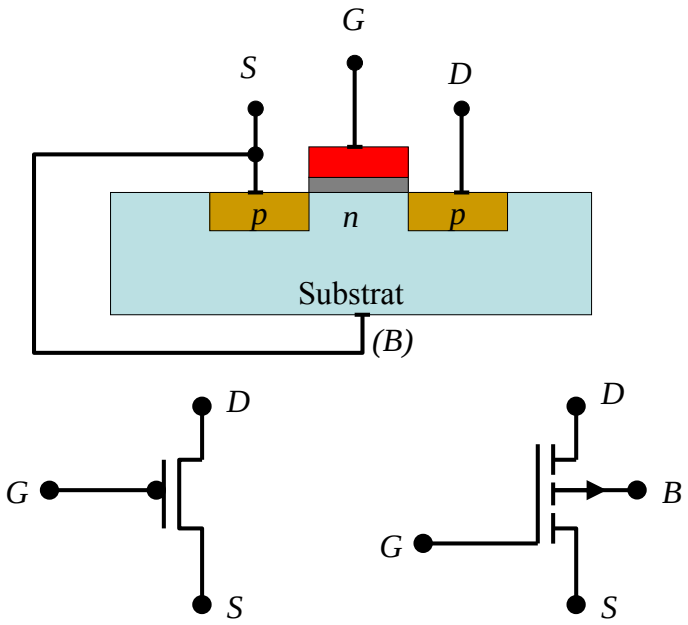



Bild: Leitender Kanal im p-Kanal-MOSFET bei negativer Gate-Substrat-Spannung

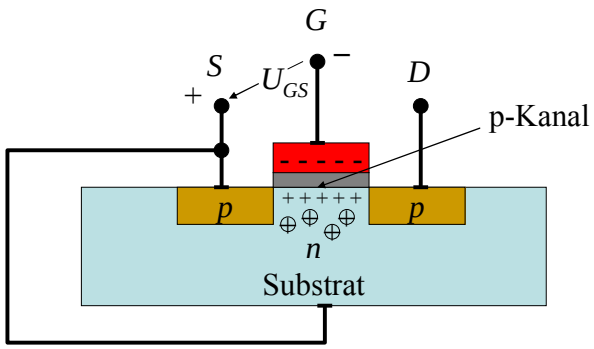
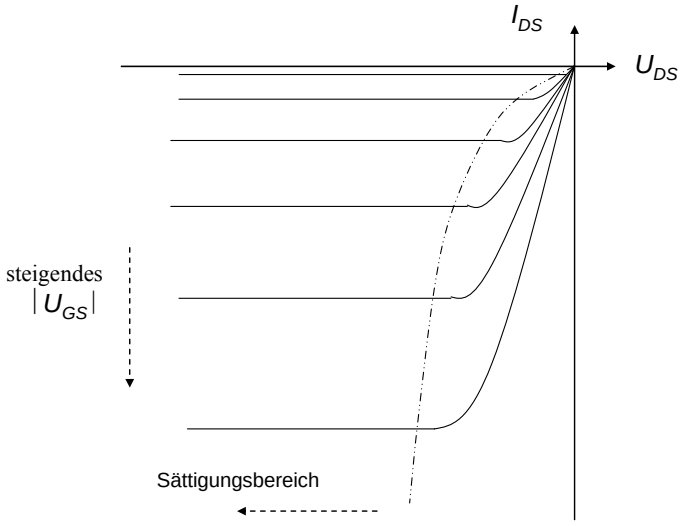



Bild: Kennlinienfeld des p-Kanal-MOSFET




**Aubau einfacher Gatter aus MOS-Transistoren**

MOSFETs können als Schalter benutzt werden. Ein n-Kanal Transistor zum Beispiel verbindet Drain und Source, wenn an seinem Gate eine ausreichend hohe Spannung anliegt

($U_{GS}>U_{th}$). Wenn die Eingangsspannung niedrig ist, sind Drain und Source getrennt ($U_{GS}<U_{th}$). Der p-Kanal Transistor verbindet, wenn seine Eingangsspannung hinreichend klein (negativ) ist im Vergleich zur Source ($U_{GS}<V_{dd}-U_{th}$). und trennt, wenn sie nicht klein genug ist ($U_{GS}>V_{dd}-U_{th}$).

Durch Kombination dieser beiden Typen von Transistoren können wir jetzt logische Schaltungen aufbauen. Da zu jeder Zusammenschaltung von n-Transistoren (die für die logische 0 am Ausgang zuständig ist) immer eine komplementäre Schaltung aus p-Transistoren (für die logische 1 am Ausgang) benutzt wird, nennt man diese Technik **CMOS (complementary MOS) Technik**. Die einfachste solche Schaltung ist ein **Inverter**.

Der Inverter hat einen Eingang und einen Ausgang. Die Spannungen am Ein- und Ausgang identifizieren wir mit logischen Werten, z.B. die volle Versorgungsspannung $V_{dd}$ mit logisch 1 (oder wahr oder TRUE) und das Massepotenzial *GND* (0V) mit logisch 0 (oder falsch oder FALSE).

Bild: Wertetabelle des Inverters

| In | out |
|---|---|
| 0 | 1 |
| 1 | 0 |

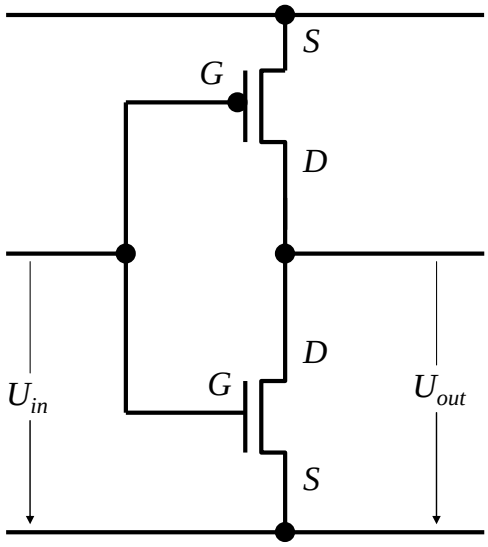
In Spannungen geschrieben sieht diese Tabelle so aus (dabei wird von einer Versorgungsspannung $V_{dd}$ von 2,5 V ausgegangen, die bei heutigen integrierten Schaltkreisen üblich ist:

Bild: Spannungstabelle des Inverters

| $U_{in}$ | $U_{out}$ |
|---|---|
| 0V | 2,5V |
| 2,5V | 0V |

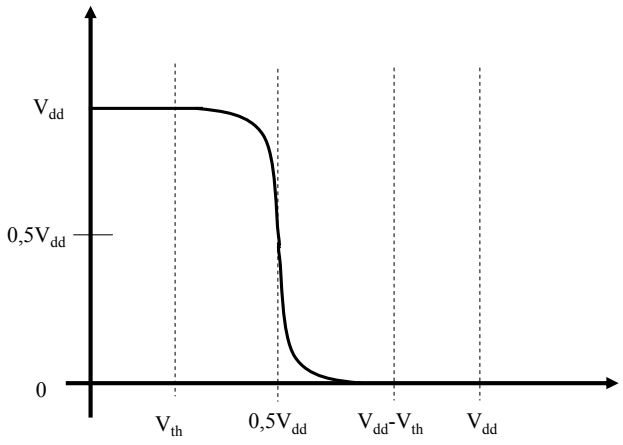
Wir bauen einen Inverter auf, indem wir einen p-Tansistor und einen n-Transistor in Serie zwischen $V_{dd}$ und *GND* schalten, deren Gate gemeinsam mit dem Eingang beschaltet wird. Da abhängig vom logischen Pegel des Gates immer einer der Transistoren sperrt, kann nie ein Kurzschluss zwischen $V_{dd}$ und *GND* entstehen, bei dem Strom fließt. (Genaugenommen stimmt das nicht ganz: Im Moment des Umschaltens des Eingangs von 0 auf 1 oder umgekehrt von 1 auf 0 werden kurzzeitig beide Transistoren leitfähig. In diesem kurzen Zeitraum fließt ein kleiner Strom von $V_{dd}$ nach *GND*).

Bild: Inverter auf Transistorebene




Das folgende Bild zeigt die Ausgangsspannung in Abhängigkeit von der Eingangsspannung am CMOS-Inverter. Wir sehen, dass genau das in der Wertetabelle vorgegebene Verhalten gezeigt wird. Mehr noch: Auch wenn die Eingangsspannung nicht genau auf $V_{dd}$- oder *GND*-Potenzial liegt, wird der Ausgang ein sauberes Spannungssignal liefern. Ein CMOS-Inverter kann also „schlechte“ (geringfügig verfälschte) Eingangssignale verarbeiten (und durch sein Schaltverhalten „reparieren“).

Bild: Ausgangsspannung in Abhängigkeit von der Eingangsspannung




Eine einfache (für Informatiker, die sich den elektrotechnischen Hintergrund nicht verinnerlichen können) Veranschaulichung der Inverterfunktion liefert das folgende Ersatzschaltbild, in dem die Transistoren als Schalter modelliert werden: Der p-Transistor ist ein Schalter, der bei Eingabe einer 0 geschlossen ist und bei Eingabe einer 1 offen. Der n-Transistor ist bei Eingabe 0 offen und bei Eingabe 1 geschlossen.

Bild: Veranschaulichung der Transsitorfunktion als Schalter

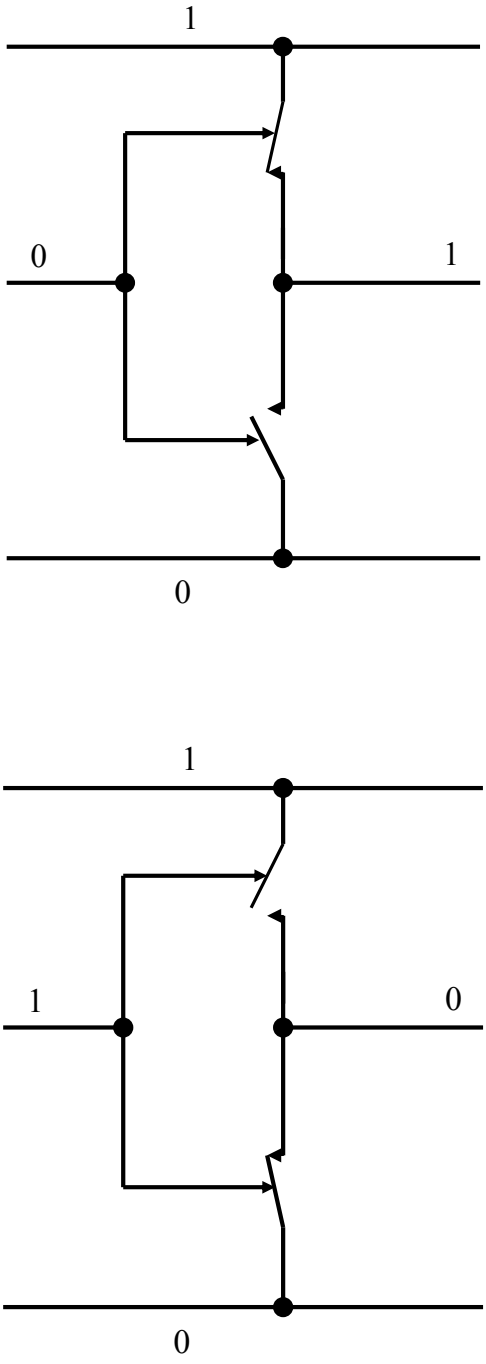
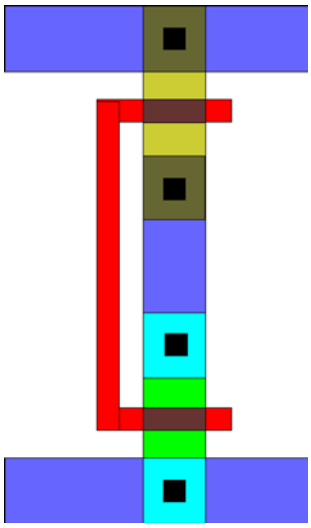



Bild: Ansicht eines Inverters auf dem Chip von oben



Ein Effekt muss hier (auch den Informatikern gegenüber) erwähnt werden. Wenn man n-MOS-Transistoren im Sättigungsbereich betreibt, besteht nur ein Kanal, wenn die Spannung zwischen Gate und Source größer als die Schwellspannung ist ($U_{GS} > U_{th}$). Das bedeutet, wenn die volle Versorgungsspannung an der Drain und am Gate anliegt, und die Source offen ist, kann sich an der Source kein Potenzial einstellen, das höher ist als $U_{GS} - U_{th}$. Das bedeutet, ein n-Transistor ist zwar gut geeignet, um das *GND*-Potenzial weiterzuleiten, bei der Weiterleitung der vollen Versorgungsspannung aber wird diese um eine Schwellspannung vermindert.

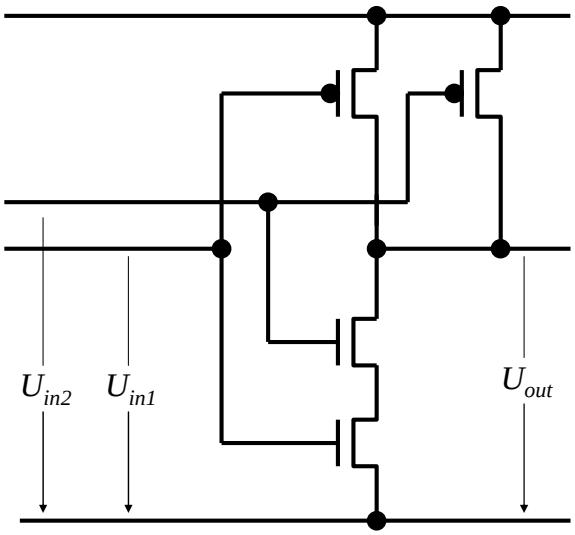
In der Begrifflichkeit der logischen Werte bedeutet das, eine 0 kann von einem n-Transistor gut weitergegeben werden, eine 1 aber nicht. Am Ausgang würde eine „schlechte“ 1 entstehen, also eine Spannung, die um eine Schwellspannung niedriger ist als die Eingangsspannung.

Bei Eingabe einer 0 öffnet der p-Transistor und wird im Widerstandsbereich betrieben, d.h. am Ausgang ensteht eine gute 1. Der n-Transistor sperrt, da keine positive Gate-Spannung gegenüber dem Substrat vorliegt.

Die nächste Funktion, die wir in einem Gatter mit MOS-Transistoren realisieren wollen, ist ein Nand-Gatter. Der Ausgang eines Nand-Gatters ist 1, wenn nicht beide Eingänge auf 1 sind. Wir müssen also erreichen, dass auf den Ausgang das *GND*-Potenzial gelegt wird, wenn beide Eingänge auf 1 sind (*). Ferner müssen wir das Versorgungspotenzial $V_{dd}$ an den Ausgang bringen, falls mindestens einer der Eingänge 0 ist (**).

Wie gelingt dies? Wir schalten zwei n-Transistoren in Serie und verbinden die Source des ersten mit *GND*. Damit erfüllt die Drain des zweiten die erste Bedingung (*) für den Ausgang. Ferner schalten wir zwei p-Transistoren parallel, deren Sourcen wir an $V_{dd}$ anschließen. Wiederum ist die Drain der Ausgang. Damit ist die zweite Bedingung (**) für den Ausgang erfüllt. Verbinden wir nun die Drainanschlüsse dieser beiden Pfade, so ist das Nand-Gatter fertig.

Bild: Nand-Gatter



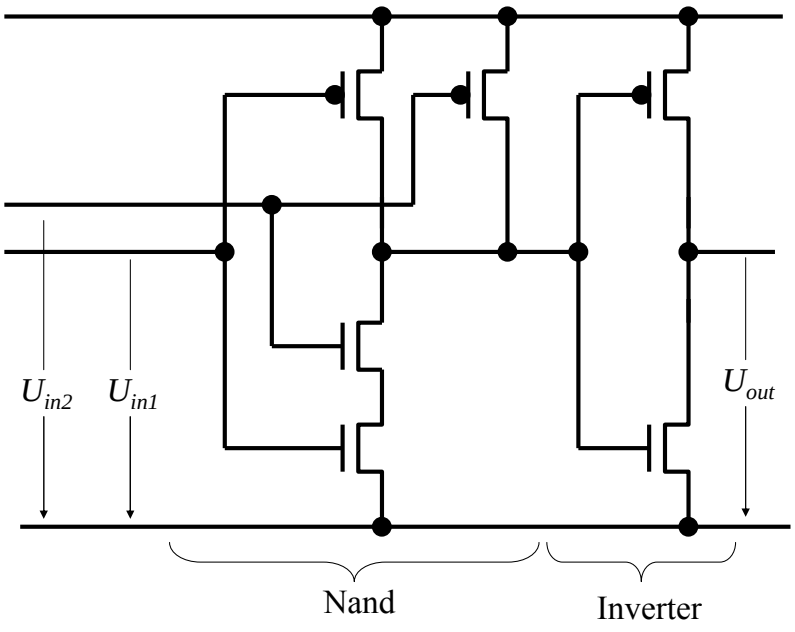

| in1 | in2 | out |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Man beachte, dass die n-Transistoren wieder lediglich gebraucht werden, um das Potenzial der logischen 0 (*GND*) an den Ausgang zu bringen. Ebenso benutzen wir die p-Transistoren nur zur Weiterleitung des logischen 1-Potentials ($V_{dd}$). Somit sind die Signale am Ausgang nicht um eine Schwellspannung unterschiedlich zu den beabsichtigten Potenzialen für die entsprechenden logischen Werte. Es ist ein „gute“ 0 und eine „gute“ 1, die am Ausgang zu beobachten ist.

Wie können wir nun ein Und-Gatter aufbauen?

Durch Hintereinanderschalten eines Nand-Gatters mit einem Inverter. Diese Schaltung ist im nächsten Bild zu sehen.

Bild: Und-Gatter

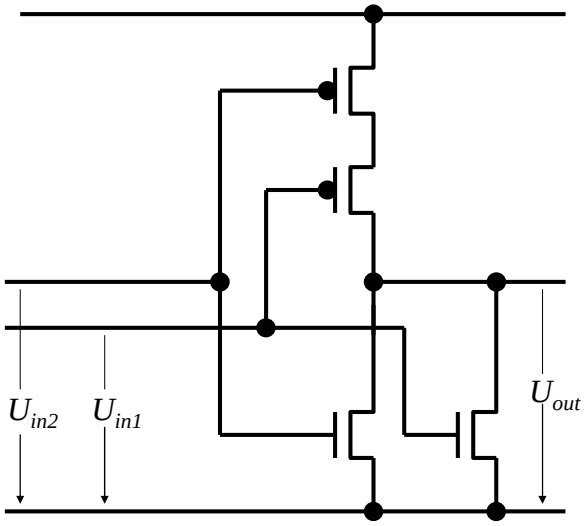


Geht das auch mit weniger Transistoren?

Bitte probieren Sie es aus. Beachten Sie aber dabei, dass am Ausgang „gute“ Signale entstehen sollen. Warum?

Auch ein Nor-Gatter kann man mit zwei n-Transistoren und zwei p-Transistoren aufbauen. Hier müssen allerdings die p-Transistoren in Serie geschaltet werden und die n-Transistoren parallel.

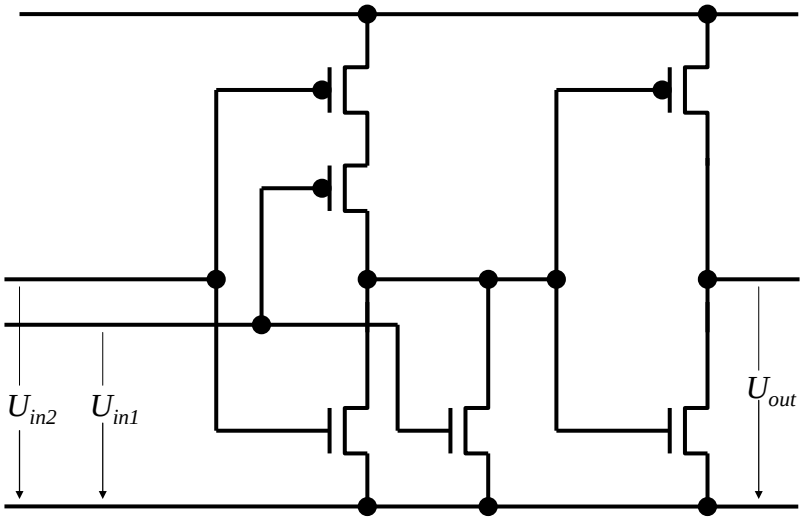
Bild: Nor-Gatter




| in1 | in2 | out |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Wiederum können wir ein Oder-Gatter aus einem Nor-Gatter mit einem nachgeschalteten Inverter bauen.

Bild: Oder-Gatter




Für die technisch Interessierten Hörer oder Leser sei erwähnt, dass ein Nand-Gatter etwas schneller schaltet als ein Nor-Gatter. Das liegt an der höheren Mobilität der Elektronen im n-

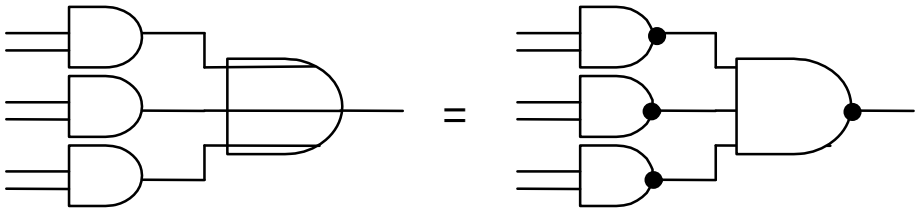
Kanal im Vergleich zu den Löchern im p-Kanal. Da beim Nand-Gatter zwei (schnelle) n-Transistoren in Reihe (also hintereinander) zu durchlaufen sind und beim Nor-Gatter zwei (langsame) p-Transisoren ist das Nand-Gatter schneller. Da es im Gegensatz zu Und- und Oder-Gattern einstufing statt zweistufig aufgebaut werden kann, ist Nand in vielen Technologien die gebräuchlichste Gatterform. Aus diesem Grunde kann man sich fragen, ob man die disjunktiven und konjunktiven Normalformen nicht auch mit Nand- und Nor-Gattern aufbauen kann.

### 2.6.7. Normalformen und Minimalformen in Nand- und Nor-Logik

Alle Funktionen, die in der Normalform bzw. in der Minimalform beschrieben sind, kann man mit Nand- Nor-Logik implementieren. Dabei gelten folgende Regeln:

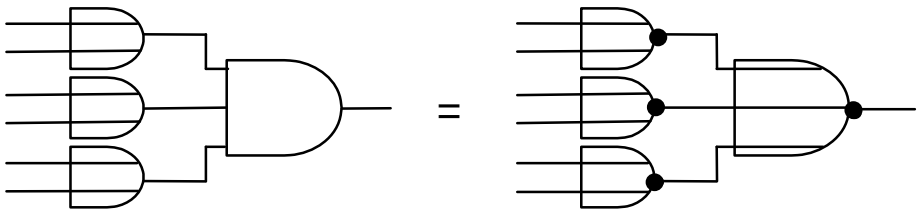
Bei der KDNF werden alle And-Gatter übernommen und invertiert, und aus dem Oder-Gatter wird ein Nand-Gatter, so entsteht Nand-Logik.

**Beispiel:**



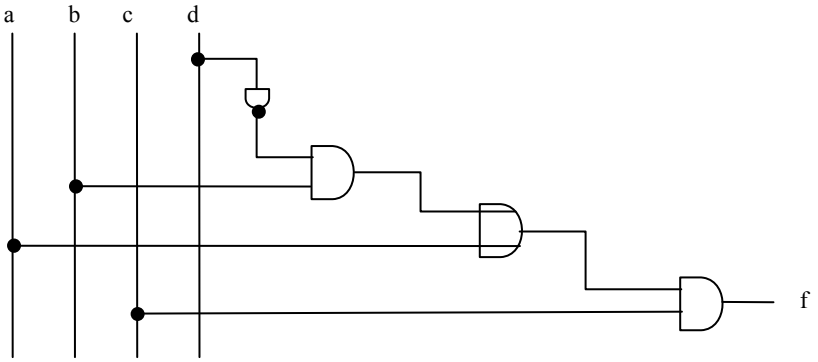
Bei KNF werden alle Oder-Gatter übernommen und invertiert, und aus And-Gatter wird ein Nor-Gatter, so entsteht Nor-Logik.

**Beispiel:**



### 2.6.8. Mehrstufigkeit

Viele boolsche Funktionen kann man so umformen, dass die Anzahl von Gattern verkleinert wird. Z. B. $f = abc + a\bar{b}c + bc\bar{d} = c(ab + a\bar{b} + b\bar{d}) = c(a + b\bar{d})$. Wir haben jetzt eine mehrstufige Funktion bekommen. Wenn jedes Gatter eine bestimmte Schaltzeit benötigt, ist eine mehrstufige Realisierung langsamer als die zweistufige in DMF oder KDNF, da die Schaltvorgänge nacheinander ausgeführt werden müssen.

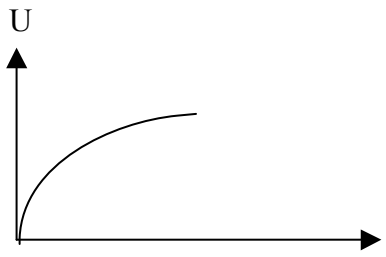


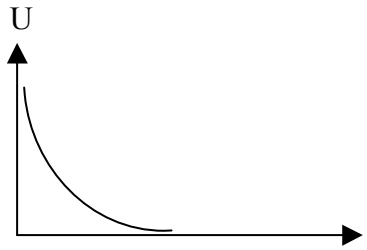

### 2.6.9. Fan-out und Fan-in

Bisher haben wir die Schaltelemente, mit denen wir gearbeitet haben als „ideale“ Bauteile benutzt, d.h. wir haben vorausgesetzt, dass Änderungen von Werten auf Signalleitungen in der Zeit 0 vollzogen werden. Ein Signal springt von 0 auf 1 und umgekehrt. Dies ist natürlich in der Realität nicht der Fall, sondern der Übergang von logisch 0 auf logisch 1 vollzieht sich als kontinuierlicher Prozess, z. B. indem eine Spannung von 0V auf 3,3V ansteigt. Im Verlaufe eines solchen Übergangs gibt es fünf Phasen:

1. Das Signal ist (nach Beendigung eines vorherigen Schaltvorgangs) im Zustand 0, d.h. die Spannung, die das Signal repräsentiert ist annähernd 0V.
2. Die Spannung beginnt anzusteigen, sie ist größer als 0V, aber sie ist deutlich näher an 0V als an 3,3V, so dass das Signal noch eindeutig als logisch 0 zu erkennen ist.
3. Das Signal ist deutlich größer als 0V aber noch deutlich kleiner als 3,3V, d.h. es kann kein logischer Wert zugeordnet werden.
4. Die Spannung hat einen Wert erreicht, der so dicht an 3,3V liegt, dass das Signal eindeutig als logisch 1 erkannt werden kann.
5. Die Spannung ist annähernd gleich 3,3V, das Signal ist logisch 1, der Schaltvorgang ist abgeschlossen.

Jede dieser Phasen ist für ein positives Zeitintervall aktiv, insbesondere die dritte, in der das Signal weder 0 noch 1 ist. Um die Schaltung insgesamt so schnell (und stabil) wie möglich zu machen, ist es natürlich das Bestreben des Schaltungsentwerfers, die Phase 3 so kurz wie möglich zu machen. Die Strategie dafür hängt einerseits von den Schwellwerten ab, bis zu denen ein Signal eindeutig als 0 bzw. ab denen es eindeutig als 1 angesehen wird, andererseits von der Technologie, die der Realisierung zugrunde liegt. Als Schwellenwerte sind 10% und 90% des Spannungspegels für die 1 (hier 3,3V) sinnvolle Richtwerte. Bei der in Abschnitt 2.6.6 vorgestellten CMOS -Technologie (sowie bei den meisten anderen Schaltkreisfamilien) kann der Schaltvorgang als Umladen einer Kapazität über einen Widerstand modelliert werden. Diese Anordnung stellt den aus der Vorlesung Technische Informatik II bekannten Tiefpass dar, dessen Spannungskurven in der Abbildung skizziert sind.




Wechsel von 0 auf 1 Wechsel von 1 auf 0

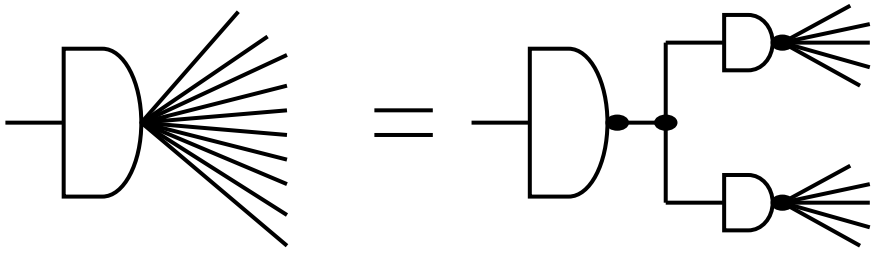
Die Zeiten des Wechsels von 0 auf 1 und von 1 auf 0 lassen sich wie folgt berechnen.

$U_a=U_e e^{-t/RC} \Rightarrow U_a/U_e=e^{-t/RC} \Rightarrow \ln(U_a/U_e)=-t/RC \Rightarrow t=RC(-\ln(U_a/U_e))$ mit $U_a=0{,}9U_e$ erhält man als Nährung $t=0{,}105RC\approx 0{,}1RC$

Man sieht also, dass sowohl der Widerstand als auch die Kapazität als linearer Faktor in die Schaltzeit eingehen. Der Widerstand ist durch den Widerstand des Transistors in den schaltenden Bauteil und die Kapazität durch die Eingangskapazität des beschalteten Bauteils gegeben.

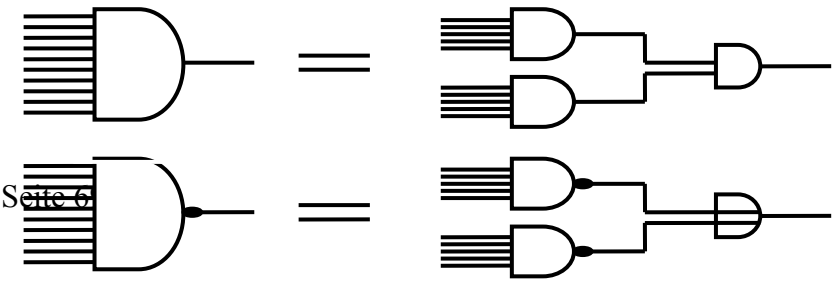
Wenn nun ein Ausgang mit n Eingänge verbunden wird, so muss der Ausgang die n-fache Kapazität eines einzelnen Eingangs auf- bzw. entladen. Die Schaltzeit vergrößert sich also entsprechend. Der Begriff Fan-out (Ausfächerung) skizziert diese Problematik. Der Fan-out eines Gatters ist die Anzahl der Eingänge (der Größe des Gattereingangs selbst), die vom Ausgang des Gatters beschaltet werden. Wenn der Fan-out nun so groß ist, dass die Schaltzeiten zu langsam werden (bei CMOS etwa bei einem Fan-out von 5-10), behilft man sich, indem man zwischen den Ausgang und die Eingänge ein Schaltnetz schaltet, das die Identität realisiert, bei dem aber der Fan-out an jeder Stelle geringer ist.

Beispiel: Fan-Out

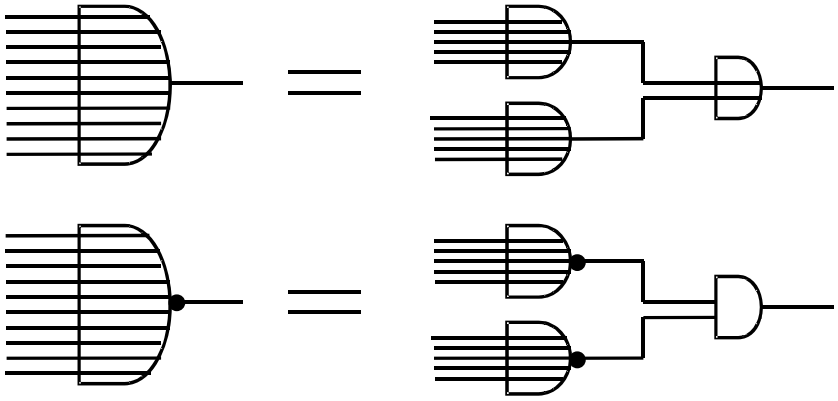


Ein ähnliches Problem tritt auf, wenn die Anzahl der Eingänge in ein Gatter zu groß wird (Fan-in). Hier wird der Widerstand des Gatters zu groß, denn die Widerstände der in Reihe befindlichen Transistoren addieren sich zum Gatterwiderstand (typisch für CMOS 1-10KΩ pro Transistor). Mit der Anzahl der Eingänge sinkt also die Schaltgeschwindigkeit. Auch hier behilft man sich mit der Verwendung von Ersatzschaltungen für Gatter mit zu großem Fan-in.

<u>Beispiel 1:</u> And, Nand



Beispiel 2: Or, Nor



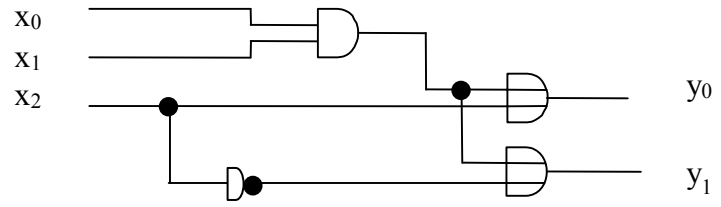
### 2.6.10. Vermaschte Logik

Gelegentlich kommt es vor, dass Schaltnetze mit mehreren Ausgängen so realisiert werden können, dass Terme (z.B. Primterme) in der disjunktiven Verknüpfung mehrerer unterschiedlicher Ausgänge auftauchen. In diesem Fall ergibt sich zusätzlich zur Minimierung die Möglichkeit der Einsparung von Gattern, da diese Terme nicht mehrfach gebildet werden müssen. Der Ausgang eines Und-Gatter, das einen solchen Term erzeugt, wird einfach mit allen Eingängen der Oder-Gatter verbunden, die diesen Term benötigen.

**Beispiel:**

$y_0 = x_0 x_1 + x_2$

$y_1 = x_0 x_1 + \overline{x}_2$




## *2.7. Standard-Schaltnetze*

Bisher haben wir gelernt, wie man von einer funktionalen Beschreibung einer booleschen Funktion über eine Wertetabelle zu einer Realisierung als Schaltnetz kommen kann. Im nun folgenden Abschnitt wollen wir einzelne Beispiele von Schaltnetzen studieren, die häufig verwendete Grundbausteine für den Aufbau komplexer Schaltnetze darstellen.

### 2.7.1. Kodierer

Kodierer wandeln einen „1 aus N Code“ in einen kompakteren Code um, meistens in den Binärcode. Wenn die i-te Eingangsleitung auf 1 ist, wird die Zahl i in binärer Verschlüsselung an die Ausgänge geführt.

Eine Voraussetzung für die einwandfreie Funktion eines Kodierers ist die Tatsache, dass jederzeit genau eine der Eingangsleitungen auf 1 ist und alle anderen auf 0. Wenn diese Bedingung verletzt wird, sind die Werte an den Ausgängen undefiniert.

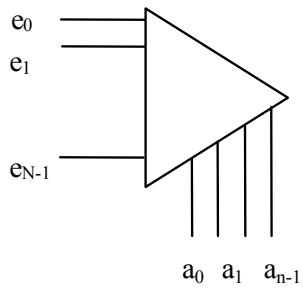
Bei N Eingangsleitungen braucht man [ld(N)] Ausgänge. ld steht für Logarithmus dualis, also Logarithmus zur Basis 2. Die eckigen Klammern bedeuten: die nächstgrößere ganze Zahl. Wenn $N = 2^n$ ist, so ist [ld(N)] = ld(N) = n. Ist N keine Zweierpotenz, so ist die Anzahl der Ausgänge der Logarithmus der nächstgrößeren Zweierpotenz.

Wir betrachten den 1-aus-16 zu 8421-Kodierer

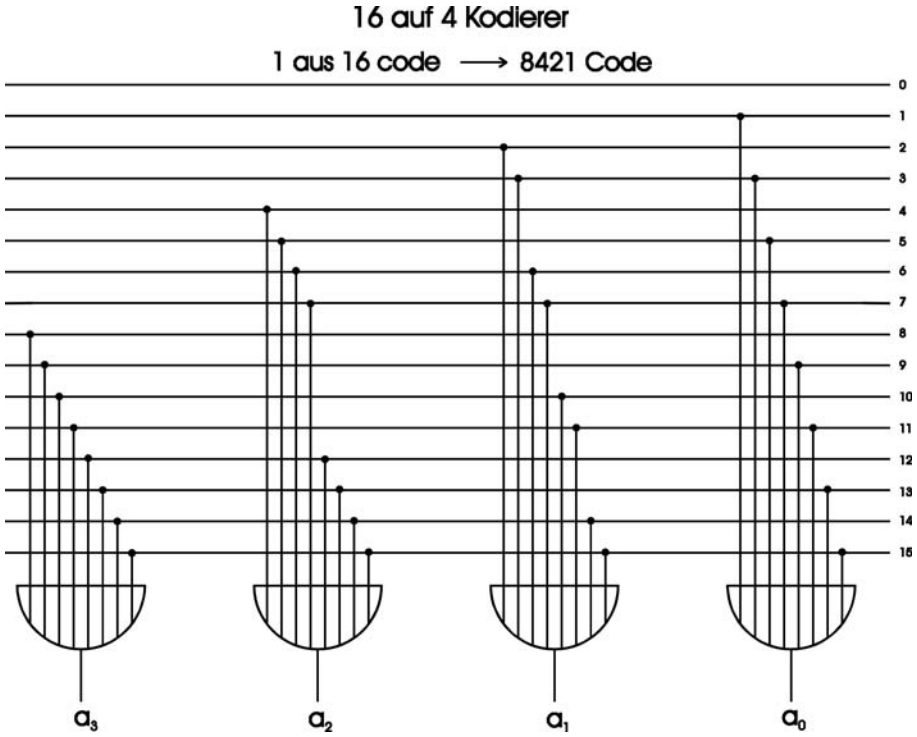
Tabelle Kodierer:

| $e_{15}$ | $e_{14}$ | $e_{13}$ | $e_{12}$ | $e_{11}$ | $e_{10}$ | $e_9$ | $e_8$ | $e_7$ | $e_6$ | $e_5$ | $e_4$ | $e_3$ | $e_2$ | $e_1$ | $e_0$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

Bild Kodierer:




Realisierung:

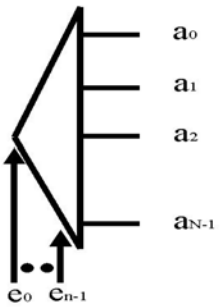



Natürlich werden die großen Oder-Gatter in der dargestellten Realisierung nicht tatsächlich in dieser einstufigen Weise realisiert (vgl. Fan-in-Problematik). Für ein 16-fach Oder bietet sich beispielsweise die Realisierung aus And- und Nor-Gattern aus Abschnitt 2.6.9 an:

### 2.7.2. Dekodierer

Ein Dekodierer ist das Gegenstück zum Kodierer: Die Eingabevariablen liefern einen codierten Wert, und die Ausgänge wandeln diesen in einen „1 aus N Code“ um. Er „entschlüsselt“ den Eingangscode und weist jeder Eingabekombination genau ein Aktivsignal auf einer einzigen Ausgabeleitung zu. Bei n Eingabeleitungen sind hier $N = 2^n$ Ausgangsleitungen erforderlich. Zur Erinnerung: Bei der Erzeugung der Minterme bei der Schaltnetzrealisierung mittels ROM wurde ein Dekodierer verwendet.

Bild Dekodierer:

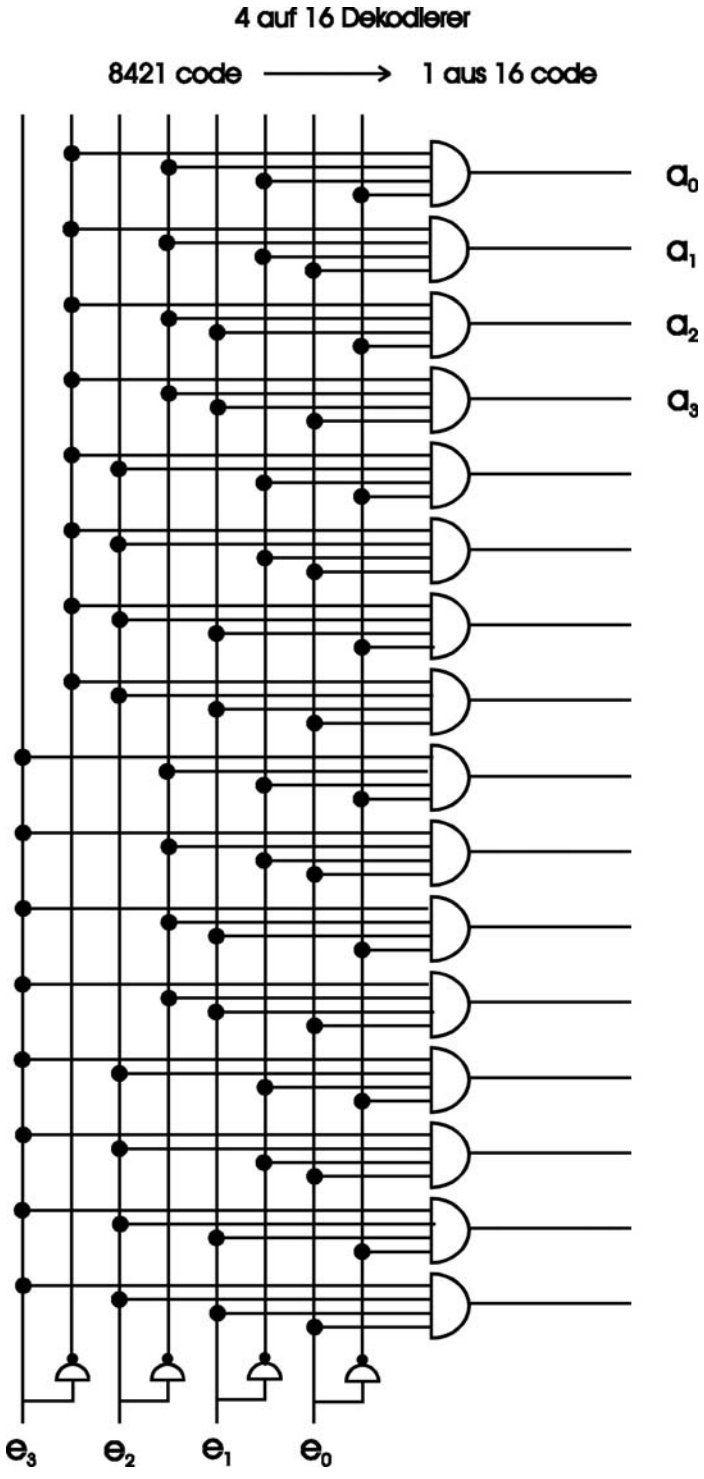



Als Beispiel betrachten wir 8421 zu 1-aus-16 Decodierer

Tabelle:

| $e_3$ | $e_2$ | $e_1$ | $e_0$ | $a_{15}$ | $a_{14}$ | $a_{13}$ | $a_{12}$ | $a_{11}$ | $a_{10}$ | $a_9$ | $a_8$ | $a_7$ | $a_6$ | $a_5$ | $a_4$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 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 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Realisierung:



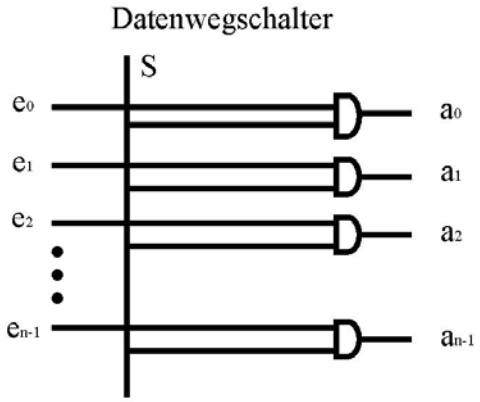

### 2.7.3. Datenwegschalter

Ein Datenwegschalter schaltet die Eingänge $e_i$ abhängig von einem Steuereingang s auf die Ausgänge $a_i$. Wenn s = 1 ist, werden die Eingänge unverändert durchgeleitet, wenn s = 0 ist, werden sie abgekoppelt, die Ausgänge werden auf 0 gelegt.

Funktionsbeschreibung: Wir haben n Eingänge ($e_0$, e1, $e_2$, ... , $e_{n-1}$), n Ausgänge ($a_0$, a1, $a_2$, ... , $a_{n-1}$) und „Schalter“ S. Die folgenden Funktionen beschreiben die Funktionsweise eines Datenwegschalters:

$a_0 = e_0S$; $a_1 = e_1S$; $a_2 = e_2S$; ... ; $a_{n-1} = e_{n-1}S$.

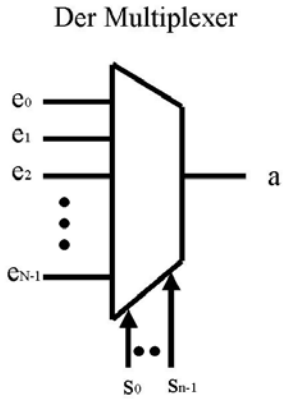
Realisierung:




### 2.7.4. Multiplexer

Ein Multiplexer (kurz Mux) wählt, abhängig von den Steuereingängen $s_i$, einen der Eingänge $e_i$ aus und schaltet diesen auf den Ausgang a. Mit n Steuereingängen können so $N = 2^n$ Signaleingänge $e_0$ , $e_1$ ,..., $e_{N-1}$ kontrolliert werden.

Funktionsbild Multiplexer:



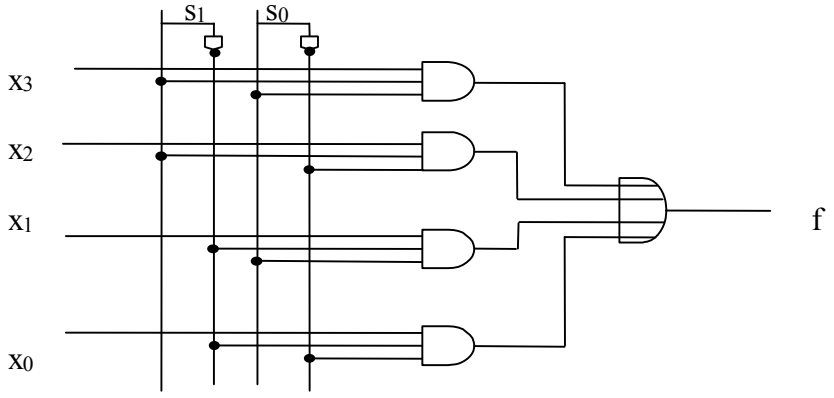

Als Beispiel sehen wir uns ein 16-auf-1 Multiplexer an.

Wertetabelle:

| $s_3$ | $s_2$ | $s_1$ | $s_0$ | a |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $e_0$ |
| 0 | 0 | 0 | 1 | $e_1$ |
| 0 | 0 | 1 | 0 | $e_2$ |
| 0 | 0 | 1 | 1 | $e_3$ |
| 0 | 1 | 0 | 0 | $e_4$ |
| 0 | 1 | 0 | 1 | $e_5$ |
| 0 | 1 | 1 | 0 | $e_6$ |
| 0 | 1 | 1 | 1 | $e_7$ |
| 1 | 0 | 0 | 0 | $e_8$ |
| 1 | 0 | 0 | 1 | $e_9$ |
| 1 | 0 | 1 | 0 | $e_{10}$ |
| 1 | 0 | 1 | 1 | $e_{11}$ |
| 1 | 1 | 0 | 0 | $e_{12}$ |
| 1 | 1 | 0 | 1 | $e_{13}$ |
| 1 | 1 | 1 | 0 | $e_{14}$ |
| 1 | 1 | 1 | 1 | $e_{15}$ |

$a = e_i$, wenn $(i)_{10} = (s_3 s_2 s_1 s_0)_2$

Beispiel: Realisierung eines 4-1-Multiplexers in disjunktiver Minimalform




### 2.7.5. Demultiplexer

Ein Demultiplexer (kurz Demux) bildet das Gegenstück zum Multiplexer. Ein Eingang e wird auf einen der möglichen Ausgänge $a_i$ gelegt in Abhängigkeiten von den Steuereingängen $s_i$. Mit n Steuereingängen kann das Eingangssignal an einen von $N=2^n$ Ausgängen $a_0$ , $a_1$ , ... , $a_{N-1}$ verbunden werden.

Funktionsbild Demultiplexer:

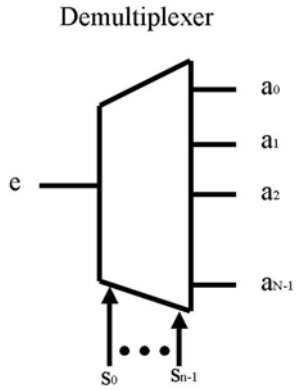
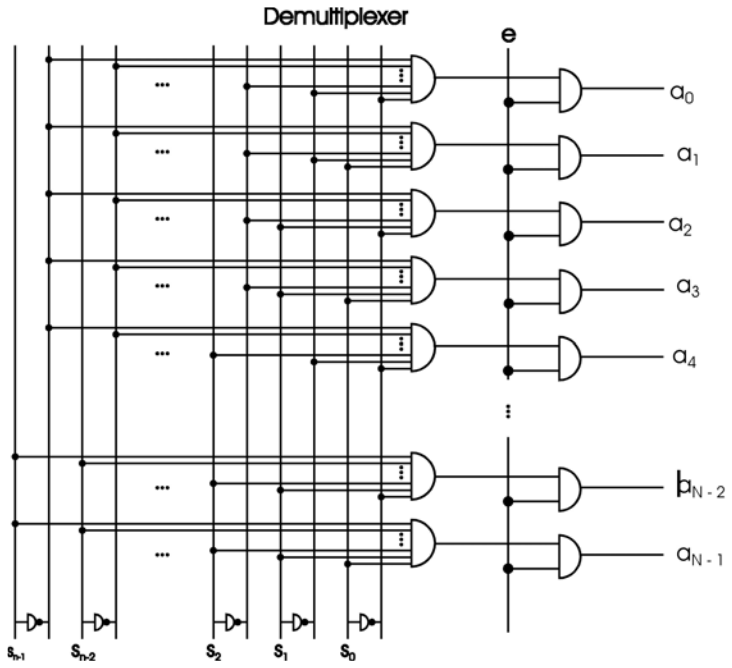



Tabelle:

# 1 auf 16 Demultiplexer

| e | $s_3$ | $s_2$ | $s_1$ | $s_0$ | $a_{15}$ | $a_{14}$ | $a_{13}$ | $a_{12}$ | $a_{11}$ | $a_{10}$ | $a_9$ | $a_8$ | $a_7$ | $a_6$ | $a_5$ | $a_4$ | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | X | X | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

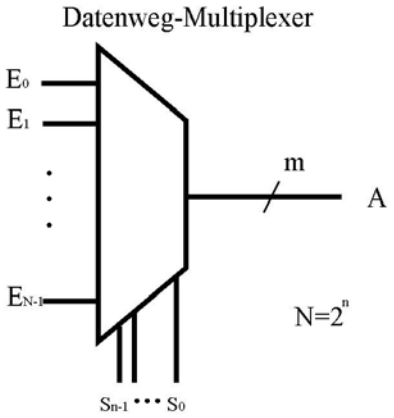
Realisierung:




### 2.7.6. Datenweg-Multiplexer

Ein Datenweg-Multiplexer ist ein Multiplexer, der gleichzeitig eine Menge von m Eingangssignalen auf m Ausgangsleitungen verschaltet. Aus einer Menge von m*N Eingängen wird also gemäß den Werten auf den Steuereingängen $s_i$, eine Teilmenge von m Eingängen ausgewählt, die auf die m Ausgänge geschaltet werden.

Bild:




### 2.7.7. Datenweg-Demultiplexer

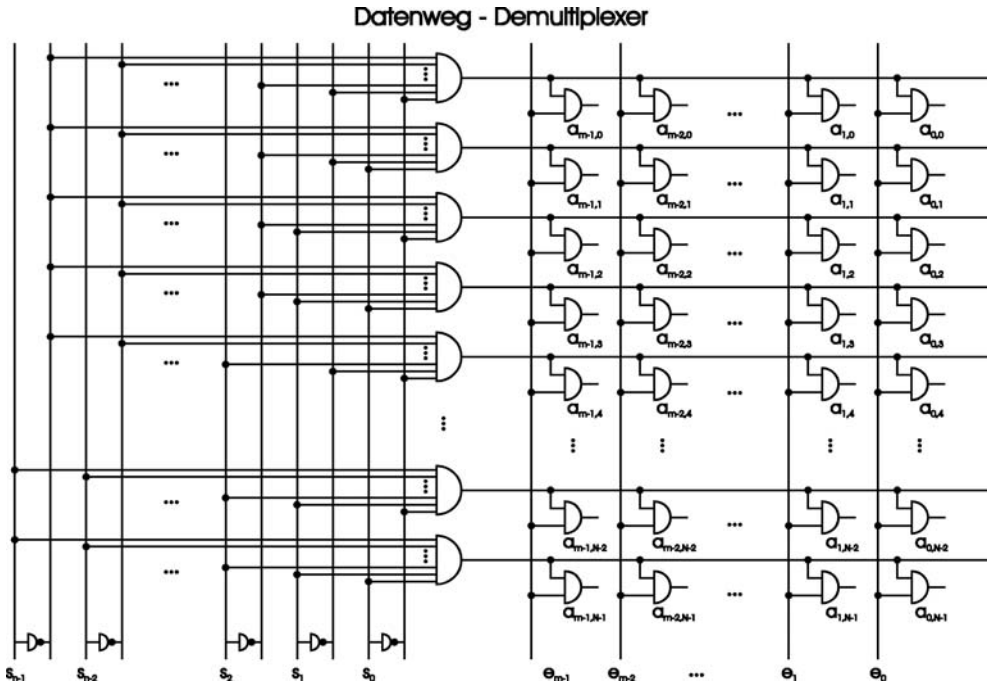
Ein Datenweg-Demultiplexer ist ein Demultiplexer, der gleichzeitig eine Menge von m Eingangssignalen auf m Ausgangsleitungen verschaltet. Aus einer Menge von m*N

Eingängen wird also gemäß den Werten auf den Steuereingängen $s_i$, eine Teilmenge von m Eingängen ausgewählt, die auf die m Ausgänge geschaltet werden.

Tabelle:

| $s_{n-1}$ | $s_{n-1}$ | ... | $s_2$ | $s_1$ | $s_0$ | A |
|---|---|---|---|---|---|---|
| 0 | 0 | ... | 0 | 0 | 0 | $A_0$=E, $A_i$=0 für i≠0 |
| 0 | 0 | ... | 0 | 0 | 1 | $A_1$=E, $A_i$=0 für i≠1 |
| 0 | 0 | ... | 0 | 1 | 0 | $A_2$=E, $A_i$=0 für i≠2 |
| 0 | 0 | ... | 0 | 1 | 1 | $A_3$=E, $A_i$=0 für i≠3 |
| 0 | 0 | ... | 1 | 0 | 0 | $A_4$=E, $A_i$=0 für i≠4 |
| | | • | | | | • |
| | | • | | | | • |
| | | • | | | | • |
| 1 | 1 | ... | 1 | 1 | 0 | $A_{N-2}$=E, $A_i$=0 für i≠N-2 |
| 1 | 1 | ... | 1 | 1 | 1 | $A_{N-1}$=E, $A_i$=0 für i≠N-1 |

Realisierung:

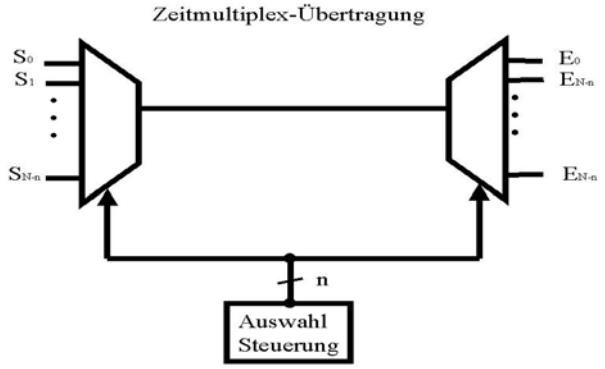



### 2.7.8. Zeitmultiplex-Übertragung

Wenn eine Menge von Paaren (Sender/Empfänger) Daten austauschen wollen, für diese aber nur ein Kanal zur Verfügung steht, kann man unter Benutzung eines Multiplexers und eines

Demultiplexers den Kanal über die Zeit an die Kommunikationspartner aufteilen. Das Bild unten stellt die Anordnung einer Zeitmultiplex-Übertragung dar. Jedes Paar $E_i$, $S_i$, bekommt den Kanal für ein Zeitintervall T, und alle Paare werden nacheinander bedient. Dies wird erreicht durch eine Zentrale Steuerung, die den Multiplexer und den Demultiplexer mit jeweils denselben Steuersignalen ansprechen. Die Steuerung verfügt über einen internen Zähler, der zyklisch von 0 bis N-1 zählt. Der Wert des Zählers stellt das Steuersignal S an den Multiplexer und den Demultiplexer dar.

Bild:




Zeitachse:

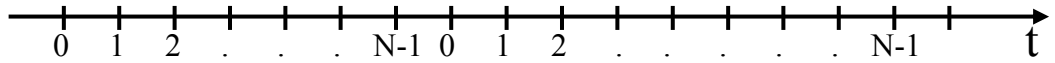



Tabelle:

| T | $S_{n-1}$ | $S_{n-2}$ | | $S_2$ | $S_1$ | $S_0$ | Verbindung |
|---|---|---|---|---|---|---|---|
| 0-Δt | 0 | 0 | ... | 0 | 0 | 0 | $E_0 \Rightarrow A_0$ |
| Δt-2Δt | 0 | 0 | ... | 0 | 0 | 1 | $E_1 \Rightarrow A_1$ |
| 2Δt-3Δt | 0 | 0 | ... | 0 | 1 | 0 | $E_2 \Rightarrow A_2$ |
| 3Δt-4Δt | 0 | 0 | ... | 1 | 0 | 0 | $E_3 \Rightarrow A_3$ |
| .<br>.<br>. | | | ... | | | | .<br>.<br>. |
| (N-2) Δt - (N-1) Δt | 1 | 1 | ... | 1 | 1 | 1 | $E_{N-1} \Rightarrow A_{N-1}$ |

### 2.7.9. Datenbus

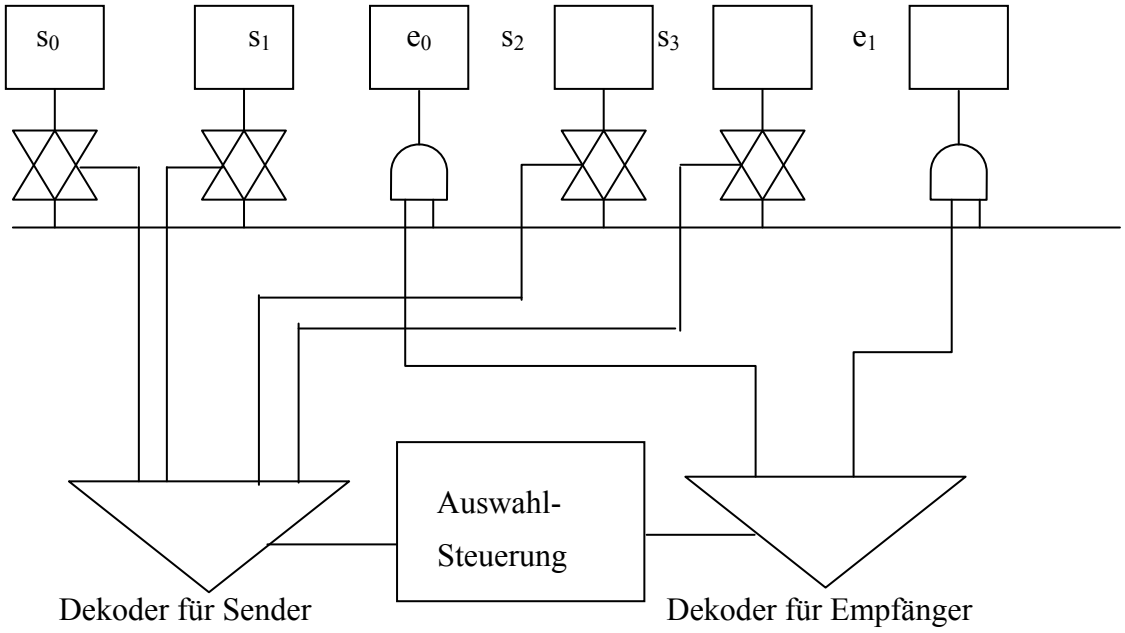
Häufig sind die Paare (Sender/Empfänger) nicht physikalisch benachbart in der Hardware eines Chips, eines Rechners oder eines Rechnersystems angeordnet. In diesem Falle müssen Multiplexer und Demultiplexer an die einzelnen Komponenten verteilt werden. Dies führt zum Konzept des Datenbus.

Ein Bus (wie Omnibus, lat.: für alle) ist ein gemeinsamer Kanal, auf den mehrere Komponenten schreibend und lesend zugreifen können. Natürlich muss geregelt werden, wer zu welchem Zeitpunkt schreibenden oder lesenden Zugriff auf den Bus bekommt. Dies wird wieder von einer zentralen Auswahlsteuerung (Bus-Master) geleistet.

Der Multiplexer aus der Zeitmultiplex-Übertragung wird hier zu einem zentralen Dekodierer, dessen Ausgänge als Steuerleitungen an die Sender geführt werden. Ist eine solche Steuerleitung auf logisch 1, so wird der Schreib-Einheit der Zugriff auf den Bus gewährt. Dies kann durch ein Transmissions-Gatter geschehen. Ist die Steuerleitung auf 0, so sperrt das Transmissionsgatter, d.h. der Ausgang ist hochohmig und es wird von dieser Einheit nicht auf den Bus geschrieben.

Der Demultiplexer aus der Zeitmultiplex-Übertragung wird hier zu einem zentralen Dekodierer, dessen Ausgänge als Steuerleitungen an die Empfänger geführt werden. Ist eine solche Steuerleitung auf logisch 1, so wird über einen Datenwegschalter der lesende Zugriff auf den Bus gewährt. Ist die Steuerleitung auf 0, so sperrt der Datenwegschalter die Übertragung vom Bus.

Bild:




Bus mit zentraler Auswahlsteuerung

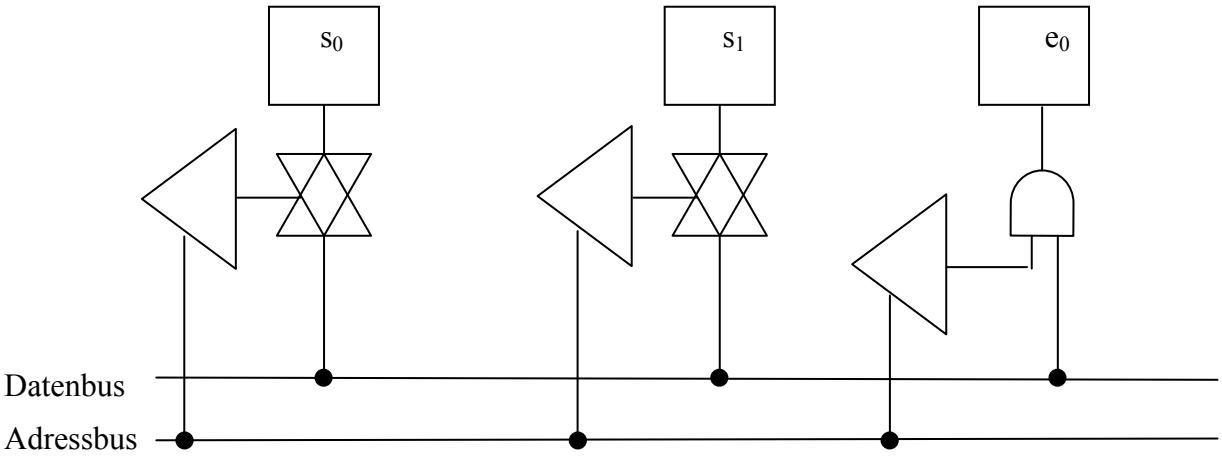
### 2.7.10. Daten- und Adressbus

Der Nachteil der Anordnung aus dem letzten Abschnitt ist darin zu sehen, dass von den Dekodierern der zentralen Auswahlsteuerung an alle Einheiten direkte Leitungen geführt werden müssen. Dies wird in der Praxis vermieden, indem man dem Datenbus zusätzliche Leitungen zuordnet, auf denen die Adressen der jeweiligen Sender und Empfänger in kodierter Form an alle Einheiten übertragen werden. Durch diese Maßnahme werden auch die Dekodierer an die Einheiten verteilt.

Die zusätzlichen Leitungen werden Adressbus genannt. Adressbusse können für Sender und Empfänger identisch sein. In Bild (siehe unten) sind sie jedoch für das einfachere Verständnis als zwei getrennte Leitungsbündel eingezeichnet. Jeder Sender hat nun lokal einen Teil des Dekodierers (meist realisiert durch eine einfache Konjunktion) zur Verfügung, über die er aus dem Sender-Adressbus ersehen kann, ob er das Schreibrecht auf den Bus besitzt. In diesem Fall schaltet er seinen Ausgang über ein Transmissions-Gatter auf den Bus.

Jeder Empfänger entschlüsselt die Adresse auf dem Empfänger-Adressbus. Findet er seine eigene Adresse vor, so liest er über seinen Datenwegschalter die Nachrichten vom Bus, solange, bis von der zentralen Auswahlsteuerung eine neue Adresse auf den Empfänger-Adressbus gelegt wird.

Bild:




Bus mit verteilter Auswahlsteuerung

## *2.8. Schaltnetzrealisierungen durch Speicher und PLAs*

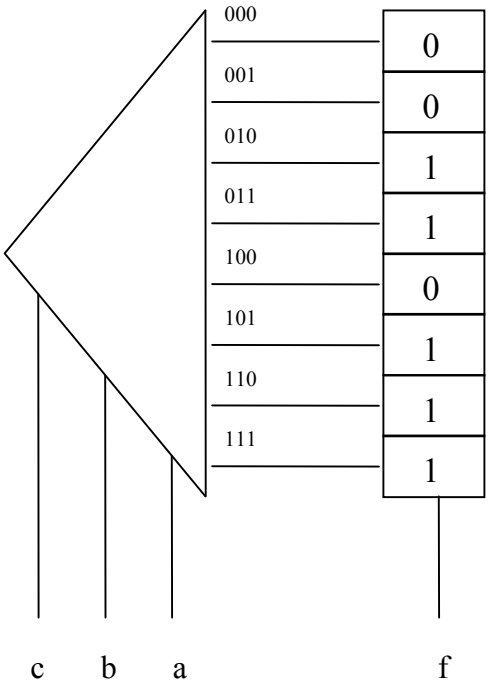
### 2.8.1. Schaltnetzrealisierungen durch Speicher (z.B. ROM, PROM, RAM)

Durch die günstige Verfügbarkeit bestimmter vorgefertigter Logik-Bausteine in unterschiedlichen Technologien können auch andere Strategien als die der Schaltnetzminimierung für die Realisierung einer Schaltung sinnvoll sein. Zum Beispiel kann man eine Schaltung in kanonischer disjunktiver Normalform unter Benutzung eines Speichers realisieren. Ein Speicher ist eine geordnete Menge von Speicherzellen, von denen jede ein Wort fester Breite (z.B. 1 Bit) aufnehmen und ggf. wiedergeben kann. Allen solchen Speichern, die wir später in der Vorlesung noch detaillierter behandeln werden, ist gemeinsam, dass jede einzelne Speicherstelle eine Adresse besitzt. Über einen Dekoder kann man diese Adresse am Speicher anlegen, und danach hat man Zugriff auf die gewählte Speicherstelle.

Wenn wir eine boolesche Funktion mit einem Speicher (anstatt mit logischen Gattern) realisieren wollen, müssen wir die Wertetabelle in diesen Speicher schreiben. Wenn wir sodann die Eingänge unserer booleschen Funktion als Adresse an den Speicher anlegen, erscheint am Ausgang automatisch der Funktionswert.

Beispiel:

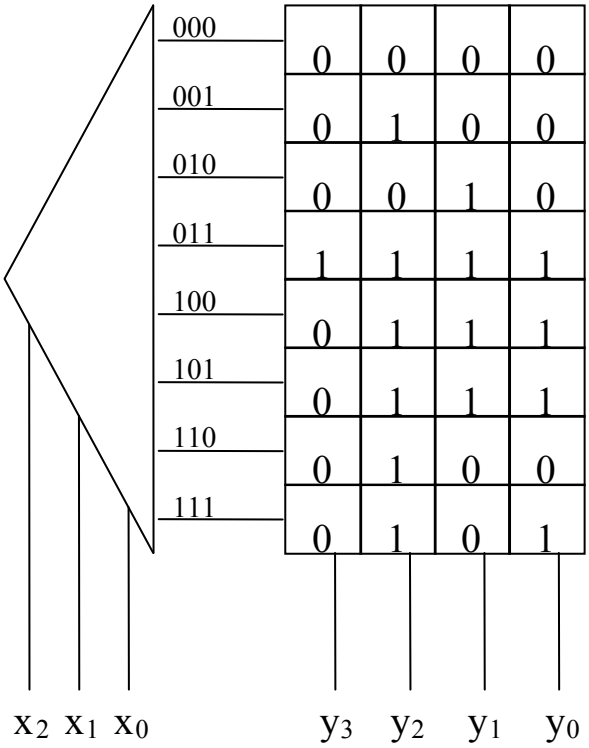
$$f = b + ac$$




Natürlich können auch mehrwertige Funktionen mit einem Speicher anstelle von Gattern realisiert werden. Dann benötigt der Speicher eine Wortbreite, die mit der Anzahl der zu berechnenden Bits der Funktion übereinstimmt.

**Beispiel:**

$$y_0 = x_0x_1 + x'_1x_2$$
$$y_1 = x'_1x_2 + x_1x'_2$$
$$y_2 = x_0 + x_2$$
$$y_3 = x_0x_1x'_2$$

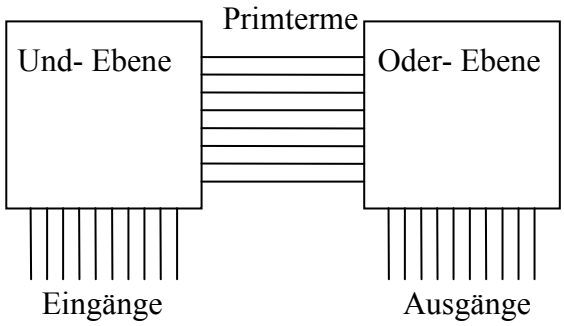


### 2.8.2. Schaltnetzrealisierungen durch PLAs

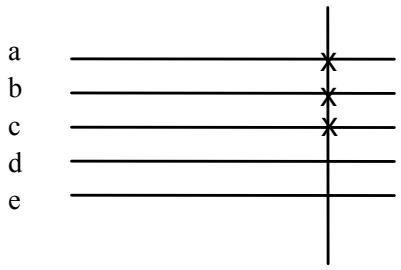
Natürlich ist eine Schaltungsrealisierung in disjunktiver Normalform im allgemeinen unverhältnismäßig aufwendig. Immerhin braucht man für die Realisierung einer Funktion mit k Eingängen $2^k$ Zeilen in der Oder-Ebene. Man kann aber die Ideen aus dem letzten Abschnitt so modifizieren, dass auch minimierte Schaltungen, z.B. in disjunktiver Minimalform, realisiert werden können. Auf diese Weise kann häufig die Anzahl der Produktterme um Größenordnungen vermindert werden. Dies führt zur Realisierung von Schaltnetzen mit PLAs (Programmable Logic Arrays).

Ein PLA besteht aus einer Und-Ebene und einer Oder-Ebene. In der Und-Ebene werden aus den Eingabewerten beliebige Produktterme (z.B. Minterme) gebildet. Diese werden in die Oder-Ebene geführt, wo sie in disjunktiver Weise verknüpft werden. Die Ausgänge der Oder-Ebene sind die Ausgänge des Schaltnetzes.

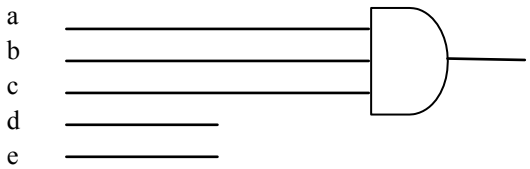
Bild: Und-Ebene, Oder-Ebene, Eingänge, Ausgänge, Primterme



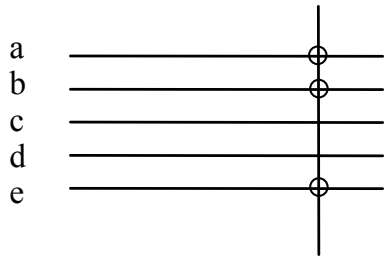

Für die einfache Darstellung benutzt man die sogenannte „verdrahtete Logik“. Bei dieser werden logische Gatterfunktionen durch zeichnen von Kreuzen (UND) und Kreisen (ODER) auf den Eingangsleitungen symbolisiert. Ein UND-Gatter mit drei Eingängen a, b und c könnte z.B. so dargestellt werden:




Die Zeichnung ist gleichbedeutend mit



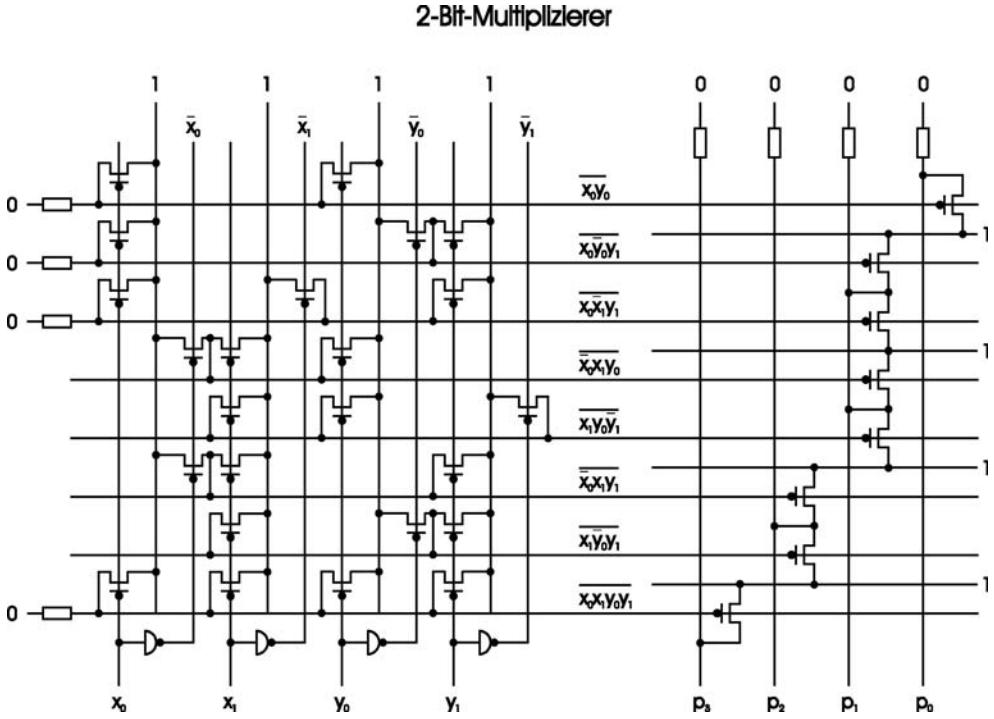
Ein ODER-Gatter mit den Eingängen a, b, e könnte so aussehen:




In der technischen Realisierung wird in der Und-Ebene und in der Oder-Ebene die Nand-Funktion (als wired Nand) gebildet, so daß die Primterme in invertierter Form von der Und-Ebene in die Oder-Ebene überführt werden. Dies Vorgehen basiert auf der in Abschnitt 2.6.7 dargestellten Form der disjunktiven Minimalform in Nand-Logik.

Das folgende Bild zeigt ein PLA mit vier Eingängen und maximal acht Produkttermen als MOS-Realisierung. Dieses PLA ist so gebrannt worden, dass der 2-Bit Multiplizierer aus der Vorlesung in DMF realisiert wird.

Bild:




Beim wired Nand arbeitet man mit pull-down-Widerständen, d. h. eine Ausgangsleitung wird über den Widerstand auf 0 „heruntergezogen“. Wenn mindestens einer der p-Transistoren jedoch eine 1 auf die Leitung schreibt, fällt an diesem Widerstand die gesamte Versorgungsspannung ab, der Ausgang ist auf 1. Die p-Transistoren sind ja leitend, wenn ihr Eingang auf 0 liegt und sie sperren, wenn eine 1 am Eingang liegt.

## 2.9. Dynamik in Schaltnetzen

Die Diskussion über Fan-out und Fan-in hatte uns bereits mit der Situation konfrontiert, dass Schaltelemente wie Gatter in der Realität ein geringfügig anderes Verhalten an den Tag legen als in der Theorie. Die Flanken auf den Signalleitungen sind nicht ideale Sprünge von 0 auf 1 oder umgekehrt, sondern kontinuierliche (exponential) Kurven. Ein wichtiger Effekt davon ist die Tatsache, dass jeder Schaltvorgang eine Zeit größer als Null benötigt. In einer ersten Näherung kann man unterstellen, dass alle Gatter die gleiche Verzögerungszeit brauchen, genannt eine Schaltzeit, $\Delta t$. Die Schaltzeit ist definiert als die Zeitdifferenz zwischen dem Zeitpunkt, an dem die Eingänge des Gatters wechseln und dem Zeitpunkt, an dem das neue Signal am Ausgang eindeutig zu erkennen ist (letzteres ist wichtig, da Signalverläufe in der Praxis so sein können, dass bei einem Signalwechsel mehrfach zwischen einem definierten und einem undefinierten Zustand gewechselt wird. Natürlich hängt das davon ab, wie „definierter Zustand“ erklärt ist. Dies wird uns noch genauer in kommenden Kapiteln beschäftigen).

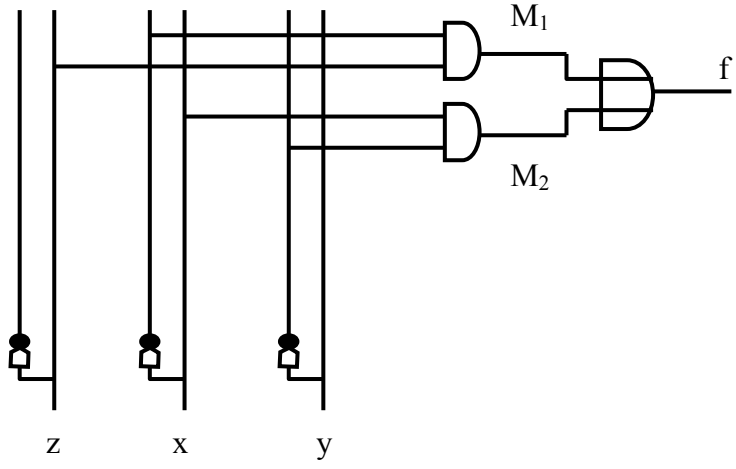
### 2.9.1. Hazards

Hazards sind kurzzeitige falsche Signale auf Leitungen, die dadurch entstehen, dass Schaltzeiten von Gattern zu Zeitdifferenzen führen zwischen den Zeitpunkten, an denen an Eingängen die Signale wechseln. Dadurch kann ein Ausgang (für kurze Zeit) einen Wert annehmen, der weder der Wert vor dem Schaltvorgang noch der Wert nach dem Schaltvorgang ist.

**Beispiel:**

$f = z\bar{x} + x\bar{y}$

Realisierung:




Seien z = 1, y = 0 und x schaltet von 1 auf 0. Dabei muss f immer gleich 0 sein. Jetzt betrachten wir wie sich f in Wirklichkeit verhält:

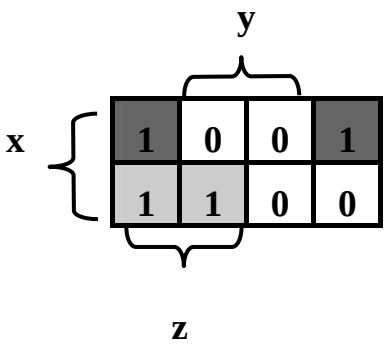
$x=1 \Rightarrow f=M_1+M_2=0+1=1;$

x=1→0 ⇒ durch die Verzögerung bei der Invertierung von x bleibt $\bar{x}$ noch ein Moment auf 0 ⇒ $M_1$=0, $M_2$=0 ⇒ f=$M_1$+$M_2$=0+0=0. Kurz danach ist $M_1$=1 und f=1+0=1. Dieses kurzzeitig falsche Signal ist ein Hazard.

### 2.9.2. Vermeidbare Hazards

Die Hazards können nur dann entstehen, wenn in der betrachteten Funktion (Beispiel siehe oben) in der KV-Diagramm die Blöcke benachbart sind.

Beispiel: KV-Diagramm zur oben angegebener Funktion:



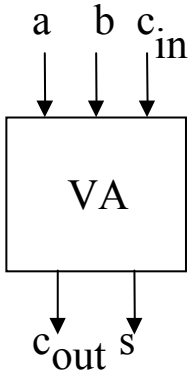

Um solche Hazards zu vermeiden, nimmt man noch einen Block dazu. Dieser Block muss nichtkritische Variable enthalten und die Arbeitsweise der Funktion dabei nicht verändern. In diesem Fall ist das $z\bar{y}$. Die neue Funktion lautet $f_{neu} = z\bar{x} + x\bar{y} + z\bar{y}$. Bei dem Übergang von x von 1 auf 0 gewährleistet $z\bar{y}$ eine 1 (z=1, y=0), sonst macht $z\bar{y}$ gar nichts. Jetzt tritt kein Hazard auf.

# 3. Computer Arithmetik

In diesem Abschnitt wollen wir einige grundlegende Techniken kennen lernen, mit denen in Computern arithmetische Operationen ausgeführt werden. Das dabei erworben Wissen werden wir später in den Abschnitten über Schaltwerke, ALU-Aufbau und Rechnerarchitektur vertiefen.

## *3.1. Addition*

Wir kennen bereits einen Volladdierer. Es ist ein Schaltnetz mit drei Eingängen a, b, $c_{in}$ und zwei Ausgängen s und $c_{out}$. Der Volladdierer ist in der Lage, drei Bits zu addieren und das Ergebnis als 2-Bit-Zahl auszugeben. Das Ergebnis liegt ja zwischen 0 und 3 und ist daher in zwei Bits zu codieren. Wir sehen hier das Schaltbild eines Volladdierers und seine Wertetabelle:

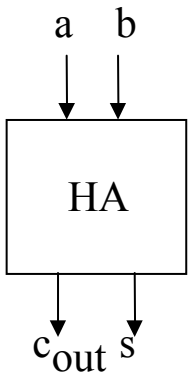



| a | b | $c_{in}$ | s | $c_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

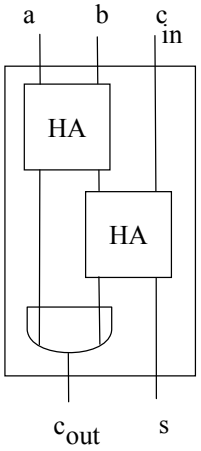
Häufig realisiert man einen Volladdierer nicht in DMF sondern in einer mehrstufigen Form, wobei man sogenannte Halbaddierer benutzt. Halbaddierer sind Schaltnetze, die zwei Bits

addieren können (und demzufolge ein Ergebnis im Bereich 0 bis 2 produzieren). Durch Zusammenschalten von zwei Halbaddierern und einem Oder-Gatter erhält man die Funktionalität eines Volladdierers. Wie sehen im folgenden das Schaltsymbol eines Halbaddierers, seine Wertetabelle und den Aufbau eines Volladdierers aus Halbaddierern.

| **a** | **b** | **s** | **$c_{out}$** |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |




Volladdierer aus zwei Halbaddierern und einem Oder- Gatter

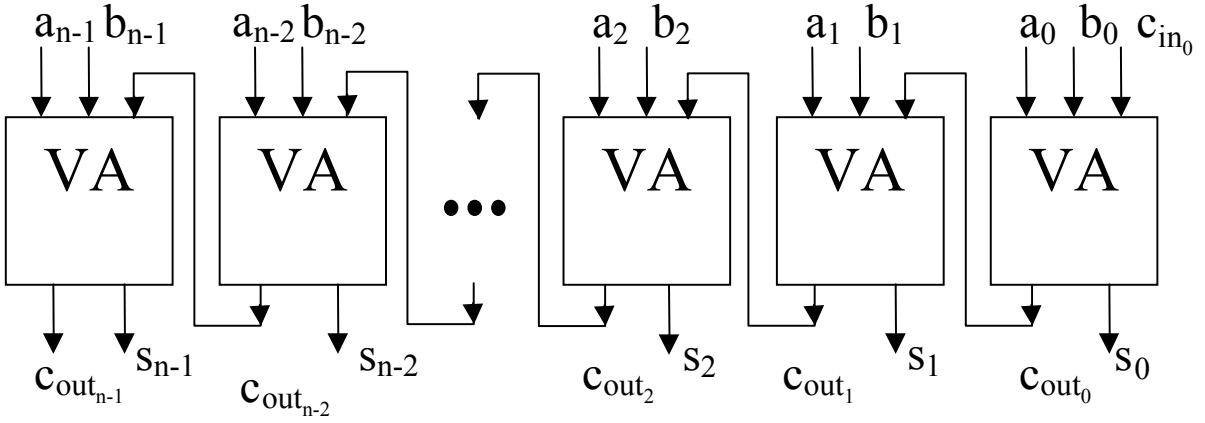



Nun wollen wir aber in der Regel längere Operanden addieren, zum Beispiel die Binärzahlen $a=a_{n-1}a_{n-2}...a_1a_0$ und $b=b_{n-1}b_{n-2}...b_1b_0$. Natürlich könnte man ein dafür erforderliches Addierwerk in KDNF oder DMF aufbauen. Dies bringt aber eine Reihe von Problemen mit sich:

- für jedes n ergibt sich eine völlig andere Realisierung

- das Fan-in und das Fan-out an den Gattern wächst polynomiell mit n

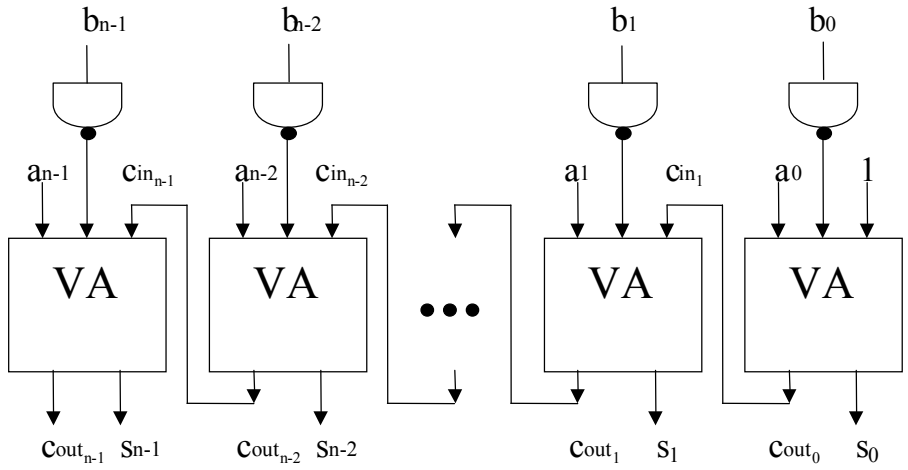
Insbesondere wegen dieser zweiten Eigenschaft ist der zweistufige Aufbau z. B. in DMF nicht sinnvoll. Statt dessen verwendet man im einfachsten Fall eine Kette von Volladdierern, die im Grunde genau das machen, was wir von der Addition in der „Schulmethode“ kennen. Man beginnt mit den LSBs (least significant bits), addiert diese, erzeugt einen Übertrag, mit dessen Kenntnis man das nächste Bit bearbeiten kann, usw. Ein entsprechendes Schaltnetz sieht dann so aus:




Das Ergebnis der Addition von zwei n-Bit-Zahlen ist eine n+1-Bit Zahl. Diese ist repräsentiert durch die Ausgänge $c_{n-1}s_{n-1}s_{n-2\ldots}s_2s_1s_0$.

Einen solchen Addierer nennt man einen **ripple-carry-adder**. Sein Vorteil ist der einfache und modulare Aufbau. Sein wesentlichster Nachteil wird bereits durch diesen Namen ausgedrückt: Wenn die Operanden gerade eine ungünstige Bit-Kombination aufweisen, muss die Carry- (übertrags-) -Information durch alle Volladdierer hindurch von der Stelle mit der geringsten Wertigkeit bis zur Stelle mit der höchsten Wertigkeit hindurchklappern (rippeln). Damit ergibt sich die Schaltzeit eines ripple-carry-adders als proportional zur Zahl n der Stellen. Dies ist insbesondere dann ein Problem, wenn in unserem Rechner ein Zahlenformat mit vielen Bits (z.B. 64 Bits) verarbeitet werden soll. Sicher wollen wir den Maschinentakt nicht so langsam machen, dass in einem Takt 64 Volladdierer nacheinander schalten können. Wir werden bald sehen, wie man dieses Problem behandeln kann.

Zunächst wollen wir uns aber damit beschäftigen, wie man mit einem Addierer auch subtrahieren kann. Wir wissen bereits: das Zweierkomplement einer Zahl lässt sich berechnen als Einerkomplement plus 1. Ferner ist das Einerkomplement die bitweise Negation der Zahl. Wenn wir nun die Addition der 1 über den Carry-Eingang $c_{in_0}$ erledigen, können wir mit dem Schaltnetz auf dem folgenden Bild a-b berechnen, indem wir zu a das Zweierkomplement von b hinzuaddieren:

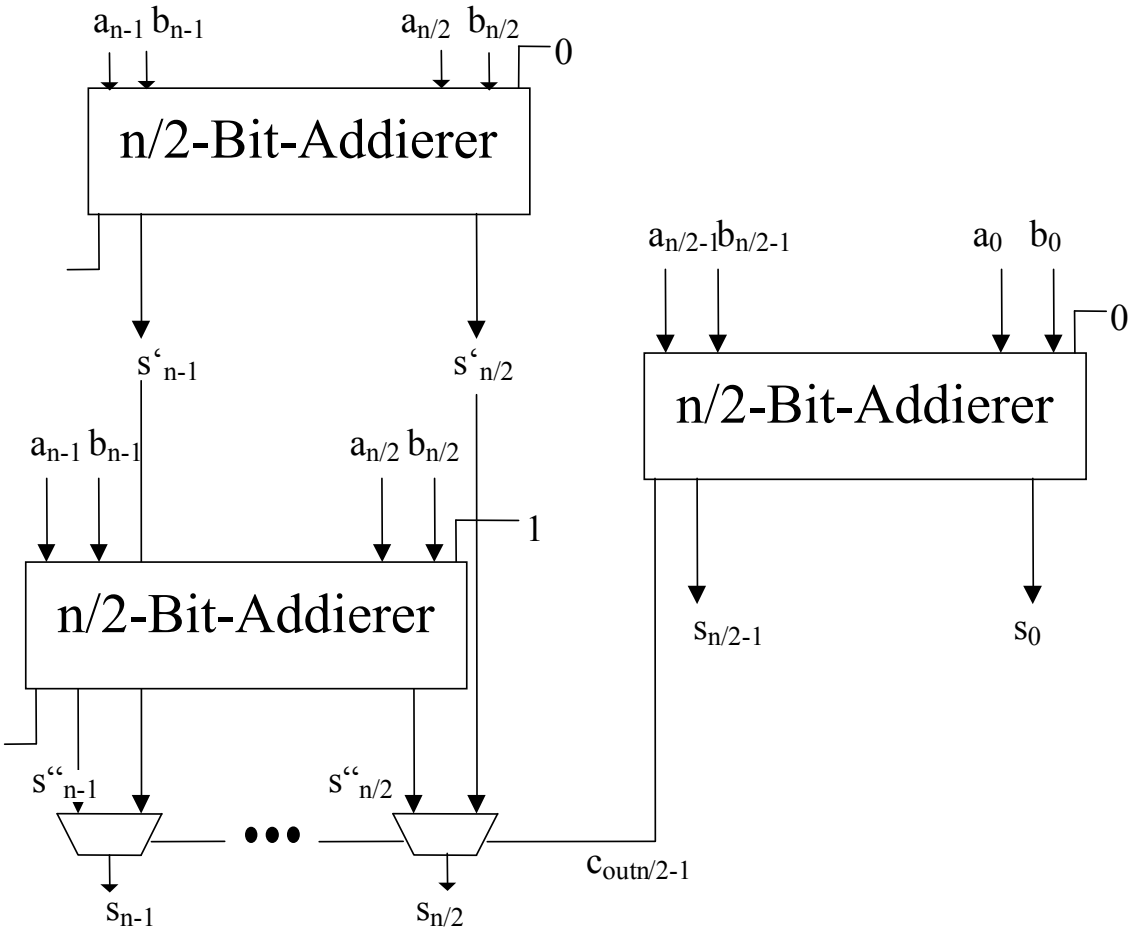



### *3.2. Schnellere Addition und Subtraktion*

Wodurch ergibt sich die Schaltzeit für eine Addition oder Subtraktion? Durch die Anzahl der Volladdierer, durch die ein Carry nacheinander hindurchklappern muss. Wenn wir z. B. +1 und -1 addieren, liegt an den Eingängen des Addierers a = 00000001 und b = 11111111. Bei der Addition entsteht an der letzten Stelle ein Übertrag, dieser bewirkt an der vorletzten Stelle einen Übertrag usw. bis hin zur ersten Stelle, wo schließlich auch ein Übertrag entsteht. Der Zeitaufwand ist also die Anzahl der Stellen, durch die ein Übertrag hindurchwandern muss. Wenn jeder Volladdierer die Zeit $t_{VA}$ benötigt, ist die Gesamtzeit also n * $t_{VA}$. Wie kann man diese Zeit nun vermindern. Eine hübsche Lösung, die auch in der Praxis der Rechnerarchitektur häufig Verwendung findet, bietet der **carry-select-adder**.

Die Idee ist folgende: Der Addierer wird in zwei gleich lange Hälften unterteilt. Und für beide Hälften wird gleichzeitig mit der Addition begonnen. Bei der linken (höher signifikanten) Hälfte wissen wir aber nicht, ob am Carry-Eingang des rechtesten Volladdierers eine 1 oder eine 0 ankommt. Deshalb führen wir die Addition der linken Hälfte gleichzeitig zweimal aus, einmal mit einer 0 am Carry-Eingang und einmal mit einer 1. Wenn die rechte Hälfte mit ihrer Addition fertig ist, kennen wir das eingehende Carry der linken Hälfte. Somit wissen wir, welches der Ergebnisse das richtige ist, das wir sodann auswählen (select). Das andere (falsche) Ergebnis wird einfach verworfen.

Die Auswahl geschieht über eine Menge von Multiplexern, die vom eingehenden Carry gesteuert werden. Das Prinzip ist auf der folgenden Folie dargestellt.



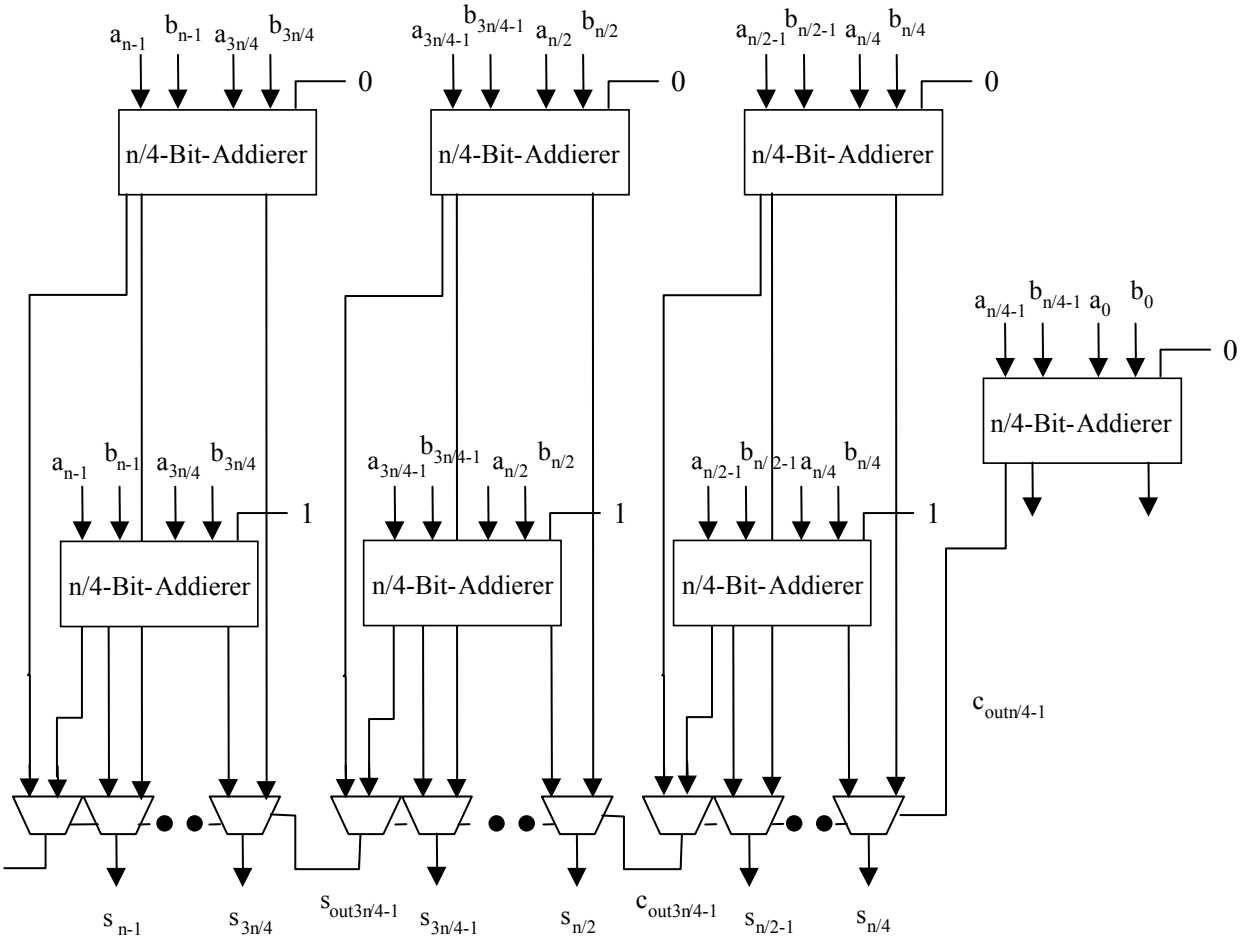

Wir sehen sofort: der Aufwand an Gattern ist etwas mehr als eineinhalb mal soviel wie beim ripple-carry-adder. Wie ist nun der Schaltzeitaufwand für einen solchen Addierer. Da alle n-2-Bit-Addierer gleichzeitig arbeiten benötigen wir nur noch die halbe Zeit, nämlich n/2 * $t_{VA}$ für die Addition. Dazu kommt noch eine kleine konstante Zeit für die Multiplexer also ist die Gesamtzeit gleich n/2 * $t_{VA}$ + $t_{MUX}$

Wir haben also etwa einen Faktor 2 in der Zeit gewonnen. Nun lässt sich dieses Prinzip natürlich wiederholt anwenden: Anstelle von Addierern der Länge n/2 können wir auch solche der Länge n/4 oder n/8 usw. verwenden. Alle solchen Addierer (außer dem am wenigsten signifikanten) werden doppelt ausgelegt, wovon einer mit einem Carryeingang 0 und der andere mit einem Carryeingang 1 arbeitet. Welches der Ergebnisse schließlich verwendet wird, entscheidet das Carry der nächst niedrigeren Stufe.

Das folgende Bild zeigt das Ergebnis dieser Technik für eine Unterteilung in vier Abschnitte. Die Laufzeit reduziert sich auf n/4 * $t_{VA}$ + 3$T_{MUX}$. Allgemein gilt für eine Unterteilung in m Abschnitte:

$$t_{Gesamt} = n/m * t_{VA} + m-1 * T_{MUX}$$

Diese Gesamtzeit nimmt ein Minimum an für $m = n^{1/2}$, also ist die Addition mit einem Carry-select-Addierer in $O(n^{1/2})$, während der Ripple-carry-Addierer in O(n) arbeitete.

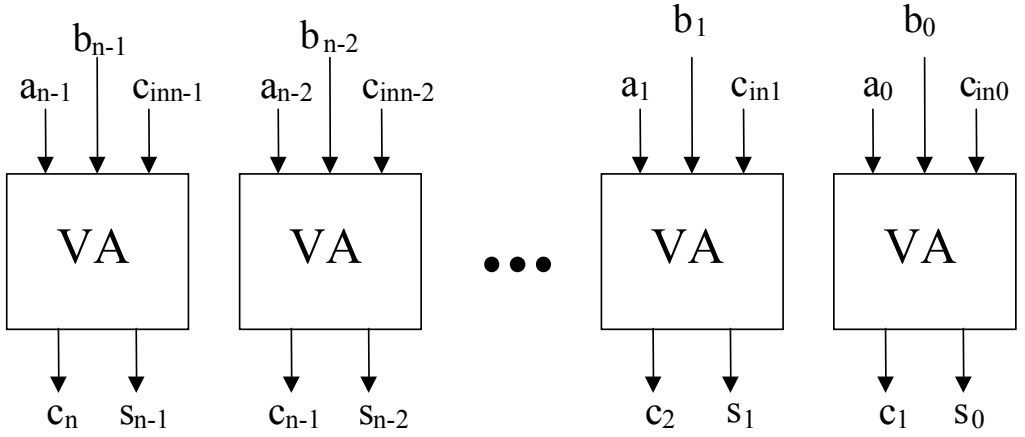



Ein anderer Ansatz ist die Verwendung einer redundanten Zahlendarstellung für die Zwischenergebnisse. Diese Technik stellt sicher, dass ein eventuell auftretendes Carry nur einen Einfluss in seinem unmittelbaren Bereich hat, aber nicht zu einer „Kettenreaktion" führen kann, wie beim ripple-carry-adder. Der hier vorgestellte Addierer heißt **carry-save-adder**.

Die Idee besteht darin, nicht zwei Operanden zu einem Ergebnis zu addieren, sondern ***drei*** Operanden zu ***zwei*** Ergebnissen. Dies erscheint zwar für unser Verständnis zunächst unnatürlich, es hat aber den Vorteil einer sehr einfachen und schnellen Realisierung. Die Anwendung eines solchen Carry-save-Addierers ist immer dann sinnvoll, wenn man mehrere Additionen nacheinander ausführen möchte. Und dies wiederum ist in Multiplizierern erforderlich. Wir werden daher sehen, wie ein extrem schneller Multiplizierer aus Carry-save-Addierern aufgebaut werden kann.

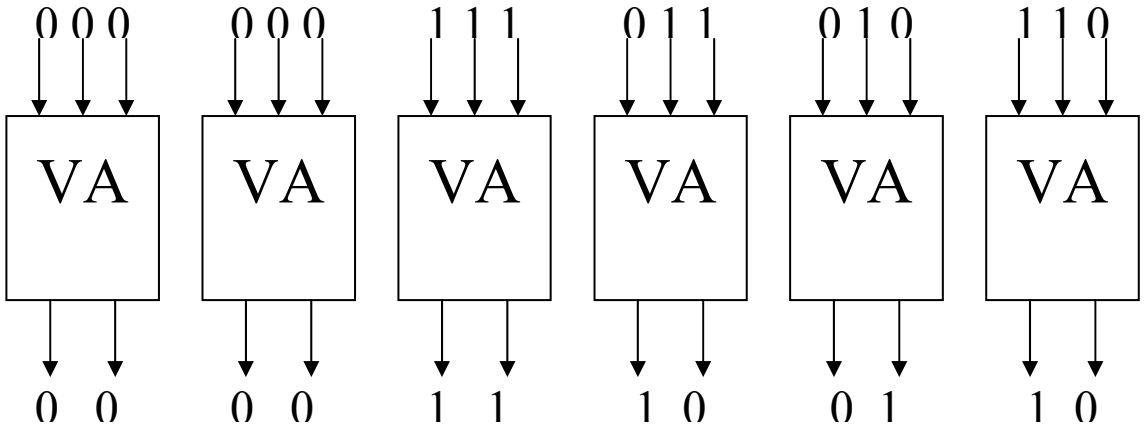
Der Aufbau eines Carry-save-Addierers ist lediglich die Parallelschaltung von n Volladdierern. Diese sind nun nicht verkettet, sondern jeder liefert zwei Ausgabebits, ein s-Bit und ein c-Bit. Aus diesen wird ein s-Wort und ein c-Wort gebildet, die zusammen die beiden Ergebnisworte darstellen. Das c-Wort wird durch eine 0 an der am wenigsten signifikanten Stelle ergänzt und das s-Wort um eine 0 an der höchst signifikanten Stelle.

Somit ist die Summe aus s- und c-Wort gleich der Summe der drei Operandenworte a, b, $c_{in}$.




**Beispiel:**

Wir addieren die Worte a = 001001, b = 001111, $c_{in}$ = 001100




Die Ergebnisworte sind also c = 011010 und s = 001010. Wenn wir die drei Operanden im Dezimalsystem addieren, kommt 36 heraus. Dasselbe Ergebnis bekommen wir, wenn wie c und s addieren. Man beachte, dass bei dieser Art der Addition kein Carry „durchklappern" kann. Die Zeit für eine Schaltung ist also $t_{VA}$ und somit in O(1).

Wo können wir diese Art von Addition sinnvoll einsetzen?

Angenommen, wir müssen eine Kolonne von 64 Zahlen addieren. Dann können wir diese mit 62 Carry-Save-Additionen zusammenzählen, so dass schließlich zwei Ergebnisworte berechnet werden. Diese müssen dann mit einem „richtigen" Addierer, z.B. einem Carry-Select-Addierer zu einem Endergebnis zusammengezählt werden. Die 62 Additionen benötigen 62 * $t_{VA}$. Die letzte Addition benötigt 8 * $t_{VA}$ + 7 $t_{MUX}$. Der Zeitaufwand gesamt ist also 70$t_{VA}$ + 7 $t_{MUX}$. Hätten wir die gesamte Addition mit einem Carry-Select-Addierer gemacht, würden wir zusammen 63 * ( 8 $t_{VA}$ + 7 $t_{MUX}$) = 504 $t_{VA}$ + 441 $t_{MUX}$ .

Man sieht, um wie viel sparsamer die Carry-Save-Addition in diesem Falle ist. Allgemein gilt: Wenn wir m Additionen der Länge n Bit machen wollen, benötigen wir mit einem Ripple-Carry-Addierer Zeit O(n*m), mit einem Carry-Select-Addierer Zeit O($n^{1/2}$*m) und mit

einem Carry-Save-Addierer (mit nachgeschaltetem Carry-Select-Addierer für den letzten Schritt) Zeit $O(n^{1/2}+m)$.

Die optimale Zeit bei der Addition zweier Zahlen erhält man mit einem sogenannten Carry-Lookahead-Addierer. Dieser benötigt nur die Zeit O(log n) für eine Addition. Aus Zeitgründen wird dieser Addierertyp aber an dieser Stelle noch nicht behandelt.

### *3.3. Multiplikation*

Bei der Multiplikation nach der Schulmethode wird für jede Stelle eines Operanden das Produkt dieser Stelle mit dem anderen Operanden berechnet. Danach werden alle diese Produkte addiert. An dieser Stelle können wir den soeben erlernten Carry-Save-Addierer einsetzen, denn jetzt haben wir den Fall einer großen Anzahl von Operanden, die addiert werden müssen.

Beispiel:

```
10110011 * 10010111

  10110011
   00000000
    00000000
     10110011
      00000000
       10110011
        10110011
         10110011
-----------------
0110100110010101
```

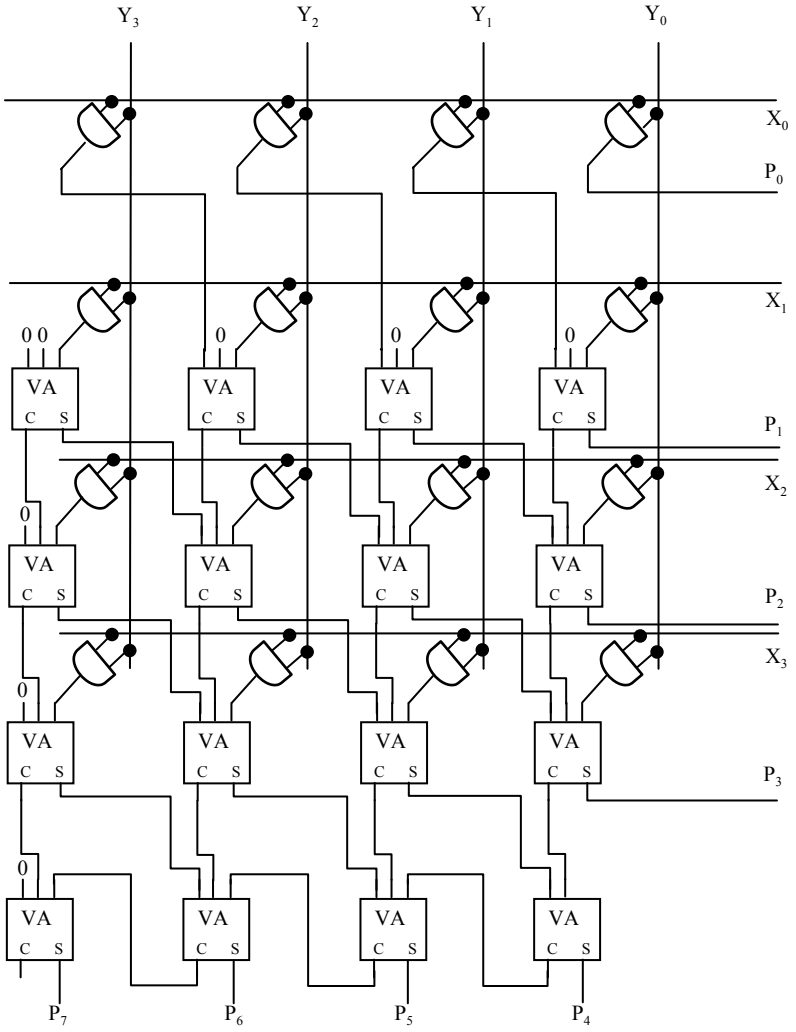
Ein Carry-Save-Multiplizierer für zwei Operanden der Länge n besteht aus $n^2$ AND-Gattern, die gleichzeitig alle erforderlichen 1-Bit-Multiplikationen ausführen (Die binäre Multiplikation von 1-Bit Zahlen ist gerade das logische „UND“). Danach sind nur noch alle Teilprodukte (wie bei der Schulmethode) zu addieren. Dies geschieht nun in der bekannten 3-auf-2 Operanden Manier, die wir soeben beim Carry-Save-Addierer kennengelernt haben. Am Ende ist für die höchstsignifikanten n Bits noch eine (klassische) 2-auf-1-Operanden-Addition erforderlich. Diese wird mit einem konventionellen Addierer ausgeführt.

Wie lang ist die Verarbeitungszeit für eine solche Multiplikation? Wir müssen in diesem Schaltnetz den „kritischen Pfad“ suchen, also den Pfad, bei dem ein Signal durch die maximale Anzahl von Schaltelementen hindurchwandern muss, bevor das Endergebnis berechnet ist. Dieser Pfad besteht zunächst einmal aus den n-2 Stufen, bei denen jeweils ein

Operand neu hinzuaddiert wird plus die n-1 Volladdierer, durch die ein Carry bei der letzten Addition hindurchklappern muss (wir setzen hier einen ripple-carry-adder voraus).

Ein Beispiel für einen solchen 4-Bit Multiplizierer sehen wir auf dem nächsten Bild.

4-Bit-carry-save-Multiplizierer



### *3.4. Division*

Einige Prozessoren haben eigene Divisionseinheiten, die in der Regel ähnlich des Carry-Save-Adders eine interne Darstellung der Zwischenergebnisse durch zwei Worte benutzt.

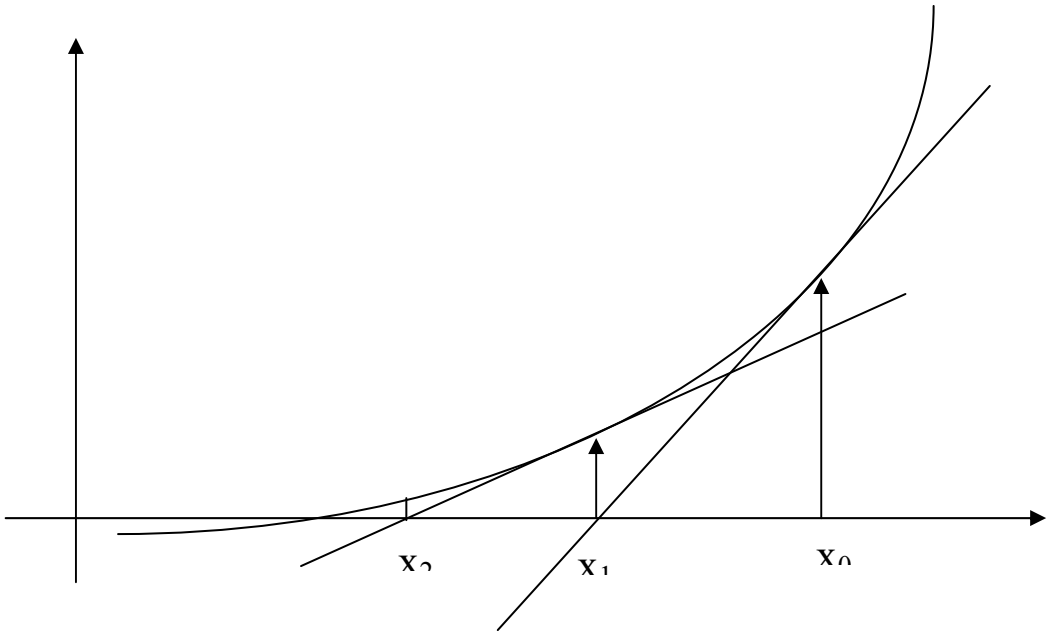
Andererseits ist die Division eine seltene Operation. Daher (make the common case fast) verzichten viele Prozessoren auf eigene Hardware für die Division, sondern implementieren sie in Software. Ein gängiges Verfahren dafür ist die Newton-Raphson-Methode, die wir hier kennen lernen wollen.

Vorweg sei erwähnt, dass frühere Rechner (< 1980) die Division in der Regel (ebenfalls in Software) entsprechend der Schulmethode ausführten, d.h. die Ergebnisbits werden eins nach dem anderen berechnet durch Vergleich des Divisors mit den verbleibenden höchstsignifikanten Stellen des Dividenden. Wenn der Divisor größer ist, ergibt sich ein Bit 0 sonst ein Bit 1. Im letzteren Falle wird sodann der Divisor von den höchstsignifikanten Stellen des Dividenden subtrahiert und eine weitere Stelle des Dividenden wird für die nächste Ergebnisstelle herangezogen.

Dieses Verfahren hat natürlich die Komplexität von $O(n^2)$, wenn der Dividend n Stellen hat. Genauer: Man braucht n Schritte, und in jedem Schritt muss ein Vergleich und eine Subtraktion ausgeführt werden. Das erwies sich zu Zeiten steigender Rechenleistung als zu langsam. Daher suchte man nach Verfahren, die in weniger Schritten zu genauen Ergebnissen führten.

Die Idee des Newton-Verfahrens ist die Approximation der Nullstelle einer Funktion durch Konstruktion einer Folge von Werten, die sehr schnell gegen die Nullstelle konvergiert. Man beginnt damit, dass man die Tangente an die Funktion in einem geschätzten Anfangswert anlegt und deren Schnittpunkt mit der Abszisse als nächsten Folgenwert berechnet. Nun legt man an dessen Funktionswert die Tangente an usw. Wenn die Funktion bestimmte Bedingungen erfüllt und wenn der Anfangswert geeignet gewählt ist, konvergiert diese Folge gegen die Nullstelle.

**Beispiel:**



Für zwei Werte $x_i$ und $x_{i+1}$ in dieser Folge gilt:

$$f'(x_i) = \frac{f(x_i)}{x_i - x_{i+1}}$$

Aufgelöst nach $x_{i+1}$ bedeutet das

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Betrachten wir nun f(x) = 1/x - B . Diese Funktion hat in 1/B eine Nullstelle. Wenn wir in die obige Formel einsetzen, ergibt sich

$$x_{i+1} = x_i - \frac{\frac{1}{x_i} - B}{-\frac{1}{x_i^2}} = x_i + \left(x_i - Bx_i^2\right) = x_i\left(2 - Bx_i\right)$$

Damit haben wir eine sehr einfache Iterationsformel, die mit zwei Multiplikationen und einer Subtraktion für einen Iterationsschritt auskommt.

Wie viele Schritte benötigen wir aber, oder anders gefragt, wie schnell konvergiert die Folge?

Betrachten wir den Fehler ε, also die Differenz des Folgenwertes $x_i$ von der gesuchten Nullstelle 1/B. Es gilt:

$$x_{i+1} = 2\left(\frac{1}{B} - \varepsilon\right) - B\left(\frac{1}{B} - \varepsilon\right)^2 = \frac{1}{B} - B\varepsilon^2$$

Somit ist der Fehler nach der nächsten Iteration nur noch $B\varepsilon^2$. Wenn B nun zwischen 0 und 1 liegt, bedeutet dies, dass wir eine quadratische Konvergenz haben, genauer: wenn das Ergebnis nach der i-ten Iteration bereits auf m Bits genau ist, ist es nach der i+1-ten Iteration auf 2m Bits genau.

Wir müssen also dafür sorgen, dass B zwischen 0 und 1 liegt und dass $x_0$ auf 1 Bit genau ist. Dann wird $x_1$ auf zwei Bits genau sein, $x_2$ auf vier Bits usw., $x_i$ auf $2^i$ Bits genau.

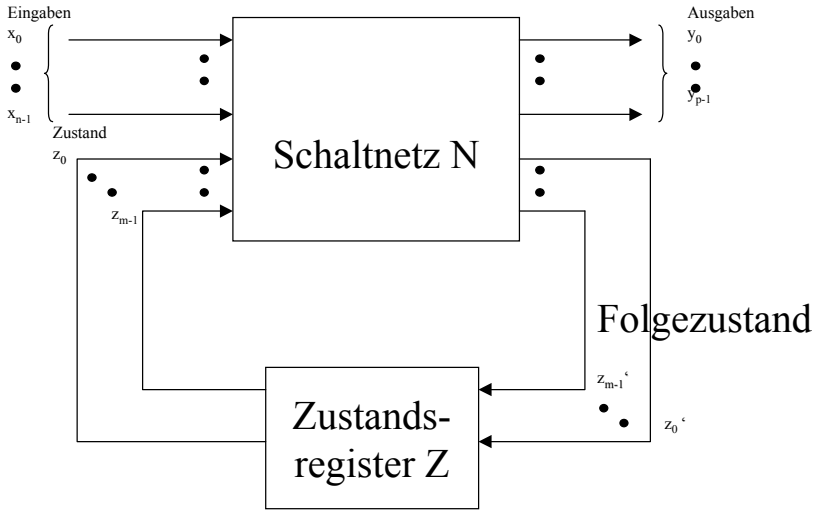
Angenommen, wir müssen a/b berechnen. Dann können wir dies mit einer Multiplikation als a * 1/b berechnen, wobei wir in der Lage sein müssen, den Kehrwert der Zahl b also 1/b zu ermitteln. Wenn b zwischen 0 und 1 ist und die erste Stelle nach dem Komma eine 1 ist (also normalisiert), dann geht das mit obiger Iteration. Wenn nicht, müssen wir $B = 2^k * b$ nehmen mit einem geeigneten k, so dass B normalisiert ist. Sodann berechnen wir 1/B und multiplizieren dies schließlich mit $2^{-k}$ (Bitverschiebung).

Fazit: Durch die Iteration wird der Aufwand von 2n Operationen auf 3log n Operationen reduziert, nämlich log n Iterationen und in jeder drei Operationen. Bei einer 64-Bit Division ist das eine Reduktion von 128 auf 18 Operationen.

# 4. Schaltwerke

Bisher haben wir uns nur mit Schaltnetzen befasst, also Schaltungen aus Gattern, die die Ausgaben als eine Funktion der Eingaben unmittelbar (durch Schaltvorgänge) berechnen. Diese Schaltnetze (engl. combinatorial circuits) haben keine Zustände, also kein Gedächtnis, in dem frühere Schaltvorgänge eingespeichert sind. Schaltnetze können nicht abhängig von einem gespeicherten Zustand auf die eine oder andere Weise auf die Eingaben reagieren. Dies ist aber bei einer elektronischen Schaltung wie einem Prozessor eines Computers erwünscht. Wir wollen uns daher jetzt mit Schaltwerken (engl. sequential circuits) beschäftigen, die man als technische Realisierung von endlichen deterministischen Automaten (finite state machines) aus der theoretischen Informatik betrachten kann.

Definition: Eine Schaltung, deren Ausgänge von der Belegung der Eingänge und ihrem inneren Zustand abhängt, wird ein **Schaltwerk** genannt.



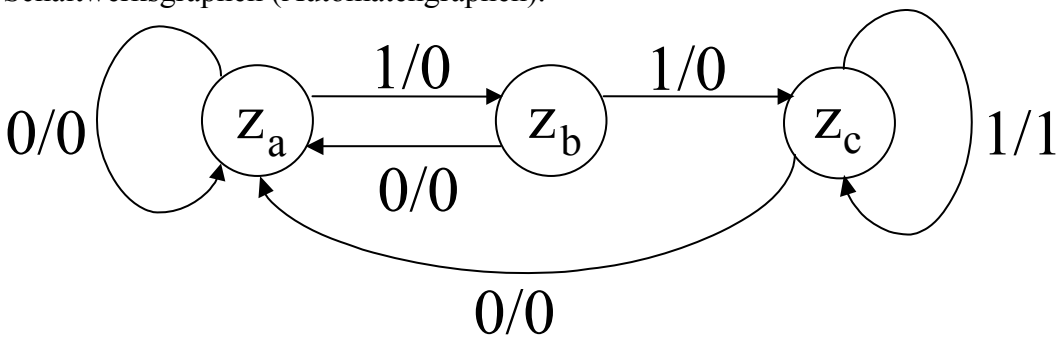

Jedes Schaltwerk enthält Speicherelemente, die den inneren Zustand speichern. Der aktuelle innere Zustand Z und die Belegung der Eingänge X bestimmen den Folgezustand Z' und die Ausgänge Y. Man kann sich die - beliebig in der Schaltung verteilten - Speicherelemente zu einem sogenannten **Zustandsregister** zusammengefasst denken.

**Definition**: Ein Speicherelement, das die beiden Werte 0 und 1 speichern (annehmen) kann, heißt **Flipflop**.

Der **Zustand** Z eines Schaltwerks ist ein m-Tupel $(z_{m-1}, z_{m-2},...,z_1, z_0)$ aus Komponenten $z_i \in B = \{ 0, 1 \}$. Jede Komponente steht dabei für ein Flipflop.

Bevor wir auf die Realisierung dieser Flipflops eingehen, wollen wir uns ein Beispiel ansehen: Zu entwerfen ist eine Schaltung, die einen Strom aus eingehenden Bits empfängt und interpretiert. Pro Takt kommt über eine Leitung x ein Bit. Wenn hintereinander drei Einsen empfangen worden sind, soll am Ausgang eine Lampe y angehen. Also, y ist 1, wenn die letzten drei empfangenen Bits 1 waren, sonst ist y 0. Ein Schaltnetz wäre mit dieser Aufgabe überfordert, da es kein Gedächtnis hat, in dem es beispielsweise den vorletzten Wert speichern kann.

Um nun unser Schaltwerk zu bauen, veranschaulichen wir uns zunächst die Funktion anhand eines Schaltwerksgraphen (Automatengraphen).




Dieser Graph ist folgendermaßen zu interpretieren: Jeder Kreis ist ein möglicher Zustand, jeder Pfeil ein möglicher Zustandsübergang. An einem Pfeil steht eine Beschriftung vom Typ Eingabe/Ausgabe, d.h. bei Eingabe des Wertes vor dem / im Zustand, in dem der Pfeil beginnt, wird der Wert nach dem / ausgegeben und in den Zustand, auf den der Pfeil zeigt, gewechselt.

In unserem Falle beginnen wir im Zustand $z_a$. Bei Eingabe einer 0 bleiben wir in diesem Zustand und wir geben eine 0 aus (y=0, die Lampe leuchtet nicht). Bei Eingabe einer 1 wechseln wir in den Zustand $z_b$ und geben eine 0 aus, denn bisher ist nur eine 1 eingegeben worden. Wenn nun eine 0 eingegeben wird, gehen wir wieder zurück in den Zustand $z_a$ und geben eine 0 aus.

Wenn wir in $z_b$ eine 1 eingeben, haben wir bereits zwei 1en hintereinander gesehen. Wir gehen in den Zustand $z_c$ und geben eine 0 aus. Nun betrachten wir $z_c$. Wenn wir in diesem Zustand sind, haben wir als letzte beide Eingaben eine 1 gehabt. Wenn jetzt noch eine 1 kommt, dann muss die Lampe leuchten, d.h. unsere Ausgabe y muss aus 1 gehen. Aber welches ist in diesem Fall der Folgezustand? Nun, wiederum $z_c$, denn auch hier waren die beiden letzten Eingaben 1en. Wenn in $z_c$ eine 0 eingegeben wird, gehen wir wieder mit Ausgabe 0 in $z_a$ zurück, denn eine Folge von drei 1en müsste ganz neu beginnen.

Der nächste Schritt ist die Codierung der Eingaben, Ausgaben und Zustände als Binärzahlen. Nun, x und y können nur zwei Werte annehmen, daher sind sie bereits Binär codiert. Für die Zustände wählen wir folgende Codierung: $z_a$ = 00, $z_b$ = 01, $z_c$ = 10. Diese beiden Bits bezeichnen wir mit $z_1$ und $z_0$.

Jetzt können wir die Wertetabelle für das erforderliche Schaltnetz aufstellen:

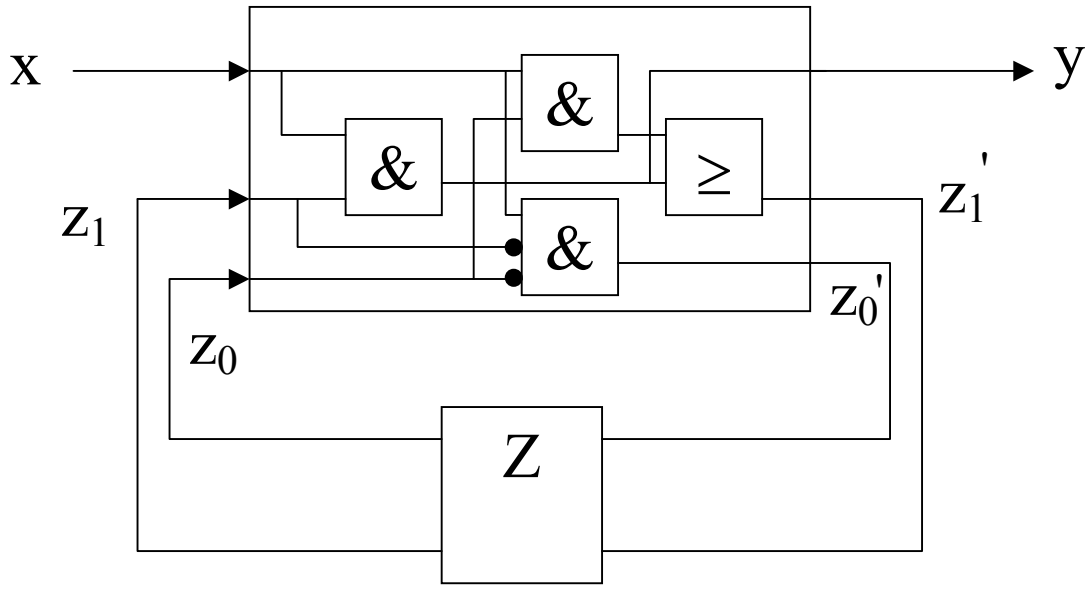
| **x** | **$z_1$** | **$z_0$** | **$z_1$'** | **$z_0$'** | **y** |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | X | X | X |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | X | X | X |

Die Anwendung von KV-Diagrammen führt uns zu folgenden disjunktiven Minimalformen für die $z_i$‘ und für y:

$z_0' = x \overline{z_0}\ \overline{z_1}$

$z_1' = x\ z_0 + x\ z_1$

$y = x\ z_1$

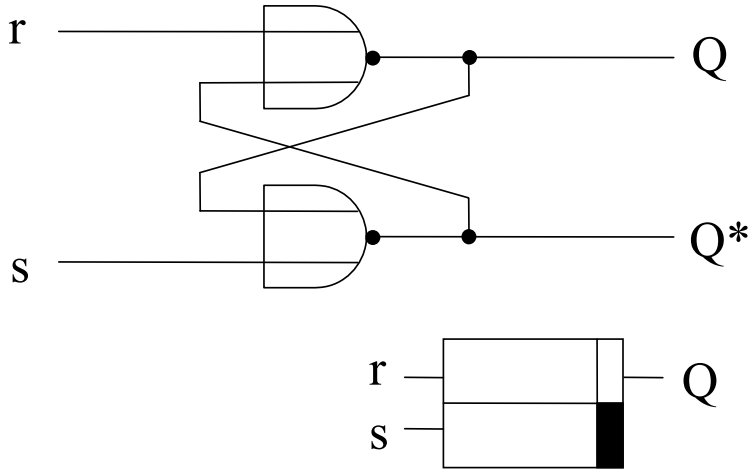



Das gesamte Schaltwerk kann also folgendermaßen realisiert werden:
Was uns nun noch fehlt, ist die Realisierung des Zustandsregisters Z. Wie sieht überhaupt ein Baustein aus, der ein Bit speichern kann, also das Flipflop?

Im folgenden werden wir einige verschiedene Flipfloptypen kennen lernen. Wir beginnen mit einfachen Flipflops, die wir dann mit zusätzlichen Funktionen ausstatten bis hin zu den sogenannten System-Flipflops, die wir schließlich für den Schaltwerksaufbau verwenden können.

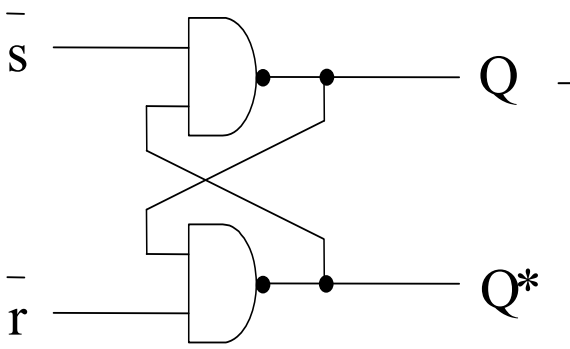
### *4.1. Das r-s-Flipflop*

Das einfachste Speicherelement ist das r-s-Flipflop. Das r steht für rücksetzen (reset), das s für setzen (set). Es kann mit zwei rückgekoppelten Nor-Gattern realisiert werden:




| $\overline{r}$ | $\overline{s}$ | Q | Q* |
|---|---|---|---|
| 0 | 0 | $Q_{alt}$ | $Q^*_{alt}$ |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |

Wenn r = 0 und s = 0 ist, speichert das Flipflop seinen alten Wert am Ausgang. Wenn r = 0 und s = 1 ist, wird es gesetzt (Q = 1); bei r = 1 und s = 0 wird es rückgesetzt (Q = 0). Die Eingabe r = 1 und s = 1 ist verboten. Am Ausgang würde sich nämlich bei dieser Beschaltung der Zustand Q = 0 und Q* = 0 einstellen. Bei einer direkt darauf folgenden Eingabe r = 0 und s = 0 würde das Flipflop in einen instabilen Zustand geraten.




| $\overline{r}$ | $\overline{s}$ | r | s | Q | Q* | |
|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | $Q_{alt}$ | $Q^*_{alt}$ | |
| 1 | 0 | 0 | 1 | 1 | 0 | |
| 0 | 1 | 1 | 0 | 0 | 1 | |
| 0 | 0 | 1 | 1 | 1 | 1 | ← verboten |

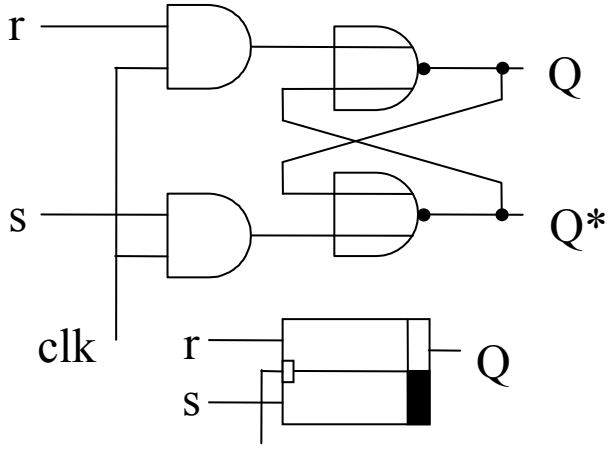
Das r-s-Flipflop kann auch mit NAND-Gattern aufgebaut werden. Bis auf die verbotene Eingabe ist die Wertetabelle identisch mit der des NOR-Flipflops (weil r und s invertiert eingehen).

Ein solches Basisflipflop übernimmt beim Setzen oder Rücksetzen den Wert sofort an den Ausgang. Häufig ist diese Funktion aber nicht erwünscht, sondern man möchte den Zustand des Flipflops nur zu definierten Zeiten ändern. Zu diesem Zweck macht man das Übernehmen des Wertes abhängig von einem weiteren Eingang, z.B. einem Takteingang: Solange der Takt

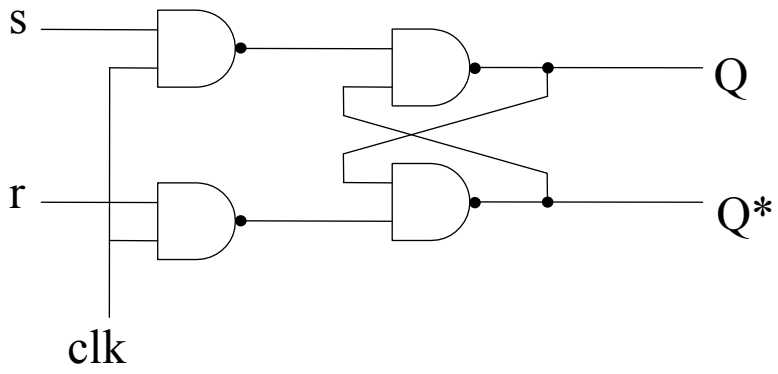
0 ist, soll das Flipflop seinen alten Wert speichern. Wenn der Takt jedoch 1 ist, soll es gesetzt oder rückgesetzt werden können, es soll aber auch bei r=s=0 seinen alten Wert erhalten können. r=s=1 ist wiederum verboten.

Wertetabelle:

| clk | r | s | Q | Q* |
|---|---|---|---|---|
| 0 | 0 | 0 | $Q_{alt}$ | $Q^*_{alt}$ |
| 0 | 0 | 1 | $Q_{alt}$ | $Q^*_{alt}$ |
| 0 | 1 | 0 | $Q_{alt}$ | $Q^*_{alt}$ |
| 0 | 1 | 1 | $Q_{alt}$ | $Q^*_{alt}$ |
| 1 | 0 | 0 | $Q_{alt}$ | $Q^*_{alt}$ |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | verboten | |



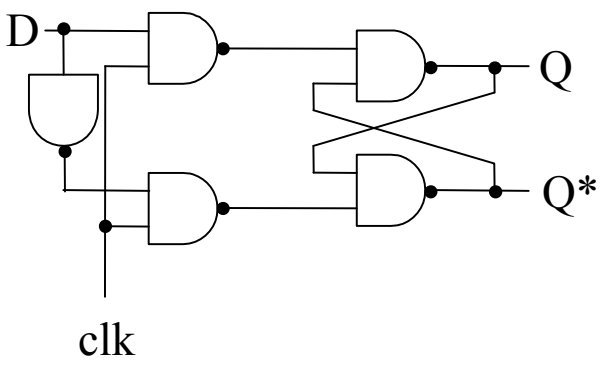

Dieses getaktete r-s-Flipflop nennt man auch ein r-s-Auffangflipflop (r-s-Latch). Es kann nämlich während der positiven Taktphase (also während clk=1 ist) einen Wert aufnehmen, der dann während der negativen Taktphase (clk=0) gespeichert bleibt. Man kann ein Latch natürlich wieder in NAND-Logik aufbauen.




Nun sind wir häufig lediglich daran interessiert, einen Eingabewert, also ein Bit, unverändert zu übernehmen und zu speichern. Dann brauchen wir nicht die Funktionalität eines r-s-Flipflops, das ja immer den Nachteil der verbotenen Eingabekombination hat, sondern ein D-Flipflop. D steht für delay. Wir sehen hier die Wertetabelle und eine mögliche Realisierung mit einem NAND-Basisflipflop.

Wertetabelle:

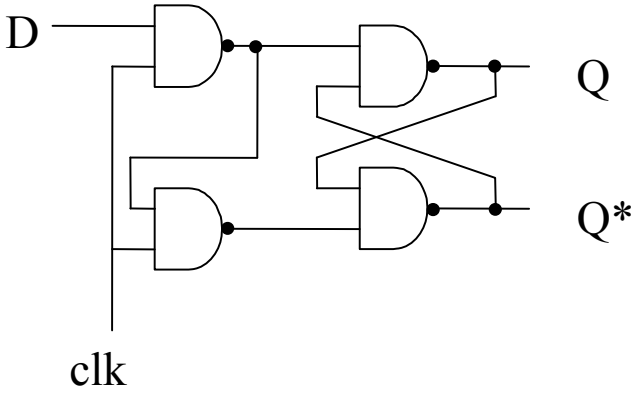
| clk | D | Q | Q* |
|---|---|---|---|
| 0 | 0 | $Q_{alt}$ | $Q^*_{alt}$ |
| 0 | 1 | $Q_{alt}$ | $Q^*_{alt}$ |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |




Man kann den Inverter einsparen, indem man das Layout geringfügig verändert. In dieser Version muss aber der Taktimpuls mindestens so lang sein wie eine Schaltzeit eines Nand-Gatters, da sonst bei clk = 1 das invertierte D sich nicht mehr auf dem unteren NAND-Gatter auswirken kann.

Wertetabelle:

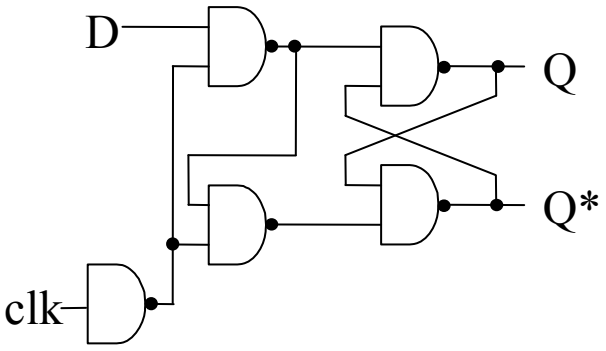
| clk | D | Q | Q* |
|---|---|---|---|
| 0 | 0 | $Q_{alt}$ | $Q^*_{alt}$ |
| 0 | 1 | $Q_{alt}$ | $Q^*_{alt}$ |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |




Alle bisher betrachteten Flipflops übernahmen die Eingangswerte, während der Takt auf 1 war und speicherten, wenn der Takt auf 0 war. Man nennt dies ein **positiv levelgesteuertes Auffangflipflop**. Man kann aber jetzt natürlich durch Inversion des Taktsignals ein negativ levelgesteuertes Auffangflipflop bauen, das bei clk = 0 übernimmt und bei clk = 1 speichert:

Wertetabelle:

| clk | D | Q | Q* |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | $Q_{alt}$ | $Q^*_{alt}$ |
| 1 | 1 | $Q_{alt}$ | $Q^*_{alt}$ |



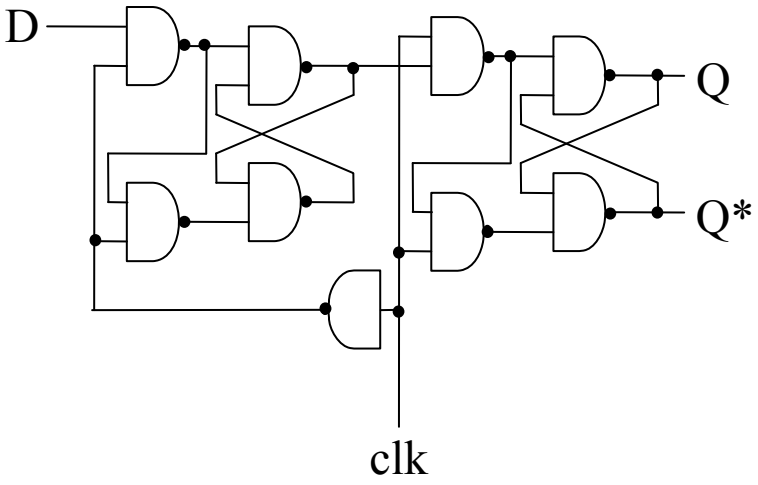

Leider genügen Auffangflipflops noch nicht den Anforderungen, die wir an die Speicherelemente unserer Schaltwerke stellen müssen. Was würde passieren? Sobald der Takt auf 1 ist, wird der Eingang der Flipflops als neuer Zustand an den Ausgang übernommen. Dieser neue Zustand liegt aber jetzt am Schaltnetz an, das nach einigen Gatterschaltzeiten neue Ausgänge produziert. Einige von diesen Ausgängen liegen als Folgezustand an den Eingängen der Flipflops. Da aber der Takt immer noch 1 sein kann, wirken diese sich nun wieder auf die Ausgänge der Flipflops aus und der Zustand wird wieder überschrieben. Mit

anderen Worten: Die Signale würden einige Male in dieser Rückkopplungsschleife herumlaufen, bis irgendwann das Taktsignal auf 0 geht, und das Flipflop schließlich speichert.

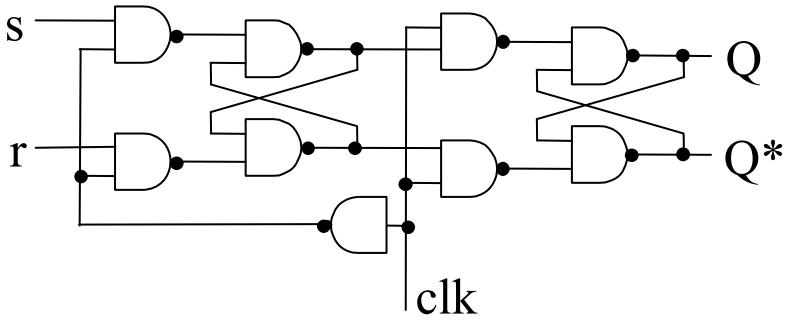
Was wir also stattdessen benötigen, ist ein Flipflop, das seinen Ausgang stabil lässt, solange die neuen Eingänge übernommen werden und dann, zu einem späteren Zeitpunkt, seine Eingänge nicht mehr übernimmt und stattdessen den übernommenen Wert an den Ausgang weiterschaltet. Ein Flipflop, das dieses leistet, nennt man ein **System-Flipflop**.

Ein System Flipflop kann man als sogenanntes **Master-Slave-Flipflop** aus zwei Auffangflipflops zusammensetzen, und zwar einem negativ levelgesteuerten und einem positiv levelgesteuerten Latch.

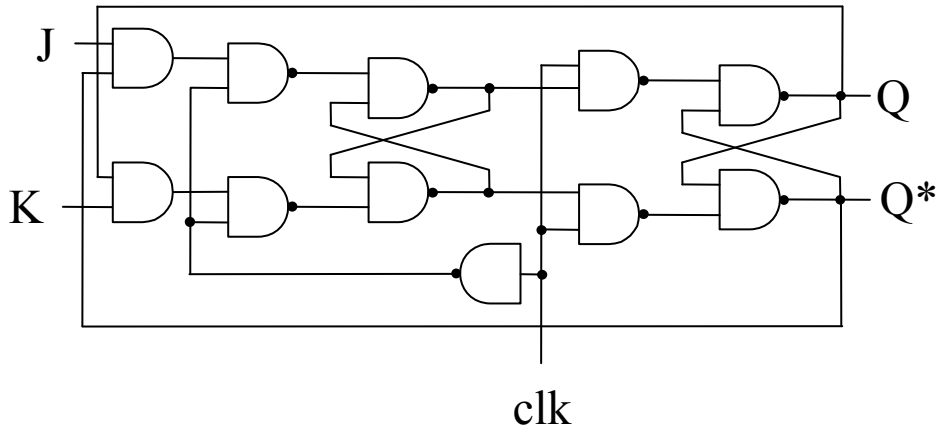
### *4.2. Master-Slave D-Flipflop*

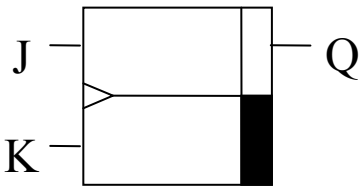

Entsprechend dem D-Flipflop kann man natürlich auch r-s-Flipflops als Master-Slave-Flipflops aufbauen. Allerdings bleibt das Problem mit der verbotenen Eingabe.



Um dieses Problem zu umgehen, kann man ein sogenanntes J-K-Flipflop bauen. Bei diesem sind die Ausgänge an die Eingänge rückgekoppelt.

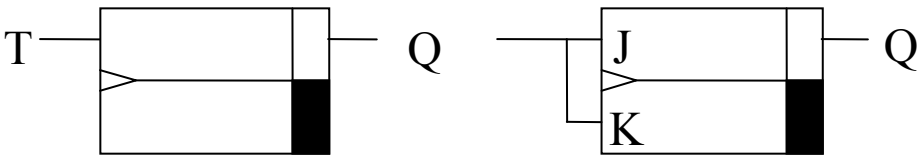



| J | K | Q |
|---|---|---|
| 0 | 0 | $Q_{alt}$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $\overline{Q_{alt}}$ |

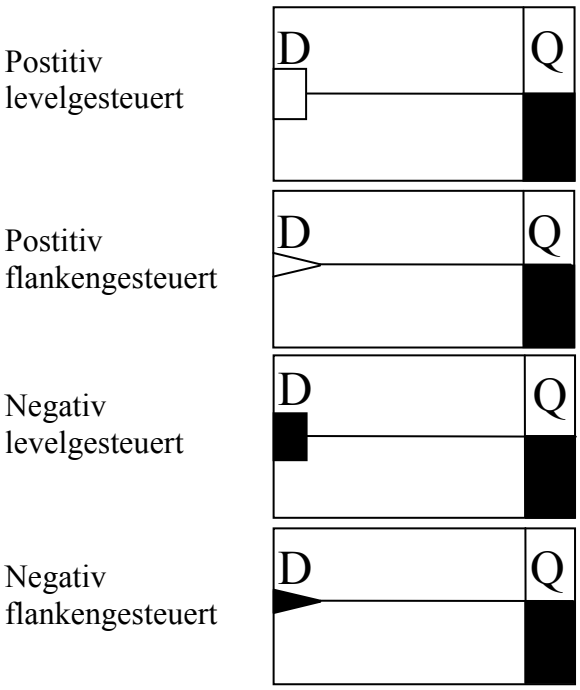



Gelegentlich benötigt man nur die Funktionalität der ersten und letzten Spalte der Wertetabelle eines J-K-Flipflops. In diesem Fall kann man ein Wechselflopflop (W-Flipflop oder auch T-Flipflop) verwenden, das nichts anderes ist als ein J-K-Flipflop mit verbundenen Eingängen:

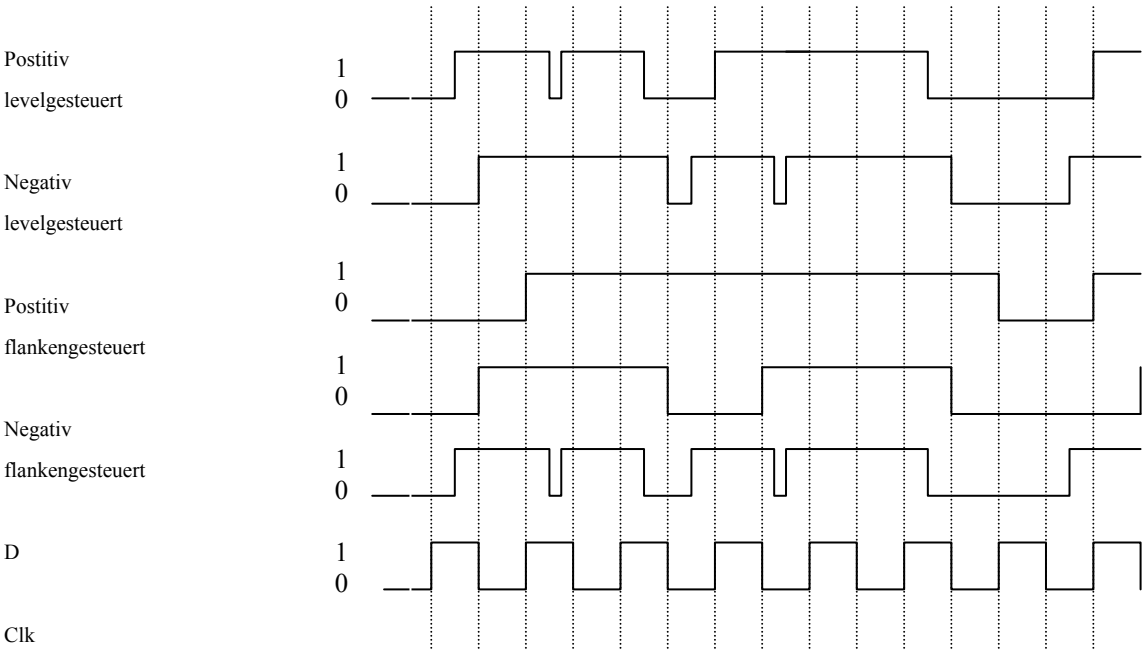
| T | Q |
|---|---|
| 0 | $Q_{alt}$ |
| 1 | $\overline{Q_{alt}}$ |




Alle diese Systemflipflops übernehmen die Werte des Eingangs während der negativen Taktphase und zeigen sie am Ausgang zu Beginn der positiven Taktphase. Sie wechseln also ihren Zustand mit der **positiven Flanke** des Taktes. Dies ist für Systemflipflops nicht zwingend erforderlich: Wenn man ein Master-Slave-Flipflop so aufbaut, dass der Master während der positiven Phase übernimmt und der Slave während der negativen, so hat man ein Systemflipflop, das bei der negativen Flanke den Zustand ändert. Man spricht auch von positiv oder negativ flankengesteuerten (oder auch flankengetriggerten) Flipflops. Entsprechend wurden Latches in positiv oder negativ pegelgesteuerte (oder auch levelgesteuerte) Flipflops eingeteilt:




Hier sehen wir beispielhaft die Verläufe der Zustandsänderungen dieser vier Flipfloptypen für ein gemeinsames Eingangssignal D:



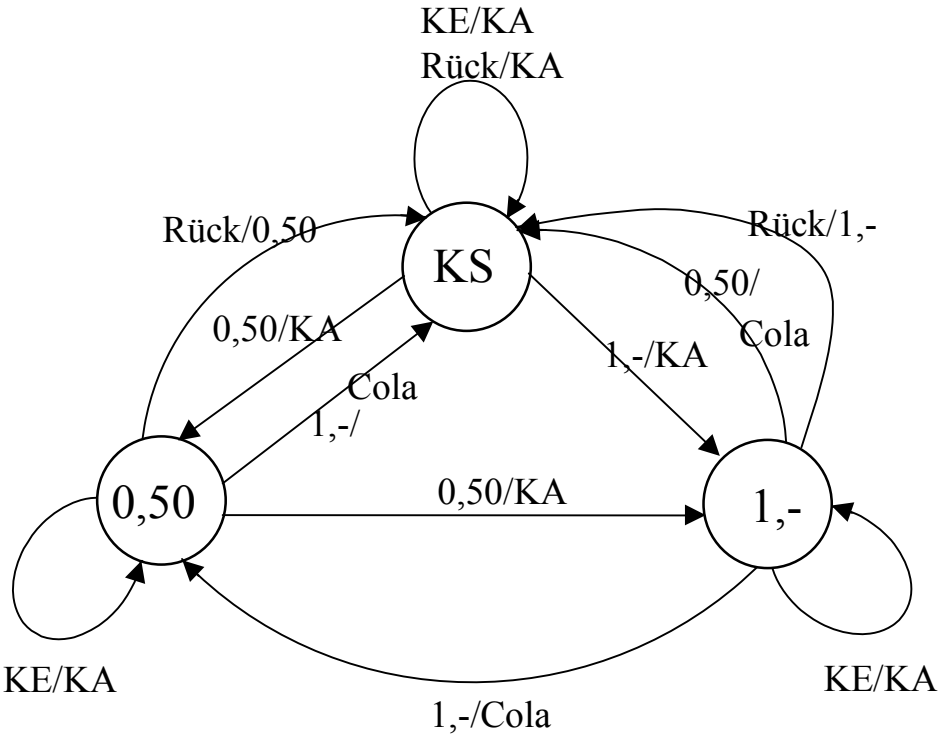

**Beispiel:** Cola-Automat

Ein Automat soll formal beschrieben und realisiert werden, an dem man für 1,50 eine Cola kaufen kann. Der Automat akzeptiert 50 Cent- und 1 Euro-Stücke. Sobald ein Betrag von 1,50 oder mehr eingegeben wurde, wird eine Cola ausgegeben. Durch Betätigung des Rückgabeknopfes kann der bereits eingeworfene Guthabenbetrag wieder ausgegeben werden. Da wir von einem getakteten System ausgehen, muss es eine "leere Eingabe" KE (keine Eingabe) und eine "leere Ausgabe" KA (keine Ausgabe) geben.

$A = (X, Y, Z, \delta, \lambda)$ $X = \{ 0.50, 1.-, KE, Rück \}$

Y = { Cola, 0.50, 1.-, KA }

Z = { KS, 0.50, 1.- }




| δ | 0.50 | 1.- | KE | Rück |
|---|---|---|---|---|
| KS | 0.50 | 1.- | KS | KS |
| 0.50 | 1.- | KS | 0.50 | KS |
| 1.- | KS | 0.50 | 1.- | KS |

| λ | 0.50 | 1.- | KE | Rück |
|---|---|---|---|---|
| KS | KA | KA | KA | KA |
| 0.50 | KA | Cola | KA | 0.50 |
| 1.- | Cola | Cola | KA | 1.- |

Codierung:

| X | KE | 0.50 | 1.- | Rück |
|---|---|---|---|---|

| $x_1x_0$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|

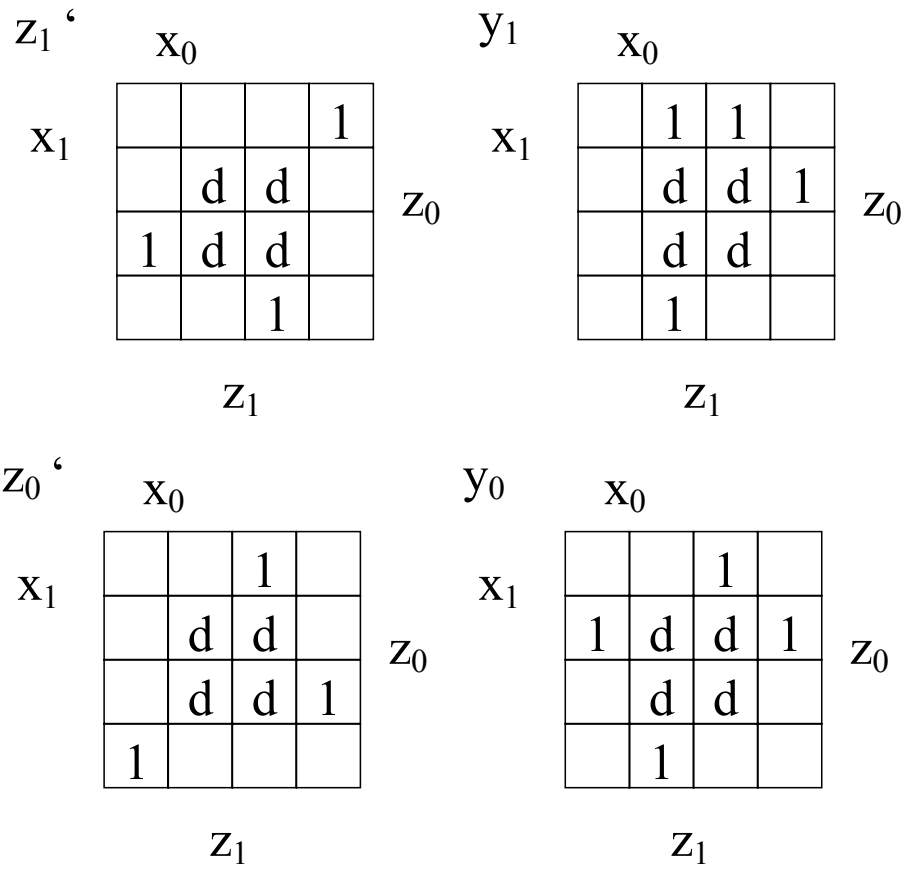
| Y | KA | 0.50 | 1.- | Cola |
|---|---|---|---|---|
| $y_1y_0$ | 00 | 01 | 10 | 11 |

| Z | KS | 0.50 | 1.- |
|---|---|---|---|
| $z_1z_0$ | 00 | 01 | 10 |

Wertetabelle:

| $x_1$ | $x_0$ | $z_1$ | $z_0$ | $z_1'$ | $z_0'$ | $y_1$ | $y_0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | X | X | X | X |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | X | X | X | X |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | X | X | X | X |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | X | X | X | X |

### *4.3. KV-Diagramme:*



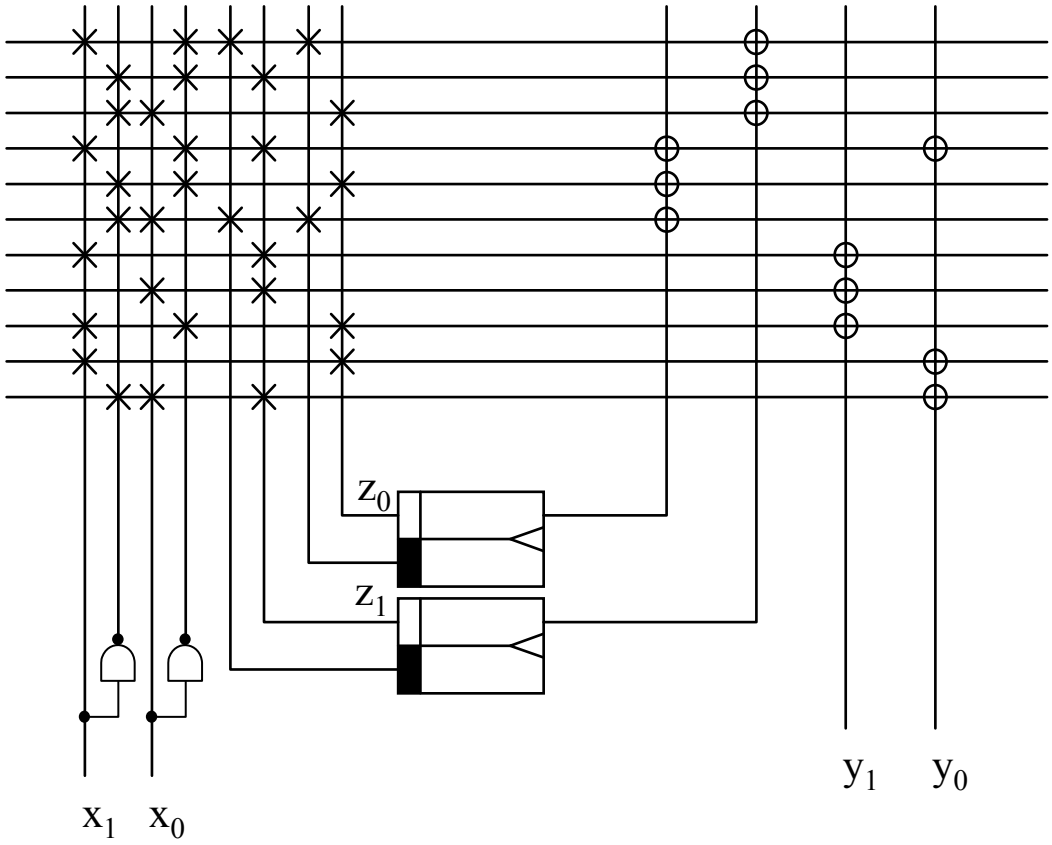

$$z_1' = \overline{x_0} x_1 \overline{z_0}\,\overline{z_1} + \overline{x_0}\,\overline{x_1} z_1 + x_0 \overline{x_1} z_0$$

$$z_0' = \overline{x_0} x_1 z_1 + \overline{x_0}\,\overline{x_1} z_0 + x_0 \overline{x_1}\,\overline{z_0}\,\overline{z_1}$$

$$y_1 = x_1 z_1 + x_0 z_1 + \overline{x_0} x_1 z_0$$

$$y_0 = x_1 z_0 + \overline{x_0} x_1 z_1 + x_0 \overline{x_1} z_1$$

Realisierung als FPLA:



# 5. Spezielle Schaltwerke

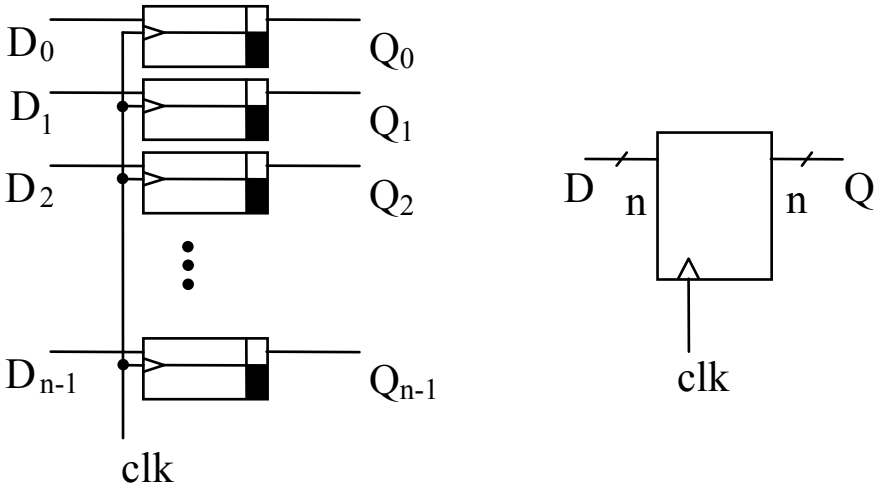
In diesem Abschnitt werden wir einige Schaltwerke kennen lernen, die als Basisbauteile überall im Aufbau digitaler Schaltungen verwendet werden.

## *5.1. Das Register*

Das Register oder der Wortspeicher ist eine Parallelschaltung von n Flipflops. In einem Register kann ein binäres Wort der Länge n gespeichert werden. Die Flipflops können Latches (Auffangflipflops) oder System-Flipflops (z.B. D-Flipflops) sein.

Die folgende Folie zeigt den Aufbau eines Registers und sein Schaltbild.

Register:



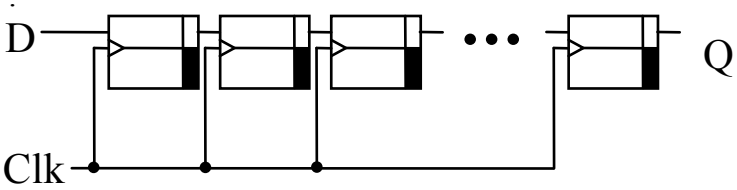

### 5.1.1. Das Schieberegister

Das Schieberegister ist ein Wortspeicher mit bit-seriellem Zugriff. Es besteht aus einer Serienschaltung von n D-Flipflops. In einem Schieberegister kann ein binäres Wort der Länge n dadurch gespeichert werden, dass pro Takt ein Bit eingegeben wird. Entsprechend wird das gespeicherte Wort in n aufeinanderfolgenden Takten am Ende des Schieberegisters ausgegeben. Die Flipflops müssen System-Flipflops sein.

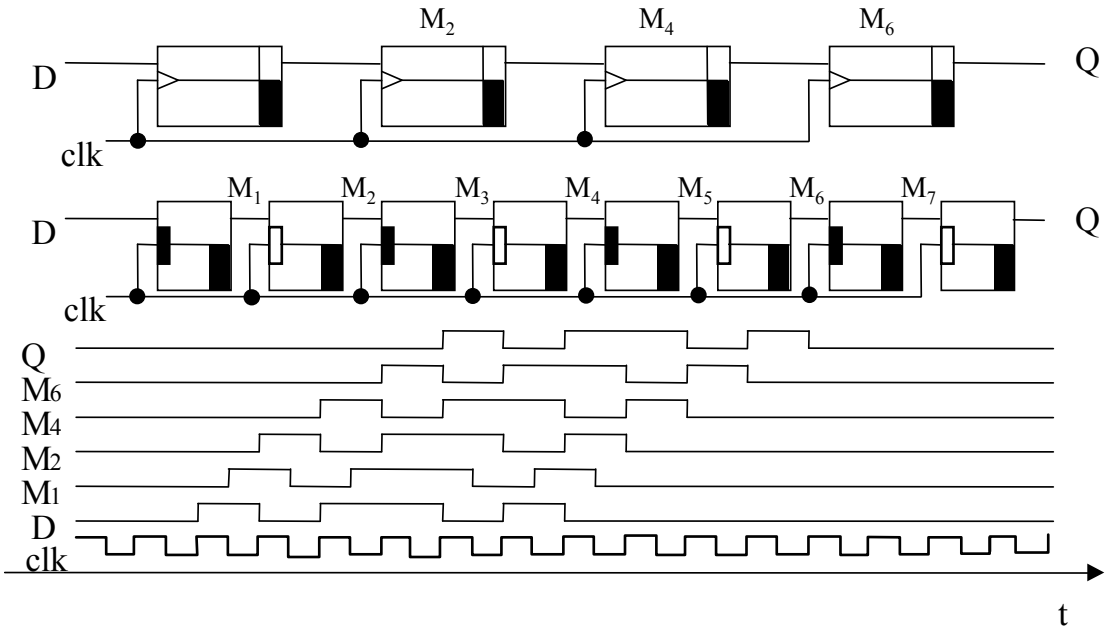
Die folgende Folie zeigt den Aufbau eines Schieberegisters.

Wir wissen bereits, dass man ein System-Flipflop als Master-Slave-Flipflop aus zwei Latches aufbauen kann. Diesen Aufbau und den Verlauf eines Signals 001011010000 entlang des Schieberegisters sieht man auf der übernächsten Folie.

Schieberegister:



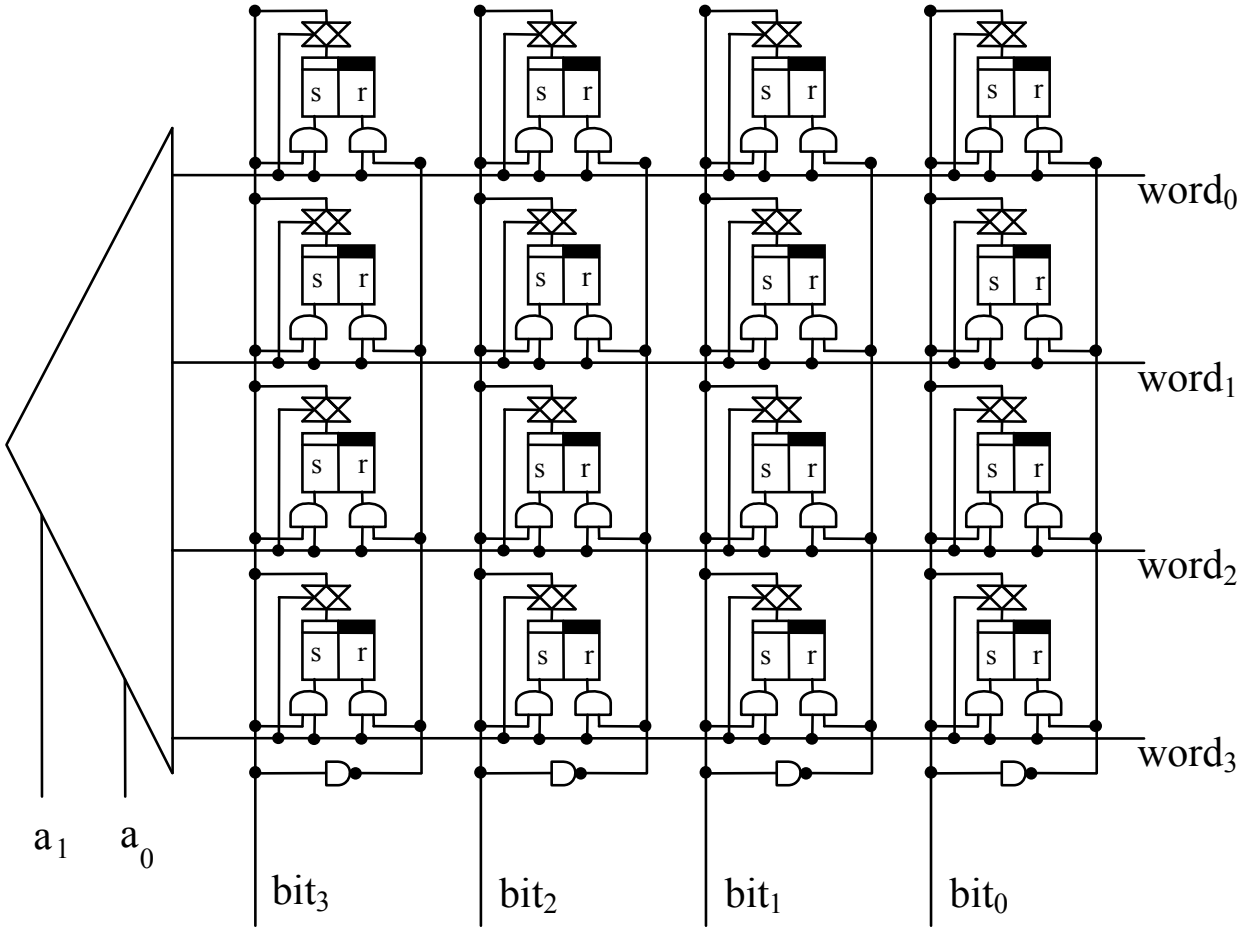

4-Bit-Schieberegister:




### *5.2. Das RAM*

Das Random Access Memory ist ein Speicher für N Worte der Breite m Bit. Jedes dieser Worte ist durch eine Adresse identifizierbar. Die Adresse hat n = log N Bits. Wenn eine Adresse $a_{n-1}a_{n-1}...a_1a_0$ angelegt wird, kann auf das zugehörige Wort lesend oder schreibend zugegriffen werden. Zu diesem Zweck legt man die Adresse an einem Dekodierer an, der sie in einen 1-aus-N-Code dekodiert. Für jedes der N Worte gibt es nun eine sogenannte word-Leitung. Durch den Dekodiervorgang wird auf die gesuchte word-Leitung eine 1 gelegt, auf alle anderen eine 0. An jeder dieser word-Leitungen liegt nun ein Register der Breite m Bit. Durch das Aktivieren der word-Leitung kann dieses Register nun von außen gelesen oder beschrieben werden.

Ein einfacher Aufbau eines RAM mit N = 4 und m = 4 ist auf der nächsten Folie dargestellt. Die Speicherbausteine sind r-s-Flipflops. Jeweils vier davon sind zu einem parallelen Wortspeicher zusammengeschaltet.




### 5.2.1. Die Funktionsweise des RAM

Beim Schreiben wird an die Bit-Leitungen ein m-Bit-Wort angelegt. Dieses soll in die Zeile i geschrieben werden. Die Adresse i liegt am Decoder an. Dadurch wird die i-te word-Leitung auf 1 gelegt. Die And-Gatter vor den Flipflops der i-ten Zeile lassen somit am s-Eingang jedes r-s-Flipflops den Bit-Wert und am r-Eingang den invertierten Bit-Wert durch. Alle anderen word-Leitungen sind auf 0, d.h. die And-Gatter legen an alle anderen r-s-Flipflops die Speicherkombination r=0 und s=0 an. Somit wird in Zeile i das neue Wort „eingespeichert“. Alle anderen Zeilen speichern die alten Werte.

Beim Lesen wird an die Bit-Leitungen von außen nichts angelegt, sie sind im hochohmigen Zustand Z. Durch die word-Leitung werden nun die Transmission-Gates der Ausgang der Flipflops in der i-ten Zeile geöffnet, die gespeicherten Werte gelangen auf die Bit-Leitungen. Somit werden die Werte der i-ten Zeile an den Ausgang des RAM transportiert.

In der Realität werden RAMs nicht aus r-s-Flipflops und And-Gattern und Transmission-Gates aufgebaut, da diese Realisierung zu aufwendig wäre. Man unterscheidet zunächst zwischen SRAM (statischem RAM) und DRAM (dynamischem RAM). Das statische RAM speichert einen Wert und hält diesen, solange die Versorgungsspannung eingeschaltet bleibt. Unser RAM aus r-s-Flipflops ist ein Beispiel für eine Realisierung eines statischen RAM.

Das dynamische RAM speichert alle Werte nur für eine kurze Zeit. Und zwar wird als Speichermedium die Kapazität eines Transistor-Gates (polychristallines Silizium) gegenüber der Source/Drain (Diffusion) ausgenutzt. Natürlich ist ein solcher Speicher flüchtig. Daher muss jedes Bit in einem dynamischen RAM von Zeit zu Zeit aufgefrischt (refresh) werden. Das geschieht, indem automatisch in festen Zeitabständen zeilenweise alle Bits im RAM

einmal gelesen und unverändert wieder geschrieben werden. Ein typisches Zeitintervall für den Refresh-Zyklus bei heutigen DRAMs ist eine Millisekunde.

SRAM ist schneller (kürzere Zugriffszeit) und teurer. Außerdem benötigt SRAM 6 Transistoren pro gespeichertem Bit, während DRAM mit einem Transistor pro Bit auskommt. Daher hat DRAM eine höhere Speicherkapazität pro Chipfläche.

Heutige Speicherchips sind quadratisch angeordnet. Es werden zwei Dokodierer verwendet, einen für die Zeile und einen für die Spalte. Zeilen- und Spaltenadresse sind dabei gleich lang. Auf diese Weise kann man mit der Hälfte der Adresspins auskommen, indem man die Adressleitungen im Zeitmultiplex verwendet. Immer wird zuerst die Zeilenadresse übertragen und dann über dieselben Pins die Spaltenadresse.
Um den Vorteil aus beiden Speicherarten (SRAM und DRAM) zu ziehen, wird heute zunehmend SDRAM verwendet. Dies ist eine Kombination aus beiden Techniken. Es handelt sich im Prinzip um DRAM, bei dem aber die Zeile, aus der zuletzt gelesen wurde in einem kleinen separaten SRAM gehalten wird. Da beim Zugriff auf den Speicher häufig mehrmals hintereinander auf dieselbe Zeile zugegriffen wird, kann jeder Zugriff mit der Geschwindigkeit des SRAM bedient werden. Trotzdem kann die hohe Speicherdichte des DRAM ausgenutzt werden.

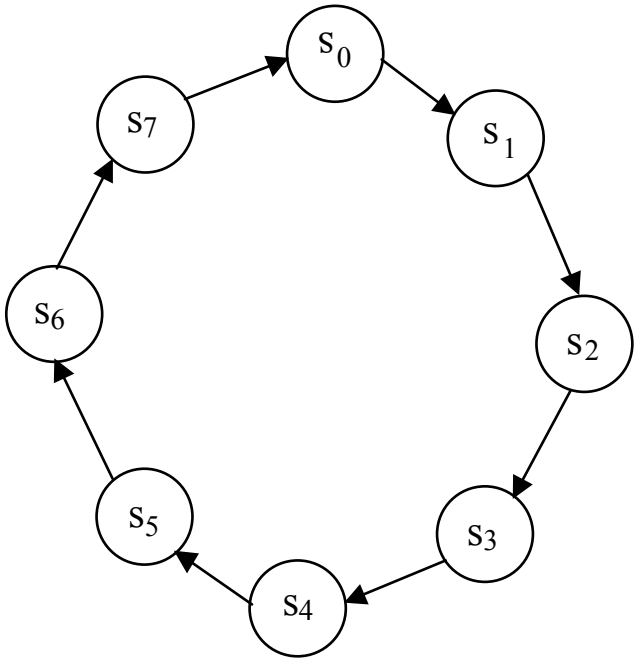
## *5.3. Zähler*

Zähler sind spezielle Schaltwerke, die in der Regel wenige (u.U. gar keine) Eingänge haben. Oft sind auch Ihre Ausgaben identisch mit den Zuständen. Der Zustand ist der Stand des Zählers. Häufig ist der Zählerstand eine binär codierte Zahl. Der Takt bewirkt, dass der Zähler zählt, also seinen Zustand wechselt. Wir zählen also quasi die Anzahl der ansteigenden Taktflanken.

Aus der Sicht der Automatentheorie haben wir bisher alle Schaltwerke als Mealy-Automaten gebaut. Wenn die Ausgabe aber nun nicht von der aktuellen Eingabe sondern nur vom aktuellen Zustand abhängt, so handelt es sich um einen Moore-Automaten. Wenn die Ausgabe identisch mit dem aktuellen Zustand ist, so sprechen wir von einem Medwedew-Automaten.

Wir werden in diesem Kapitel eine Reihe unterschiedlicher Zähler kennen lernen. Als einleitendes Beispiel wählen wir einen synchronen Modulo-8-Zähler. Auf der folgenden Folie ist der Automatengraph für diesen dargestellt: Der Modulo-8-Zähler wandert zyklisch durch 8 Zustände, wobei er bei jedem Takt einen Schritt macht.

Modulo-8-Zähler:

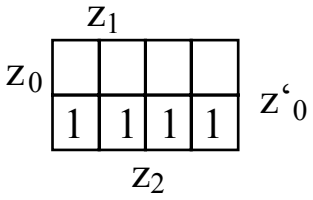


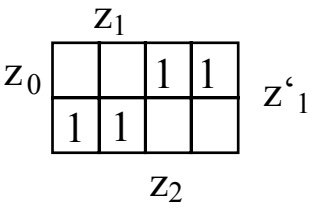

Codierung der Zustände:

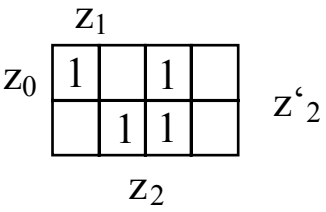
| | $z_2$ | $z_1$ | $z_0$ |
|---|---|---|---|
| $s_0$: | 0 | 0 | 0 |
| $s_1$: | 0 | 0 | 1 |
| $s_2$: | 0 | 1 | 0 |
| $s_3$: | 0 | 1 | 1 |
| $s_4$: | 1 | 0 | 0 |
| $s_5$: | 1 | 0 | 1 |
| $s_6$: | 1 | 1 | 0 |
| $s_7$: | 1 | 1 | 1 |

Wertetabelle:

| $z_2$ | $z_1$ | $z_0$ | $z'_2$ | $z'_1$ | $z'_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 |



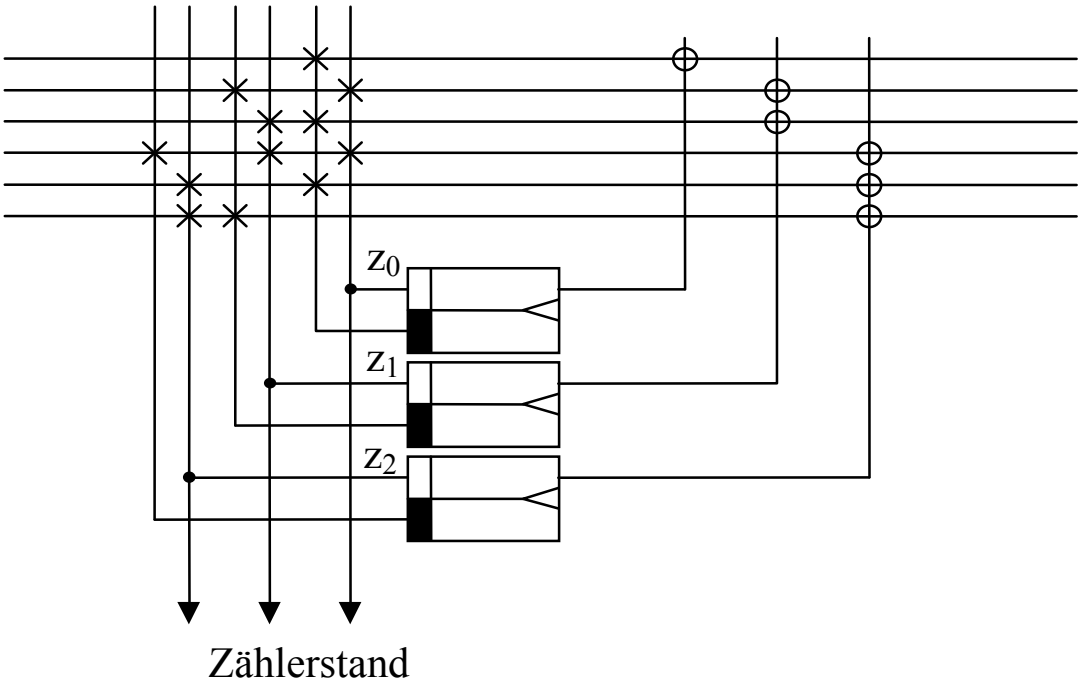

Ergebnis:

$$z'_0 = z_0$$

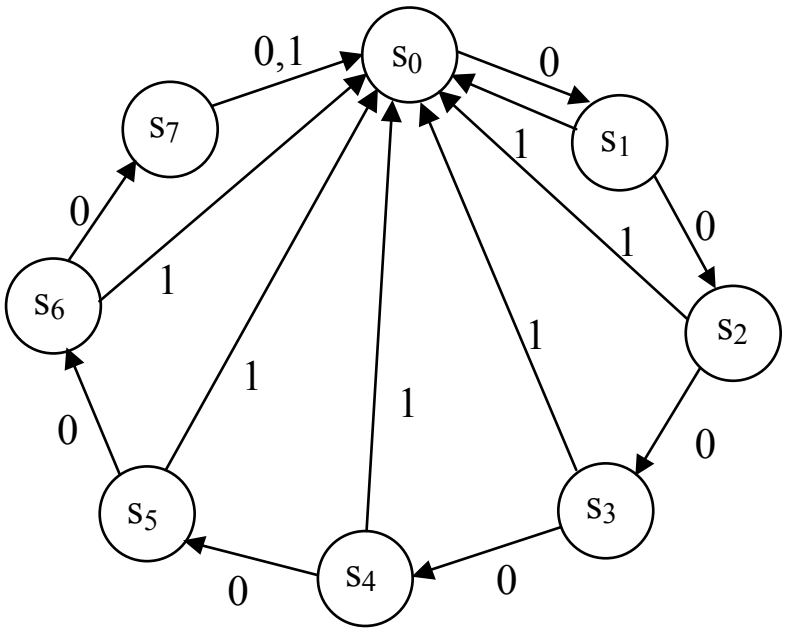
$$z'_1 = z_0 z_1 + \bar{z}_0 z_1$$

$$z'_2 = z_0 z_1 z_2 + \bar{z}_0 z_2 + \bar{z}_1 z_2$$

Realisierung als FPLA:




Unser Modulo-8-Zähler hat noch einen kleinen Schönheitsfehler: Wir können ihn noch nicht auf einen definierten Anfangszustand setzen. Wenn wir das zusätzlich wollen, braucht das Schaltwerk doch einen Eingang, z.B. ein Signal „reset“. Wenn dieses 1 ist, soll der Zähler in den Zustand $s_0$ gehen. Der entsprechend veränderte Zustandsgraph sieht dann so aus:



Wertetabelle:

| **reset** | $z_2$ | $z_1$ | $z_0$ | $z'_2$ | $z'_1$ | $z'_0$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | X | X | X | 0 | 0 | 0 |

Der Leser möge sich die Realisierung selber überlegen.

### 5.3.1. Andere Zähler

Während D-Flipflops in der Regel die geeigneten Bausteine zum Speichern von Zuständen in Schaltwerken sind, werden insbesondere bei Zählern häufig auch andere Flipfloptypen eingesetzt, weil sich dadurch Schaltungsaufwand in den Schaltnetzen einsparen lässt. Das folgende Beispiel ist ein Modulo-8-Zähler, der mit r-s-Flipflops aufgebaut werden soll. In der Wertetabelle können an vielen Stellen „dont cares“ eingesetzt werden, nämlich immer dann, wenn der erwünschte Wert durch „setzen“ oder „speichern“ erreicht werden kann:

| $z_2$ | $z_1$ | $z_0$ | $s_2$ | $r_2$ | $s_1$ | $r_1$ | $s_0$ | $r_0$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | X | 0 | X | 1 | 0 |
| 0 | 0 | 1 | 0 | X | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | X | X | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | X | 0 | 0 | X | 1 | 0 |
| 1 | 0 | 1 | X | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | X | 0 | X | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

| $z_0$ \ $z_1$ | | | | |
|---|---|---|---|---|
| | 1 | | X | |
| | | X | X | |

$z_2$

$\overline{s_2} = z_0 z_1 z_2$

| $z_0$ \ $z_1$ | | | | |
|---|---|---|---|---|
| | | 1 | | X |
| | X | | | X |

$z_2$

$r_2 = z_0 z_1 z_2$

| $z_0$ \ $z_1$ | | | | |
|---|---|---|---|---|
| | | | 1 | 1 |
| | X | X | | |

$z_2$

$\overline{s_1} = z_0 \overline{z_1}$

| $z_0$ \ $z_1$ | | | | |
|---|---|---|---|---|
| | 1 | 1 | | |
| | | | X | X |

$z_2$

$r_1 = z_0 z_1$

| $z_0$ \ $z_1$ | | | | |
|---|---|---|---|---|
| | | | | |
| | 1 | 1 | 1 | 1 |

$z_2$

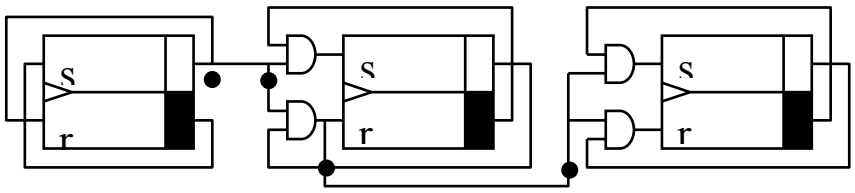
$\overline{s_0} = \overline{z_0}$

| $z_0$ \ $z_1$ | | | | |
|---|---|---|---|---|
| | 1 | 1 | 1 | 1 |
| | | | | |

$z_2$

$r_0 = z_0$

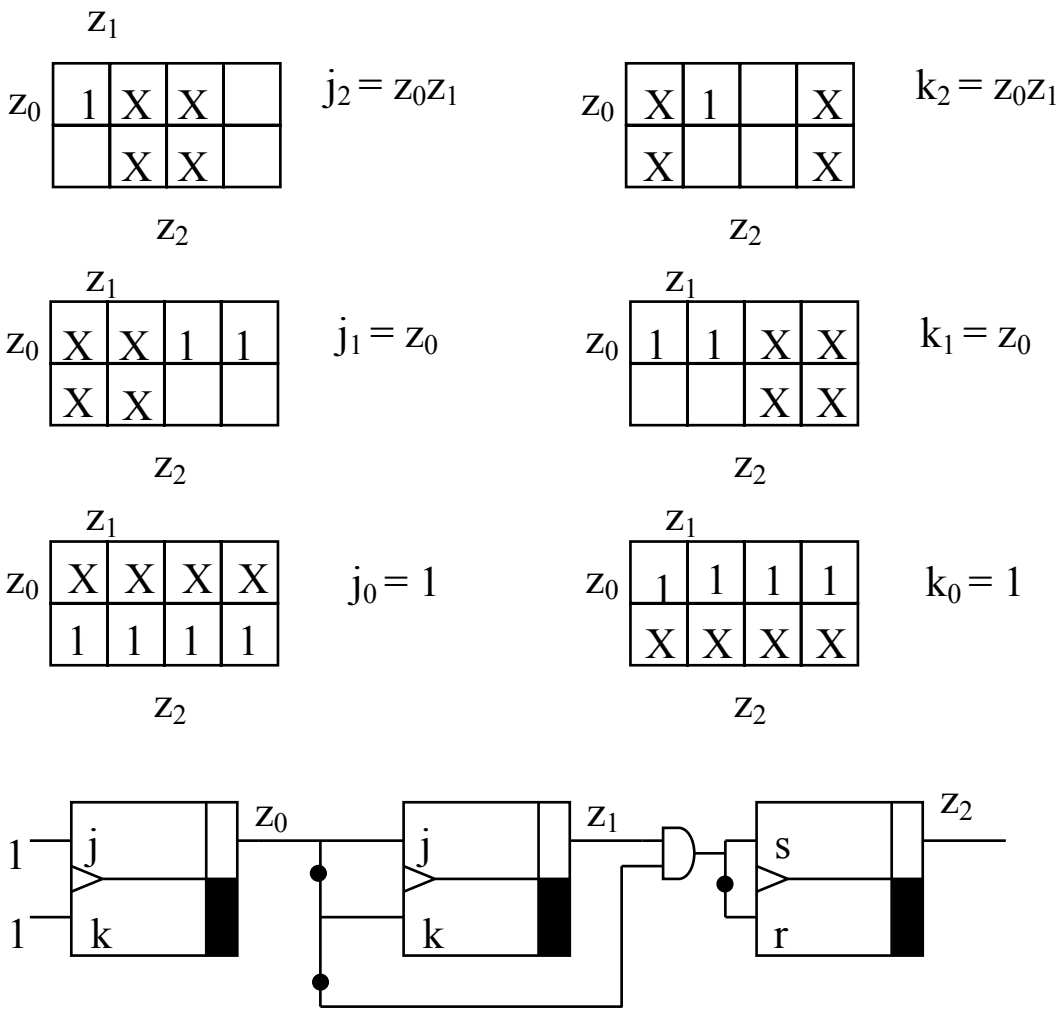
Realisierung:




### 5.3.2. Einsatz von J-K-Flipflops zum Bau von Zählern

Wir können weiteren Schaltungsaufwand einsparen, wenn wir statt r-s-Flipflops J-K-Flipflops verwenden. Hier kommt uns zu Hilfe, dass die Eingabe j=k=1 zum Toggeln des Zustands genutzt werden kann.

| $z_2$ | $z_1$ | $z_0$ | $j_2$ | $k_2$ | $j_1$ | $k_1$ | $j_0$ | $k_0$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | X | 0 | X | 1 | X |
| 0 | 0 | 1 | 0 | X | 1 | X | X | 1 |
| 0 | 1 | 0 | 0 | X | X | 0 | 1 | X |
| 0 | 1 | 1 | 1 | X | X | 1 | X | 1 |
| 1 | 0 | 0 | X | 0 | 0 | X | 1 | X |
| 1 | 0 | 1 | X | 0 | 1 | X | X | 1 |
| 1 | 1 | 0 | X | 0 | X | 0 | 1 | X |
| 1 | 1 | 1 | X | 1 | X | 1 | X | 1 |




### 5.3.3. Aufbau eines Modulo-6-vorwärts/rückwärts Zahlers mit T-Flipflops

Dieser Zähler soll über ein Eingabesignal r (für rückwärts) so gesteuert werden, dass er aufwärts (r=0) oder abwärts (r=1) zählen kann. Zu seiner Realisierung sollen T-Flipflops verwendet werden, also solche, die bei Eingabe einer 0 den alten Zustand speichern und bei Eingabe einer 1 den Zustand wechseln (toggeln).

| r | $z_2$ | $z_1$ | $z_0$ | $T_2$ | $T_1$ | $T_0$ |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | X | X | X |
| 0 | 1 | 1 | 1 | X | X | X |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | X | X | X |
| 1 | 1 | 1 | 1 | X | X | X |

| r \ $z_0$ | | | 1 | 1 |
|---|---|---|---|---|
| | | X | X | |
| 1 | | X | X | |
| | | 1 | | |

$z_1$

$z_2$

$$T_2 = \bar{z}_0 \bar{z}_1 r + z_0 z_1 \bar{r} + z_0 z_2 \bar{r}$$

| r \ $z_0$ | | | 1 | |
|---|---|---|---|---|
| | | X | X | 1 |
| 1 | | X | X | |
| 1 | | | | |

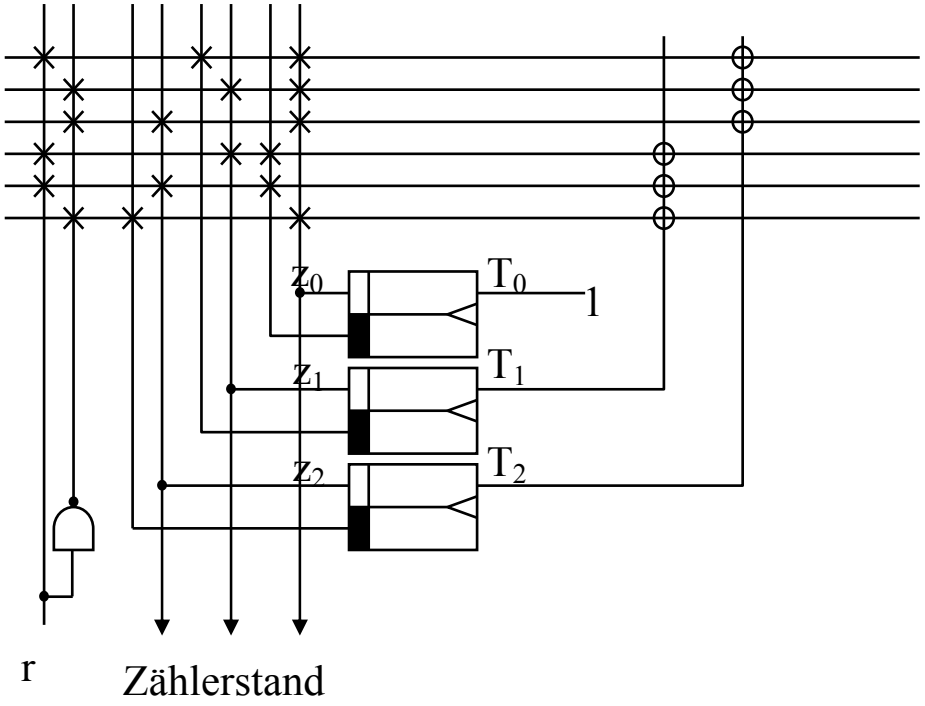
$z_1$

$z_2$

$$T_1 = \bar{z}_0 z_1 r + \bar{z}_0 z_2 r + z_0 \bar{z}_2 \bar{r}$$

$$T_0 = 1$$

Realisierung als FPLA:

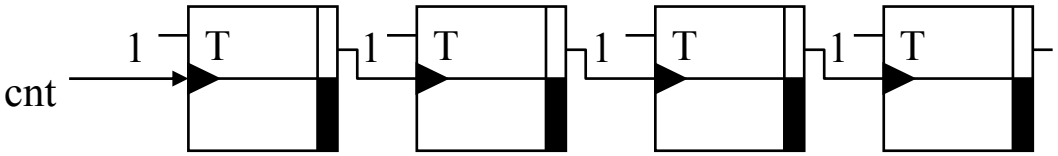



### 5.3.4. Asynchrone Zähler

Alle Schaltwerke, die wir bisher betrachtet haben, waren synchrone Schaltwerke. Das gilt insbesondere für alle Zähler.

**Definition:** Ein Schaltwerk heißt synchrones Schaltwerk, wenn alle Takteingänge mit einem einzigen, zentral verteilten Signal, genannt Taktsignal, beschaltet werden.

Es gibt viele gute Gründe, Schaltwerke synchron zu betreiben. Speziell bei den Zählern gibt es aber auch gebräuchliche asynchrone Varianten, die sich durch besonders einfachen Aufbau auszeichnen. Man betrachte den folgenden Modulo-16-Zähler aus T-Flipflops:




Dieses Schaltwerk zählt die Anzahl der Impulse am Eingang cnt. Sei der Zähler am Anfang im Zustand 0000 (gelesen von links nach rechts, so wie die Anordnung der Flipflops in der Zeichnung). Die vier Flipflops sind negativ flankengetriggert. Nach dem ersten Impuls von cnt wird das erste Flipflop getoggelt, der Zählerstand ist 1000. Alle anderen Flipflops haben noch keine negative Flanke am Takteingang gesehen, halten also ihren Zustand. Beim nächsten cnt-Impuls wechselt das erste Flipflop wieder seinen Wert (diesmal von 1 auf 0). Dadurch bekommt das zweite Flipflop seine erste negative Flanke, es toggelt, der Zählerwert ist 0100. Beim nächsten cnt-Impuls geht der Zähler auf 1100 usw.

Wir sehen, dass die Binärzahlen von 0 bis 15 durchgezählt werden (gespiegelt dargestellt). Beim nächsten Impuls würden nacheinander alle Flipflops den Zustand wechseln, der Zähler fängt wieder bei 0000 an.

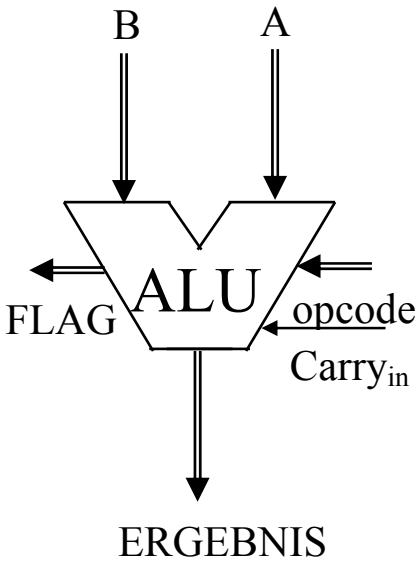
Natürlich funktioniert ein solcher asynchroner Modulo-N-Zähler genauso für alle Zweierpotenzen N mit n = log N Flipflops.

### 5.4. ALU-Aufbau

Eine ALU (arithmetisch-logische Einheit) besteht in der Regel aus

- Addierer
- Logischer Einheit
- Shifter

**Eingänge in eine ALU: zwei Operanden, Instruktionscode**




**Ausgänge einer ALU: Ergebnis, Flags**

### 5.5. Addierer

Kernstück jeder ALU ist ein Addierer. Wir sehen einen Ripple-Carry-Addierer auf der nächsten Folie.

Wie können wir diesen aber auch zum Subtrahieren benutzen?

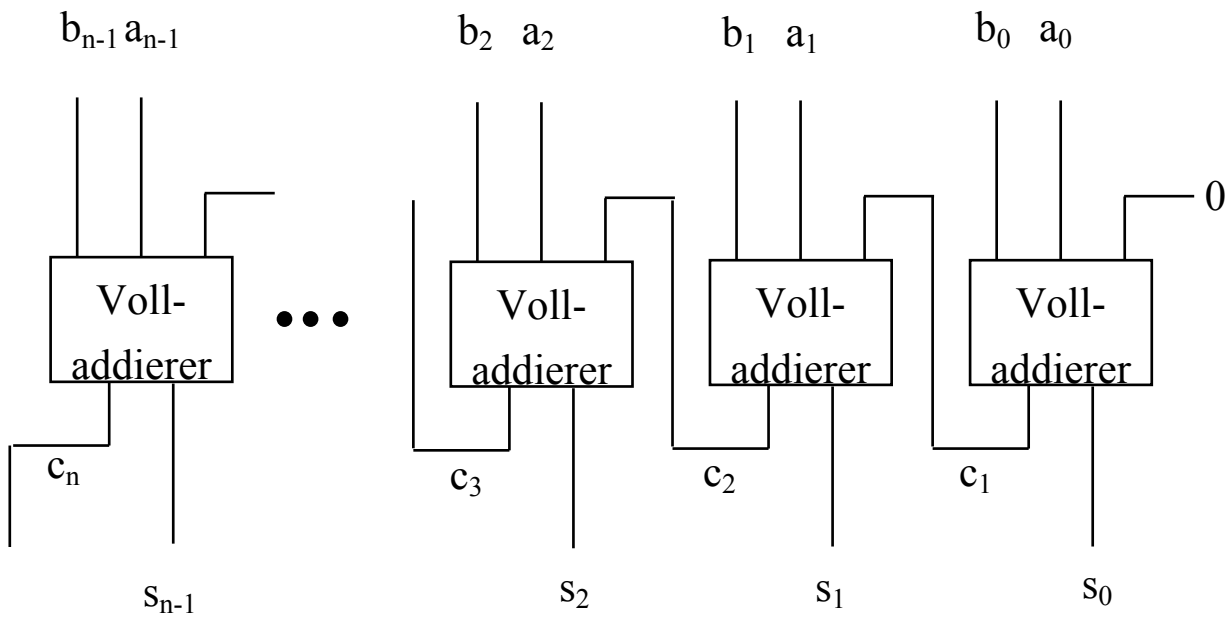
Indem wir das Zweierkomplement des einen Operanden bilden und an den einen Eingang des Addierers führen.

Wie bilden wir das Zweierkomplement?

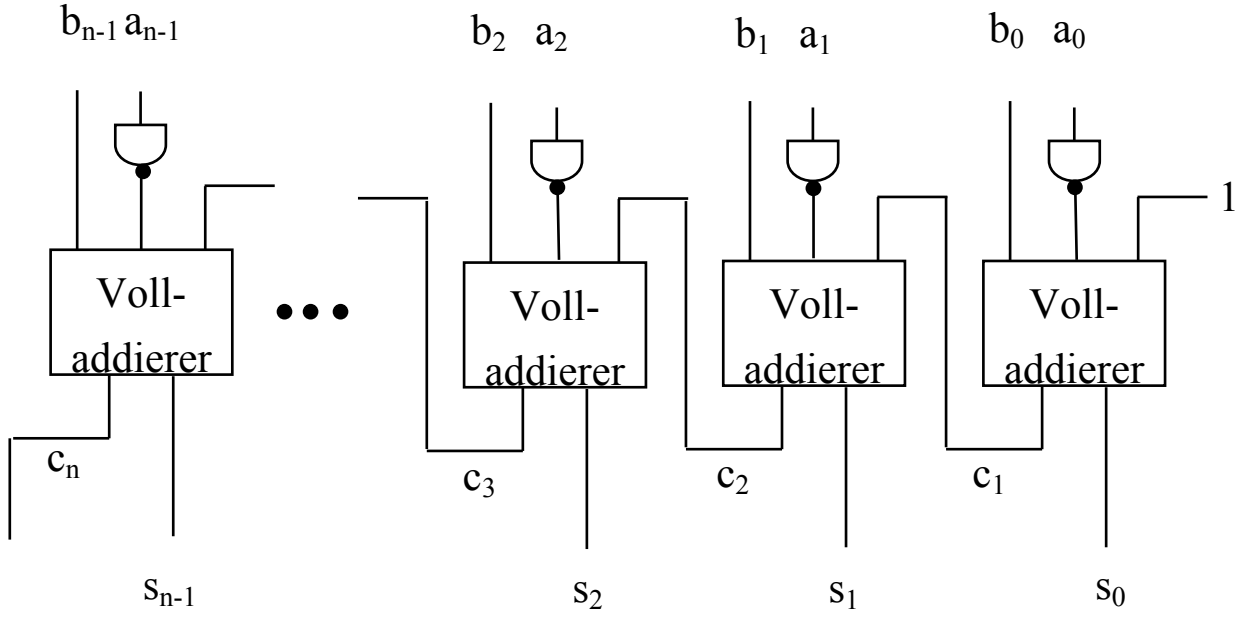
Indem wir jedes Bit invertieren (Einer-Komplement) und noch 1 addieren. Um für diese Addition nicht einen weiteren Addierer zu benötigen, missbrauchen wir einfach den Carry-Eingang des Addierers, in den wir bei der Subtraktion eine 1 anstelle der 0 (bei der Addition) eingeben. Ein entsprechendes Schaltnetz sehen wir auf der übernächsten Folie.

Später werden wir noch weitere Operationen durch den Addierer ausführen lassen. Dadurch wird die Beschaltung seiner Ein- und Ausgänge noch etwas aufwendiger.

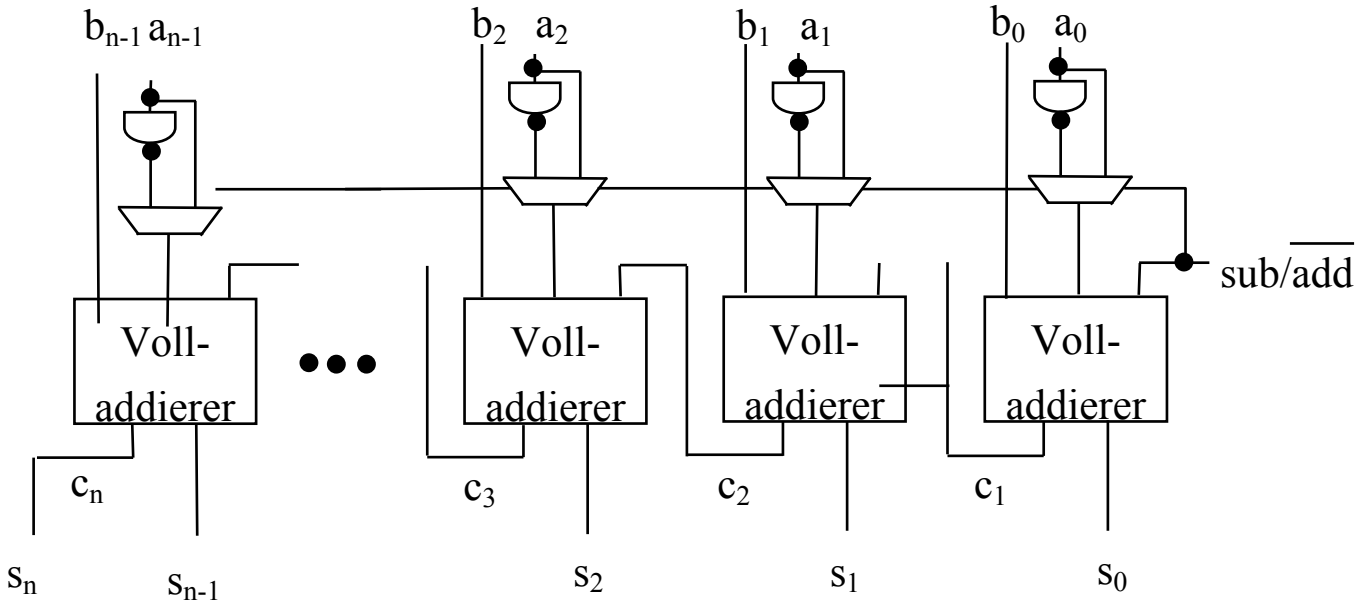
Addierer:




Subtrahierer:



Addierer/Subtrahierer:




## 5.6. Befehlssatz:

Nun müssen wir uns im klaren darüber sein, was für einen Befehlssatz wir mit unserer ALU ausführen können wollen. Die folgende Folie zeigt eine typische Auswahl der Operationen, die auf der ALU eines modernen RISC-Prozessors ausgeführt werden können. Man beachte, dass diese Befehlsauswahl einige Redundanz beinhaltet (der SET-Befehl ist zweimal vorhanden, die logischen Befehle könnten anders codiert werden usw.). Wir entscheiden uns für diese einfache Version, um die Implementierung möglichst übersichtlich zu halten.

Die 16 Befehle werden in vier Bits $cntrl_3$,...,$cntrl_0$ codiert. Dabei entscheidet $cntrl_3$, ob es eine arithmetische Operation ist oder nicht und $cntrl_2$ ob eine shift- oder logische Operation bzw. eine Addition oder Subtraktion.

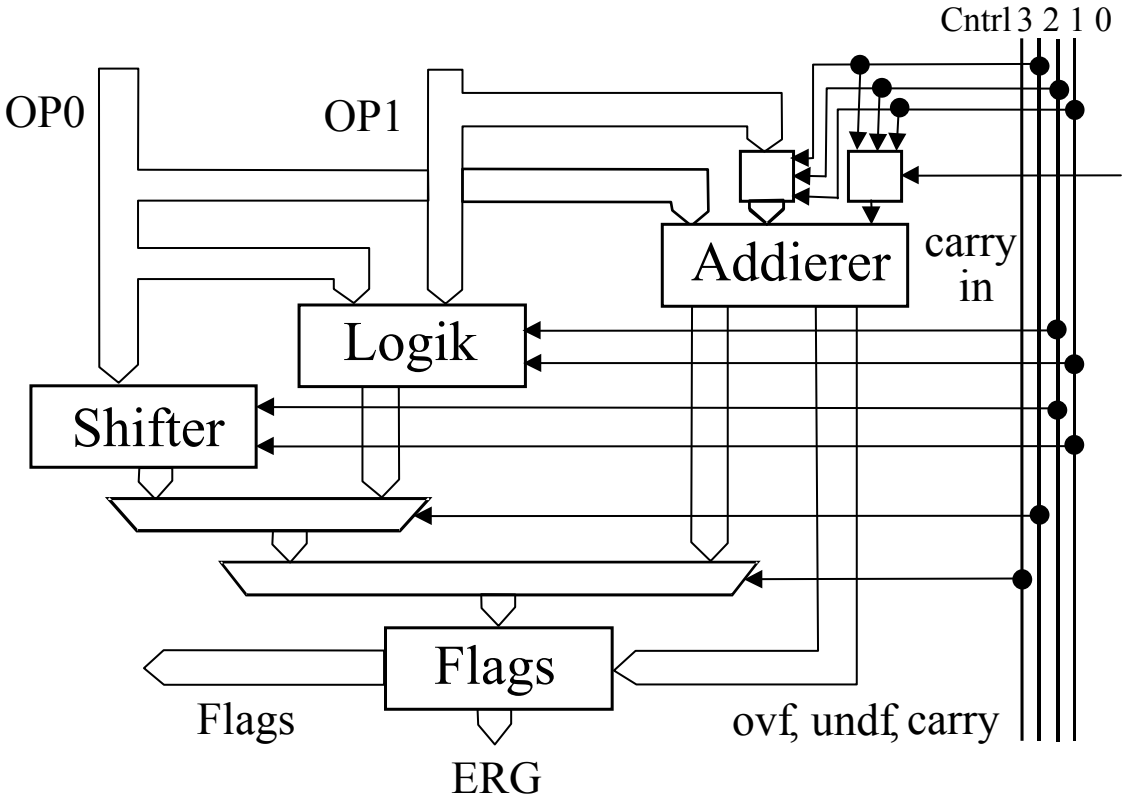
Befehlssatz:

| **Befehl** | **Bedeutung** | **Codierung** | | | |
|---|---|---|---|---|---|
| | | **cntrl3** | **cntrl2** | **cntrl1** | **cntrl0** |
| SET | ERG:=OP0 | 0 | 0 | 0 | 0 |
| DEC | ERG:=OP0-1 | 0 | 0 | 0 | 1 |
| ADD | ERG:=OP0+OP1 | 0 | 0 | 1 | 0 |
| ADC | ERG:=OP0+OP1+$Carry_{in}$ | 0 | 0 | 1 | 1 |
| SET | ERG:=OP0 | 0 | 1 | 0 | 0 |
| INC | ERG:=OP0+1 | 0 | 1 | 0 | 1 |
| SUB | ERG:=OP0-OP1 | 0 | 1 | 1 | 0 |
| SBC | ERG:=OP0-OP1+$Carry_{in}$ | 0 | 1 | 1 | 1 |
| SETF | ERG:=0 | 1 | 0 | 0 | 0 |

| | | | | | |
|---|---|---|---|---|---|
| SLL | ERG:=2*OP0 | 1 | 0 | 0 | 1 |
| SRL | ERG:=OP0 div 2 | 1 | 0 | 1 | 0 |
| SETT | ERG:=-1 | 1 | 0 | 1 | 1 |
| NAND | ERG:=OP0 NAND OP1 | 1 | 1 | 0 | 0 |
| AND | ERG:=OP0 AND OP1 | 1 | 1 | 0 | 1 |
| NOT | ERG:=NOT OP0 | 1 | 1 | 1 | 0 |
| OR | ERG:=OP0 OR OP1 | 1 | 1 | 1 | 1 |

### *5.7. Struktur der ALU:*

Die nächste Folie zeigt den prinzipiellen Aufbau der ALU. Die Steuereingänge wirken einerseits auf die Einheiten (Shifter, Logik, Addierer) selbst, andererseits wählen sie durch Steuerung zweier Datenweg-Multiplexer das Ergebnis aus der jeweils für die aktuell zuständige Einheit, um es an den ERG-Ausgang der ALU weiterzuleiten.



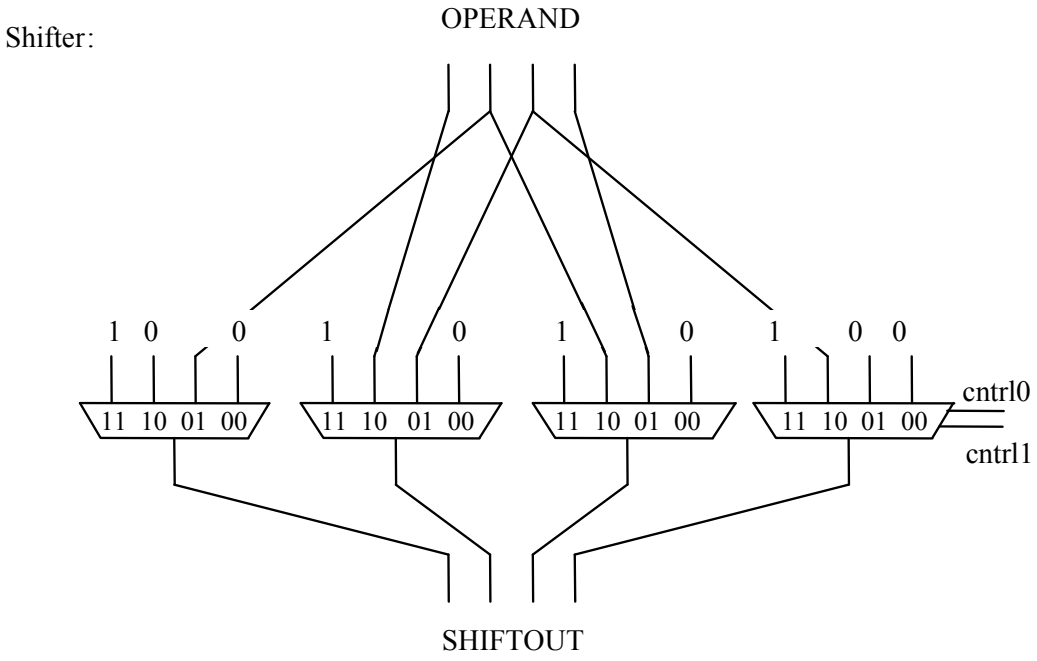
### 5.7.1. Der Shifter:

Wir wissen bereits, was eine Leitung ist und was ein Datenweg-Multiplexer ist. Es bleibt die Aufgabe, die Einheiten mit Leben zu füllen, von denen wir durch den Befehlssatz zunächst nur eine funktionale Beschreibung haben.

Wir fangen mit der einfachsten Einheit an, dem Shifter. Seine Aufgabe ist, die Befehle SETF (setze auf FALSE, setze auf 0), SLL (shift logical left), SLR (shift logical right) und SETT (setze auf TRUE, setze auf -1). Diese Funktionen können mit einfachen 4-auf 1-Multiplexern wahrgenommen werden, einen für jedes Bit des Ergebnisses. Bei den beiden Shift-Befehlen wird das jeweils neu eingeschobene Bit auf 0 gesetzt und das herausgeschobene Bit wird verworfen.

Wenn ein Ringschieben realisiert werden soll, kann man in der Einheit eine entsprechende Modifikation realisieren.

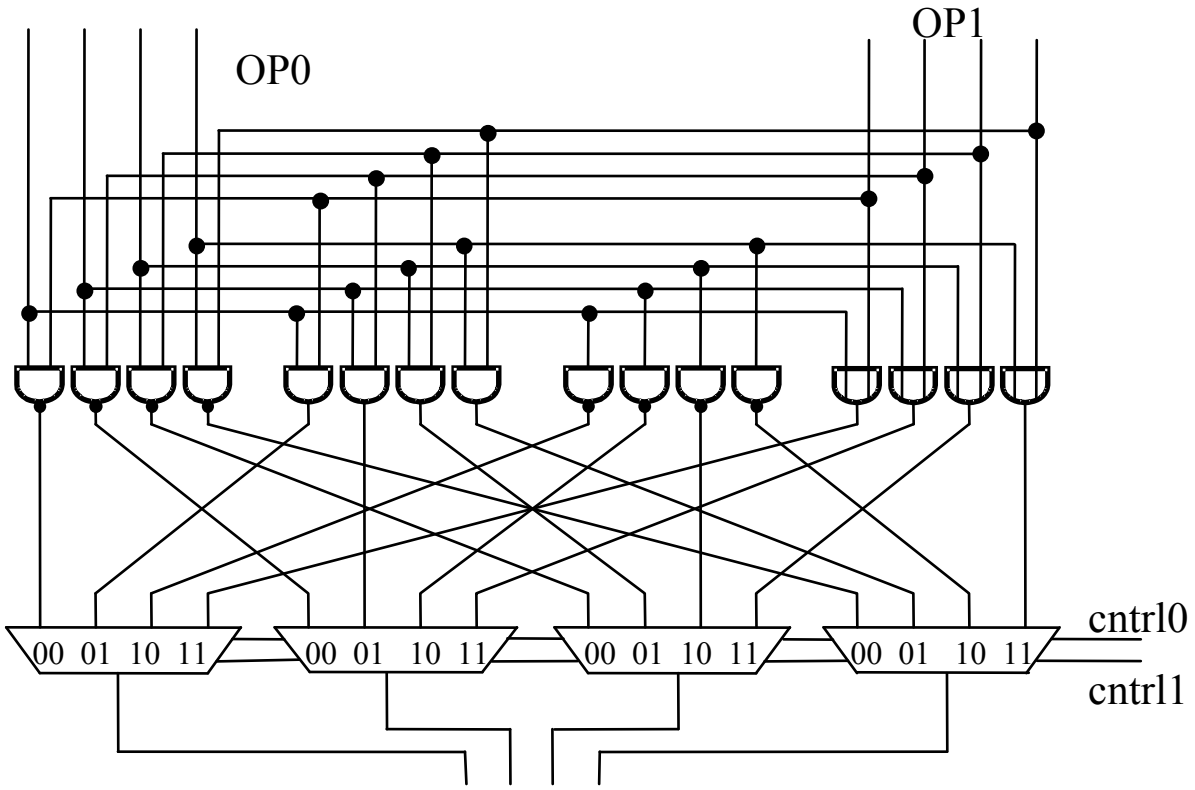
Die folgende Folie zeigt den Shifter für eine Datenbreite von 4 Bit.




### 5.7.2. Die Logik-Einheit:

Die folgende Folie stellt eine (von vielen möglichen gleichwertige) Realisierung der logischen Befehle dar. NAND, AND, NOT, oder OR werden gesondert bit-weise berechnet und über Multiplexer wird das durch den aktuellen Befehl geforderte Ergebnis ausgewählt.

Die Logik-Einheit:




Eingänge des Addierers

Der Addierer hat als einen Operanden immer OP0. Der zweite zu addierende Operand kann OP1, 0, -1, 1, oder -OP1 sein, je nachdem, ob addiert, subtrahiert, inkrementiert, dekrementiert oder unverändert durchgereicht werden soll. Diese Fälle werden folgendermaßen behandelt:

Bei der einfachen Addition ist der zweite Eingang gleich OP1, der Carry-Eingang gleich 0. Bei der Addition mit Berücksichtigung des alten $Carry_{in}$ ist der Eingang gleich OP1, der Carry-Eingang gleich $Carry_{in}$. Bei der Subtraktion werden alle Bits von OP1 invertiert und der Carry-Eingang ist 1. Auf diese Weise wird das Zweierkomplement von OP1 zu OP0 addiert. Bei der Subtraktion mit Carry werden die Bits von OP1 invertiert und $Carry_{in}$ wird an den Carry-Eingang des Addierers gelegt.

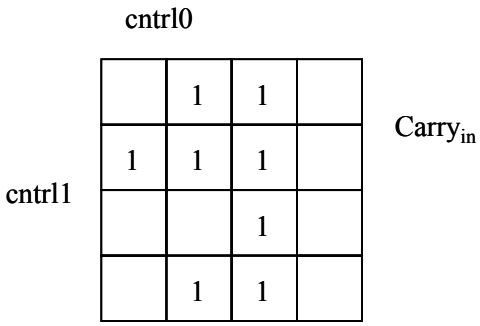
Bei SET-Befehlen wird OP0 + 0 berechnet. Es gibt zwei SET-Befehle. Beim ersten wird +0 addiert, beim zweiten -0 subtrahiert. Also ist beim ersten der zweite Addierer-Eingang gleich 0 und das Carry ist 0 und beim zweiten der zweite Addierereingang auf -1 (alle Bits sind 1) und das Carry auf 1.

Beim Inkrementieren ist der zweite Addierer-Eingang auf 0 und der Carry-Eingang auf 1, beim Dekrementieren ist der zweite Addierer-Eingang auf -1 (alle Bits sind 1) und der Carry-Eingang ist auf 0.

Die Schaltnetze für den Carry-Eingang und den zweiten Eingang des Addierers werden auf den folgenden Folien abgeleitet. Dabei ist das Schaltnetz für den zweiten Eingang des Addierers für jedes Bit einzeln vorzusehen.

C-Eingang des Addierers

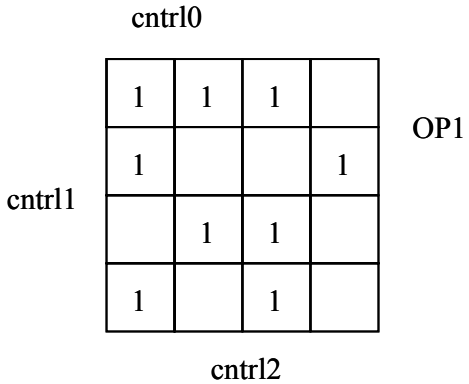
| Befehl | $Carry_{in}$ | cntrl2 | cntrl1 | cntrl0 | C-Eingang |
|---|---|---|---|---|---|
| set | 0 | 0 | 0 | 0 | 0 |
| dec | 0 | 0 | 0 | 1 | 0 |
| add | 0 | 0 | 1 | 0 | 0 |
| adc | 0 | 0 | 1 | 1 | 0 |
| set | 0 | 1 | 0 | 0 | 1 |
| inc | 0 | 1 | 0 | 1 | 1 |
| sub | 0 | 1 | 1 | 0 | 1 |
| sbc | 0 | 1 | 1 | 1 | 0 |
| set | 1 | 0 | 0 | 0 | 0 |
| dec | 1 | 0 | 0 | 1 | 0 |
| add | 1 | 0 | 1 | 0 | 0 |
| adc | 1 | 0 | 1 | 1 | 1 |
| set | 1 | 1 | 0 | 0 | 1 |
| inc | 1 | 1 | 0 | 1 | 1 |
| sub | 1 | 1 | 1 | 0 | 1 |
| sbc | 1 | 1 | 1 | 1 | 1 |




$$\text{C-Eingang} = \text{cntrl2}\,\overline{\text{cntrl1}} + \overline{\text{cntrl0}}\,\text{cntrl2} + \text{cntrl0}\,\text{cntrl1}\,\text{carry}_{in}$$

OP-Eingang des Addierers

| Befehl | OP1 | cntrl2 | cntrl1 | cntrl0 | OP-Eingang |
|---|---|---|---|---|---|
| set | 0 | 0 | 0 | 0 | 0 |
| dec | 0 | 0 | 0 | 1 | 1 |
| add | 0 | 0 | 1 | 0 | 0 |
| adc | 0 | 0 | 1 | 1 | 0 |
| set | 0 | 1 | 0 | 0 | 1 |
| inc | 0 | 1 | 0 | 1 | 0 |
| sub | 0 | 1 | 1 | 0 | 1 |
| sbc | 0 | 1 | 1 | 1 | 1 |
| set | 1 | 0 | 0 | 0 | 0 |
| dec | 1 | 0 | 0 | 1 | 1 |
| add | 1 | 0 | 1 | 0 | 1 |
| adc | 1 | 0 | 1 | 1 | 1 |
| set | 1 | 1 | 0 | 0 | 1 |
| inc | 1 | 1 | 0 | 1 | 0 |
| sub | 1 | 1 | 1 | 0 | 0 |
| sbc | 1 | 1 | 1 | 1 | 0 |



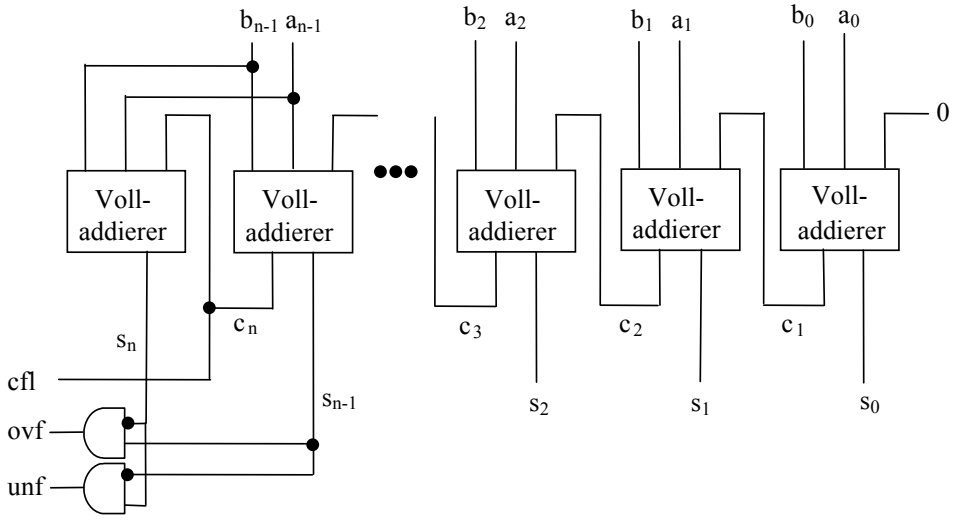

$$\text{OP-Eingang} = \text{cntrl0}\,\overline{\text{cntrl1}}\,\text{OP1} + \overline{\text{cntrl0}}\,\overline{\text{cntrl2}}\,\text{OP1} + \overline{\text{cntrl1}}\,\text{cntrl2}\,\text{OP1} + \text{cntrl1}\,\text{cntrl2}\,\overline{\text{OP1}} + \overline{\text{cntrl0}}\,\overline{\text{cntrl1}}\,\text{cntrl2}$$

Überlauf-Erkennung beim Addierer

Wie bereits aus der ersten Vorlesung bekannt, können Über- und Unterläufe bei der Addition anhand eines zusätzlichen Sicherungsbits erkannt werden, das eine Kopie des höchstsignifikanten Bits darstellt. Man benutzt nun für eine m-Bit-Addition einen m+1-Bit Addierer, wobei die Sicherungsstelle ganz normal mitaddiert wird. Das Ergebnis der Addition ist also die Summe $s_{m-1}$, $s_{m-2}$, ..., $s_1$, $s_0$, das ausgehende Carry-Bit (= Carry-Flag) $c_m$, sowie ein künstliches Summenbit $s_m$. Das künstliche Carry-Bit $c_{m+1}$ hat keine Bedeutung und wird daher nicht verwendet. Ein Überlauf (Ergebnis ist größer als die größte darstellbare Zahl) ist nun aufgetreten, wenn $s_{m-1}$ gleich 0 und $s_m$ gleich 1 ist. Wenn $s_m$ gleich 0 und $s_{m-1}$ gleich 1 ist, hat ein Unterlauf (Ergebnis ist kleiner als die kleinste darstellbare Zahl) stattgefunden. Wenn $s_{m-1}$ gleich $s_m$ ist, ist weder Überlauf noch Unterlauf aufgetreten, also ist die Addition fehlerfrei verlaufen.

Das Schaltnetz auf der folgenden Folie zeigt die Ergänzung unseres Addierers, mit der wir Über- und Unterläufe erkennen und als Flags (ovf (overflow) und unf (underflow)) an die Flag-Einheit übermitteln können.

**Signale für Flags**

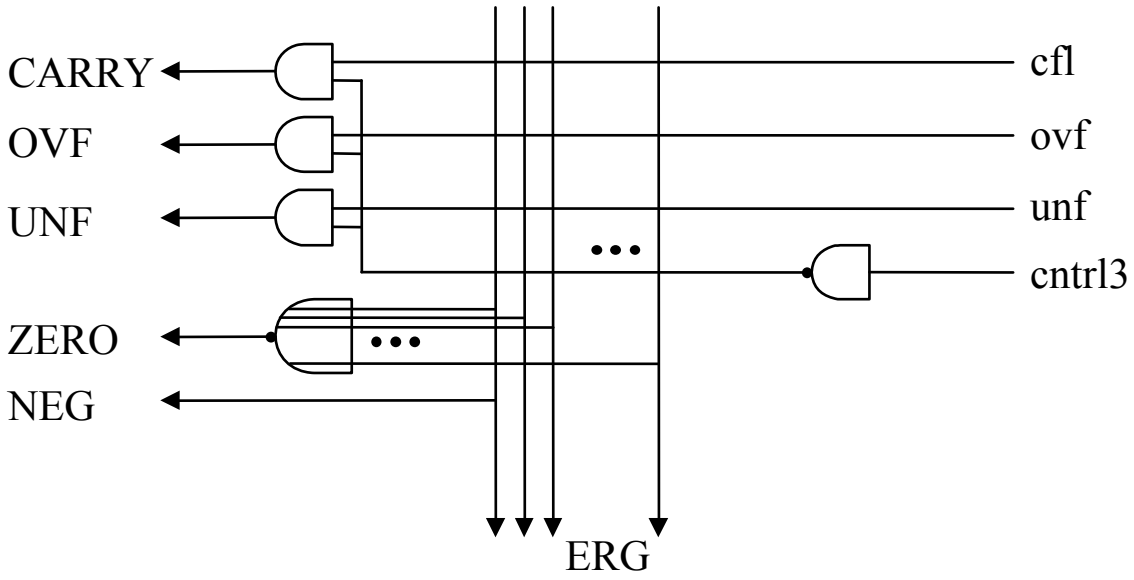



**Flag-Test**

Fünf Flags sollen generiert werden:

- Carry-Flag (=1, falls eine arithmetische Operation ein Carry erzeugt hat)
- Neg-Flag (=1, wenn das Ergebnis eine negative Zahl darstellt)
- Zero-Flag (=1, wenn das Ergebnis gleich 0 ist)
- ovf-Flag (=1, wenn eine arithmetische Operation einen Überlauf erzeugt hat)
- unf-Flag (=1, wenn eine arithmetische Operation einen Unterlauf erzeugt hat)

Im Einzelfall kann die Anforderung an Flag-Einheiten sehr viel komplizierter sein, insbesondere, wenn es sich um einen Prozessor mit impliziter Condition-Behandlung handelt. Für unsere Zwecke genügt aber eine so einfache Einheit, wie sie auf der folgenden Folie dargestellt ist.


CARRY
OVF
UNF
ZERO
NEG
cfl
ovf
unf
cntrl3
ERG

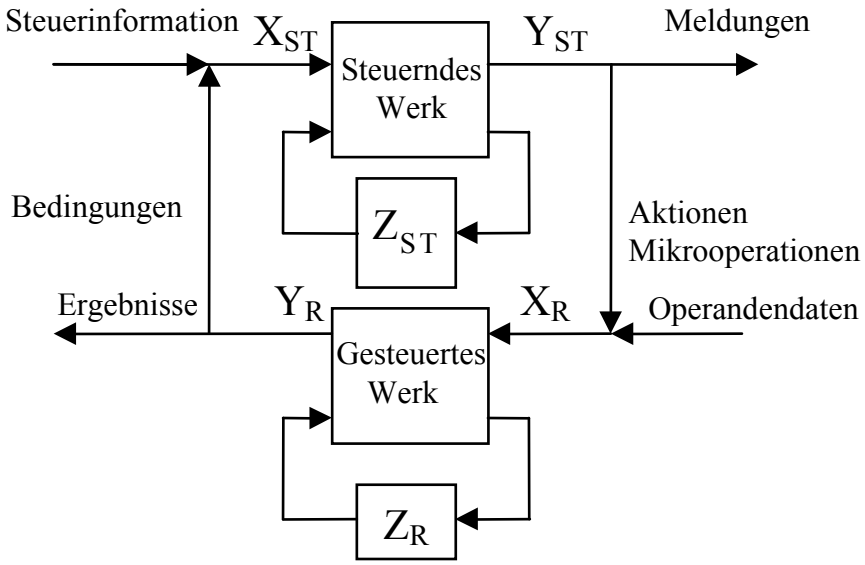
# 6. Steuerwerke

## *6.1. Steuerwerk*

Wir kennen jetzt den Aufbau von Rechenwerken, z.B. einem Addierer oder einer logischen Einheit. In einem Computer müssen diese Einheiten aber zur richtigen Zeit aktiviert werden, Daten müssen über Multiplexer an die richtigen Einheiten geführt werden, mit anderen Worten: Die Abläufe müssen gesteuert werden. Diese Funktionen übernehmen ebenfalls Schaltwerke, sogenannte **Steuerwerke**. Wir wollen in diesem Abschnitt an einem einfachen Beispiel die Funktion eines Steuerwerks studieren.

Die folgende Folie zeigt das grundsätzliche Prinzip eines Steuerwerks: Es gibt ein steuerndes (Schalt-)werk und ein gesteuertes Werk. Das gesteuerte Werk ist häufig ein Rechenwerk. Das Steuerwerk hat - wie jedes Schaltwerk - Eingaben, Ausgaben und Zustände. Die Eingaben sind einerseits die Befehle der übergeordneten Instanz (z.B. des Benutzers oder eines übergeordneten Steuerwerks), andererseits die **Bedingungen.** Dies sind Ausgaben des Rechenwerks, mit dem es dem Steuerwerk Informationen über den Ablauf der gesteuerten Aufgabe gibt. Die Ausgaben des steuernden Werks sind einerseits die Meldungen zur übergeordneten Instanz und andererseits die **Mikrobefehle (Aktionen)**, die als Eingaben in das gesteuerte Werk gehen, und dadurch die Abläufe dort kontrollieren.

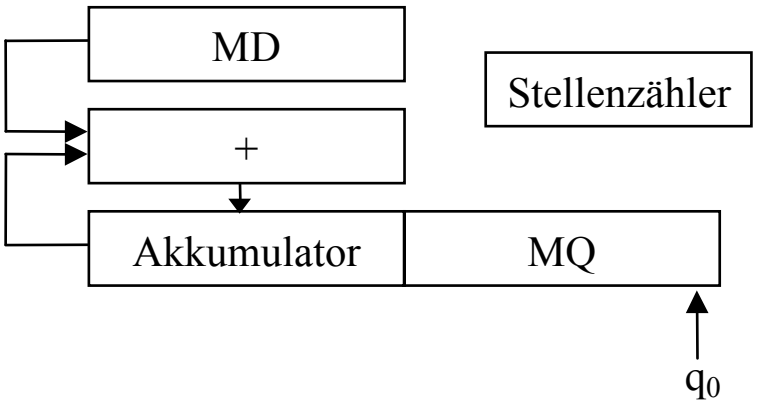
**Steuerwerksprizip:**




**Multiplizierwerk:**

Das folgende Beispiel soll das Steuerwerksprinzip illustrieren:

Ein Multiplizierwerk ist zu steuern, das zwei Zahlen (z.B. mit jeweils vier Bit) multiplizieren kann. Das Multiplizierwerk besteht aus einem Multiplikanden-Register MD, einem Akkumulatorregister AKK, das als paralleles Register und als Schieberegister genutzt werden kann, einem Multiplikatorregister MQ mit derselben Eigenschaft, einem Zähler, genannt Stellenzähler (SZ) und einem Addierer. Mit $q_0$ wird die letzte Stelle von MQ bezeichnet.




Für eine Multiplikation werden Multiplikand und Multiplikator in die entsprechenden Register geladen, der Akkumulator wird mit 0 vorbesetzt. Sodann sendet die Steuerung einen Mikrobefehl „-n“, der den Stellenzähler auf -n setzt. n soll dabei die Anzahl der Stellen der zu multiplizierenden Operanden sein. Wenn $q_0$ 1 ist, wird jetzt die Mikrooperation „+“ generiert, die das Addieren des gegenwärtigen Akkumulators mir dem Register MD bewirkt, das Ergebnis wird im Akkumulator gespeichert. Danach werden die Mikrooperationen „S“ und „SZ“ generiert. S bewirkt ein Verschieben um ein Bit nach rechts im Schieberegister bestehend aus Akkumulator und MQ. SZ bewirkt ein Inkrementieren des Stellenzählers. Wenn der Stellenzähler den Wert 0 erreicht hat, terminiert der Prozess, das Ergebnis steht im Schieberegister. Wenn der Stellenzähler nicht 0 ist, wird wiederum $q_0$ interpretiert. Wenn $q_0$ 0 ist, wird die Mikrooperation „0“ generiert, die kein Register verändert.

Die folgende Tabelle zeigt ein Beispiel für die Multiplikation der Zahlen 0101 und 1011 (5 * 11).

### 6.1.1. Abfolge im Multiplizierwerk:

MD = 0101

| Akku | MQ | SZ | q0 | SZ=0 | Start | Microoperation |
|---|---|---|---|---|---|---|
| 0 0 0 0 | 1 0 1 1 | 000 | 1 | 1 | 1 | -n |
| 0 0 0 0 | 1 0 1 1 | 100 | 1 | 0 | 0 | + |
| 0 1 0 1 | 1 0 1 1 | 100 | 1 | 0 | 0 | S,SZ |
| 0 0 1 0 | 1 1 0 1 | 101 | 1 | 0 | 0 | + |
| 0 1 1 1 | 1 1 0 1 | 101 | 1 | 0 | 0 | S,SZ |
| 0 0 1 1 | 1 1 1 0 | 110 | 0 | 0 | 0 | 0 |
| 0 0 1 1 | 1 1 1 0 | 110 | 0 | 0 | 0 | S,SZ |
| 0 0 0 1 | 1 1 1 1 | 111 | 1 | 0 | 0 | + |
| 0 1 1 0 | 1 1 1 1 | 111 | 1 | 0 | 0 | S,SZ |
| 0 0 1 1 | 0 1 1 1 | 000 | 1 | 1 | 0 | |

Wir wissen bereits, wie wir ein solches Rechenwerk bauen müssten, denn es besteht nur aus uns bekannten Komponenten (Register, Schieberegister, Zähler, Addierer). An dieser Stelle interessiert uns nun aber, wie wir die Mikrooperationen zum richtigen Zeitpunkt generieren können, also wie wir dieses Rechenwerk „steuern“ können. Dazu machen wir uns klar:

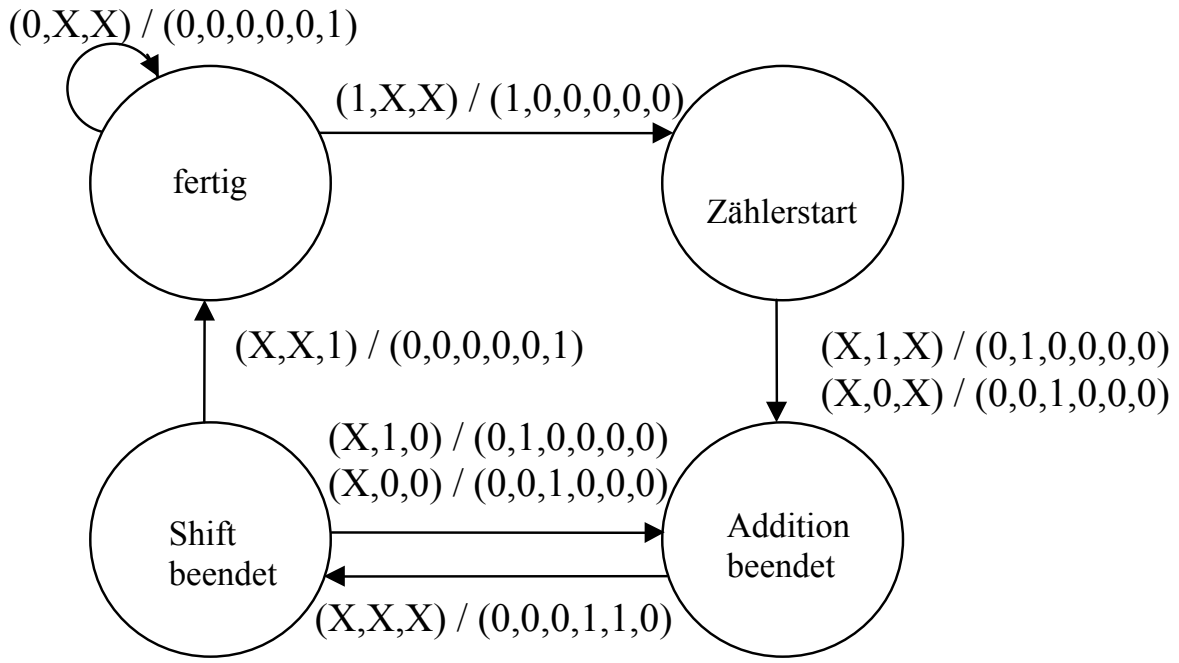
- Wenn ein „Start“-Signal kommt, muss der Stellenzähler mit „-n“ initialisiert werden.

- Wenn $q_0$ interpretiert wird, muss MD genau dann auf den Akkumulator addiert werden, wenn $q_0 = 1$ ist.
- Nach jedem solchen Additionsschritt muss der Akkumulator und das MQ um ein Bit geschoben und der Stellenzähler um eins erhöht (inkrementiert) werden.
- Wenn irgendwann der Stellenzähler den Wert 0 erreicht, ist die Multiplikation beendet, das Ergebnis steht im Schieberegister aus Akkumulator und MQ.

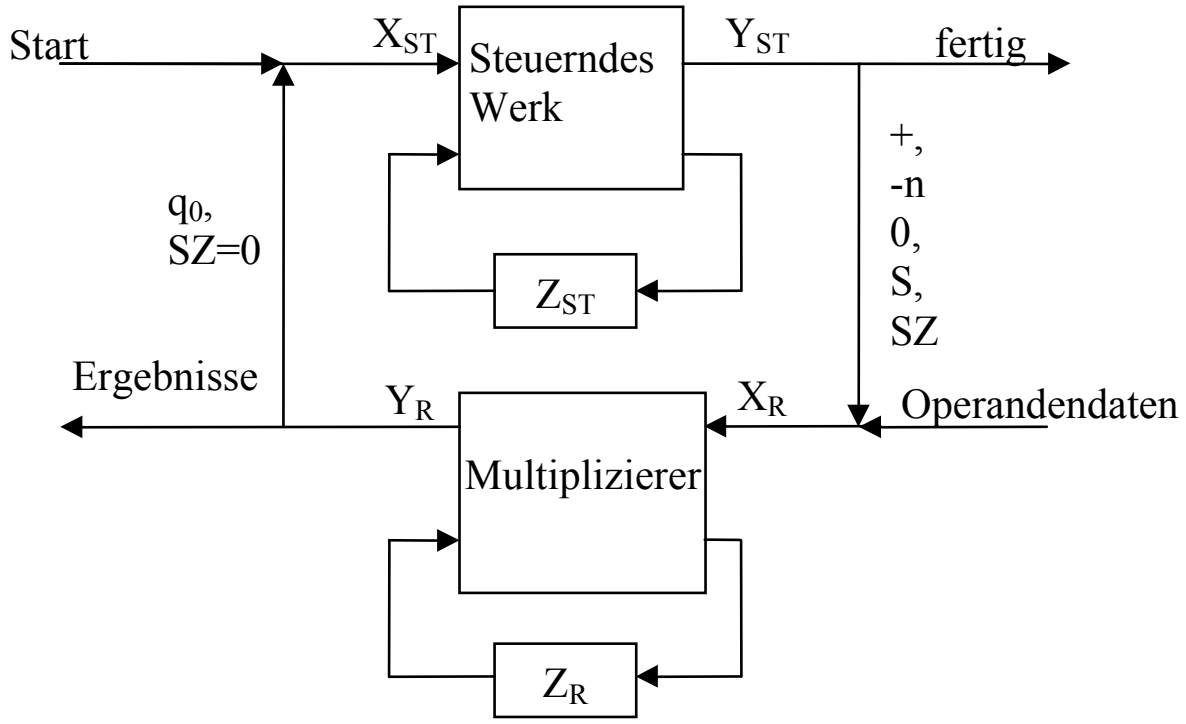
Die folgende Folie zeigt, welche dieser Signale Ein- und Ausgaben welcher Werke sind.

Aus diesen Informationen können wir danach einen Automatengraphen entwickeln.

**Steuerwerk:**




Eingaben: (Start, $q_0$, SZ=0) Ausgaben (-n, +, 0, S, SZ, fertig)




Wir codieren die Zustände mit

00: fertig

01: Zählerstart

10: Addition beendet

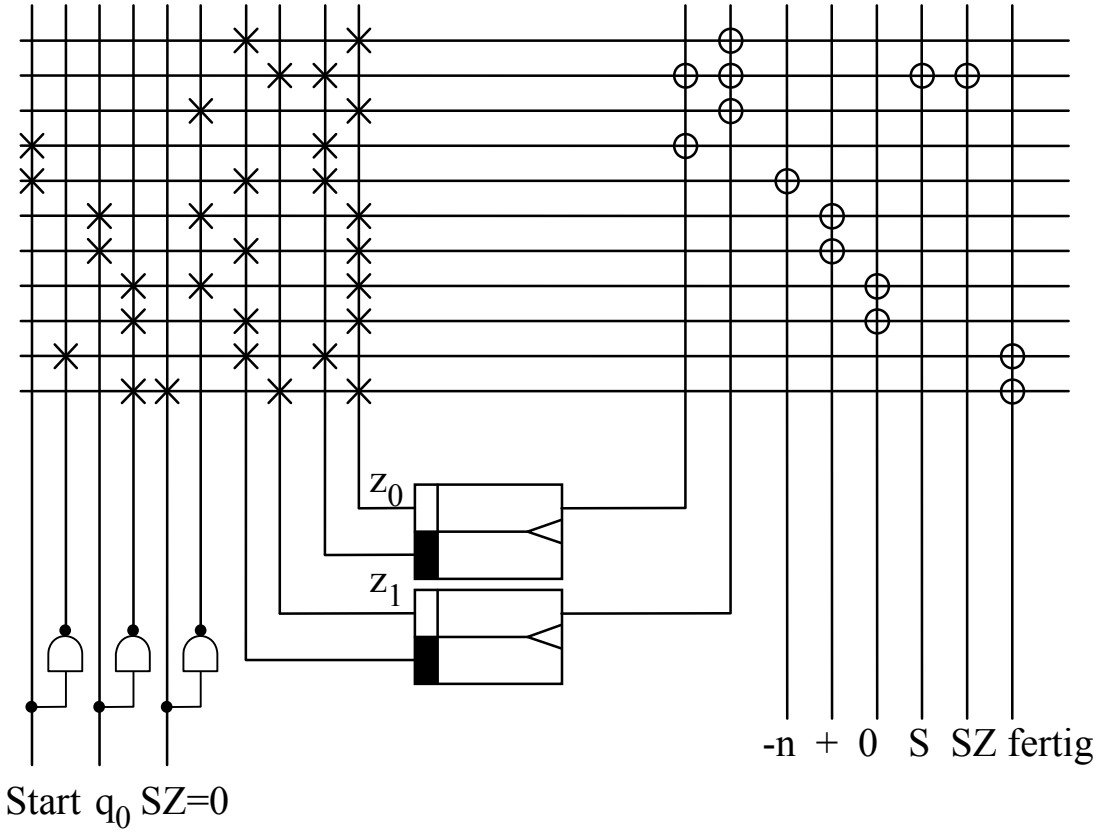
11: Shift beendet

Dann entspricht dieser Automat der folgenden Wertetabelle

| **Start** | **q0** | **SZ=0** | **z1** | **z0** | **z‘1** | **z‘0** | **-n** | **+** | **0** | **S** | **SZ** | **fertig** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | X | X | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | X | X | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| X | 1 | X | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| X | 0 | X | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| X | X | X | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| X | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| X | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| X | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Das ergibt nach Minimierung die Schaltwerksrealisierung auf der nächsten Folie:

Realisierung als FPLA:


$z_0$
$z_1$
-n + 0 S SZ fertig
Start $q_0$ SZ=0

# 7. Befehlssatzarchitektur

Die Befehlssatzarchitektur ist der Teil der Maschine, die sichtbar ist für den Programmierer und Compiler-Schreiber.

## *7.1. Die RISC Idee*

Mitte der 80-er Jahre wurde die Prozessorlandschaft revolutioniert durch ein neues Grundkonzept: RISC (reduced instruction set computer). Nachdem vorher die Philosophie der Architekten darin bestand, möglichst komfortable Funktionen bereitzustellen, damit der Anwender alle erdenklichen Operationen als Maschineninstruktionen der Hardware vorfand, fand man jetzt zurück zur Einfachheit:

Es zeigte sich, dass durch Vereinfachung des Befehlformats, der Speicheradressierung, des Befehlssatzes und des Registerzugriffs die Leistung eines Rechners signifikant gesteigert werden konnte. Dies wurde erreicht, indem man quantitative Entwurfsprinzipien entwickelte, mit denen man die Qualität von Rechnerarchitekturentscheidungen bewerten kann. Dieser Ansatz wurde von wenigen Grundprinzipien geleitet:

1. Klein ist schnell: Kleinere Hardwareeinheiten können schneller sein, weil Leitungen kürzer sind, Transistoren kleiner usw.
2. Make the common case fast: Wenn eine Entwurfsentscheidung zu treffen ist, ist zu untersuchen, wie häufig eine Verbesserung ggf. benutzt wird.
3. Amdahls Gesetz: Die Gesamtbeschleunigung einer Architekturveränderung ist begrenzt durch den Anteil $A_u$ an der ursprünglichen Berechnung, der nicht beschleunigt werden kann:

$$\text{Speedup} = \frac{1}{A_u + \frac{1 - A_u}{\text{Teilbeschleunigung}}}$$

4. Die CPU-Performance-Gleichung: Die Ausführungszeit eines Programms kann berechnet werden als

$$\text{CPU-Zeit} = \text{IC} * \text{CPI} * \text{Zykluszeit};$$

Dabei ist IC (instruction count) die Anzahl der Instruktionen des Programms, CPI (clocks per instruction) die mittlere Anzahl von Takten für die Ausführung einer Instruktion und Zykluszeit die Zeit für einen Taktzyklus also der reziproke Wert der Taktfrequenz.

Die RISC Idee zeichnet sich durch vier Charakteristika aus:

1. Einfacher Instruktionssatz
2. Feste Befehlswortlänge, wenige Befehlsformate
3. Wenige Adressierungsarten, einfache Adressierungsarten
4. Registermaschine mit Load-Store Architektur

Diese Punkte werden im folgenden näher erläutert.

## *7.2. Akkumulator-Architektur*

Die Akkumulator-Architektur hat ein ausgezeichnetes Register: den Akkumulator. Dieser ist in jeder Operation als ein (impliziter) Operand beteiligt. Load und Store wirken nur auf den Akkumulator. Alle arithmetischen und logischen Operationen benutzen den Akkumulator

sowohl als einen Operanden als auch als Zielregister. Somit kommen alle Operationen mit nur einer Adresse aus. Dies ist ein Vorteil, wenn das Befehlsformat klein ist, so dass nicht genügend Bits für die Adressen mehrerer Operanden zur Verfügung stehen. Daher ist die Akkumulator-Architektur (insbesondere im 8-Bit Bereich, z.B. Mikrocontroller) auch heute noch gebräuchlich.

Eine typische Befehlsfolge zur Addition zweier Zahlen in der Akkumulator-Architektur sieht folgendermaßen aus:

**Load A**

**Add B**

**Store C**

## *7.3. General Purpose Register Architekturen (GPR-architectures)*

General purpose Register sind innerhalb eines Taktzyklus zugreifbare Speicher im Prozessor, die als Quell und Zieladresse zugreifbar sind. Sie sind nicht für bestimmte Maschinenbefehle reserviert.

GPR-Architekturen sind heute die gebräuchlichsten. Fast jeder nach 1980 entwickelte Prozessor ist ein GPR-Prozessor. Warum?

1. **Register sind schneller als Hauptspeicher.**
2. **Register sind einfacher und effektiver für einen Compiler zu nutzen.**

   **Beispiel:**

   **Operation G = (A*B)+(C*D)+(E*F)**

   die Multiplikationen können in beliebiger Reihenfolge ausgeführt werden. Dies hat Vorteile in Sinne von Pipelining. Auf einer Stack-Maschine dagegen z.B. müssten für eine Vertauschung aufwendige Push- und Pop-Operationen zwischengeschaltet werden eine Akkumulator Architektur würde durch Umladen des Akkumulators gebremst.

3. **Register können Variablen enthalten.**

Das verringert den Speicherverkehr und beschleunigt das Programm (da Register schneller sind als Speicher). Sogar die Codierung wird kompakter, da Register mit weniger Bits adressierbar sind als Speicherstellen. Dies erlaubt ein einfacheres Befehlsformat, was sich in der Compiler-Effizienz und im Speicherverkehr positiv auswirkt.

Für Compiler-Schreiber ist es optimal, wenn alle Register wirklich GPRs sind. Ältere Maschinen treffen manchmal Kompromisse, indem bestimmte Register für bestimmte Befehle vorzusehen sind. Dies reduziert jedoch faktisch die Anzahl der GPRs, die zur Speicherung von Variablen benutzt werden können.

Zwei Kriterien unterteilen die Klasse der GPR-Architekturen:

1. **Wie viele Operanden kann ein Typischer ALU-Befehl haben?**
2. **Wie viele Operanden dürfen Speicheradressen sein?**

Bei einem 3-Operanden Format gibt es zwei Quell-Operanden und einen Zieloperanden.

Bei einem 2-Operanden Format muss ein Operand als Quelle und Ziel dienen, d.h. er wird durch die Operation zerstört.

### 7.3.1. Typen von GPR-Architekturen

1. **Register-Register-Maschinen (load-store-Architekturen)**

   Alle arithmetischen Operationen greifen nur auf Register zu. Die einzigen Befehle, die Speicherzugriffe machen sind **load** und **store**. Eine typische Befehlsfolge für das obige Beispiel wäre:

   **Load R1, A**

   **Load R2, B**

   **Add R3, R1, R2**

   **Store C, R3**

2. **Register-Speicher Maschinen**

   Bei Register-Speicher-Maschinen können die Befehle ihre Operanden aus dem Hauptspeicher oder aus den Registern wählen. Dies erlaubt gegenüber den load-store-Architekturen einen geringeren IC. Dafür muss aber der größere Aufwand eines Speicherzugriffs in jeder Operation in Kauf genommen werden, was in der Regel auf Kosten der CPI geht. Bekannteste Vertreter der Klasse ist die Intel 80x86-Familie und der Motorola 68000. Das obige Beispiel sieht auf einer Register-Speicher-Architektur so aus:

   **Load R1, A**

   **Add R1, B**

   **Store C, R1**

3. **Speicher-Speicher-Maschinen**

   Diese verfügen über die größte Allgemeinheit, was den Zugriff auf Operanden angeht. Operanden können aus Registern oder Speicherzellen geholt werden, die Zieladresse kann ebenfalls ein Register oder eine Speicherzelle sein. Unterschiedliche in der Praxis übliche Befehlsformante erlauben 2 oder 3 Operanden in einem Befehl. Im Falle einer 3-Operanden Maschine ist das Beispiel mit **Add C,A,B** abgehandelt. Dieser Vorteil wird mit aufwendigem CPU-Speicher-Verkehr bezahlt.

### 7.3.2. Typen von GPR-Architekturen:

| Typ | Vorteile | Nachteile |
|---|---|---|
| Register-Register (0,3) | Einfaches Befehlsformat fester Länge. Einfaches Modell zur Code Generierung. Instruktionen brauchen alle etwa gleichviel Takte. | Höherer IC als die beiden anderen. |
| Register-Speicher (1,2) | Daten können zugegriffen werden, ohne sie erst zu laden. Instruktions-Format einfach. | Jeweils ein Operand in einer zweistelligen Operation wird zerstört. CPI variiert je nachdem von wo die Operanden geholt werden. |
| Speicher-Speicher (3,3) | Sehr kompakt. Braucht keine Register für Zwischenergebnisse | Hoher CPI. Viele Speicherzugriffe. Speicher-CPU-Flaschenhals. |

### 7.3.3. Speicheradressierung

Unabhängig davon, welche Art von Maschine man gewählt hat, muss festgelegt werden, wie ein Operand im Hauptspeicher adressiert werden kann.

Was können Operanden sein?

**Bytes (8 Bit), Half Words 16 Bit), Words (32 Bit), Double Words (64 Bit).**

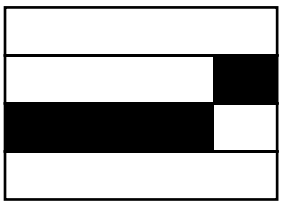
Es gibt unterschiedliche Konventionen, wie die Bytes innerhalb eines Wortes anzuordnen sind:

**Litte Endian bedeutet: Das Byte mit Adresse xxxx....x00 ist das Least Significant Byte.**

**Big Endian bedeutet: Das Byte mit Adresse xxxx....x00 ist das Most Significant Byte.**

In Big Endian ist die Adresse eines Wortes die des MSByte, in Little Endian die des LSByte. **Little Endian: 80x86, VAX, Alpha; Big Endian: MIPS, Motorola, Sparc;**

In vielen Maschinen müssen Daten entsprechend ihrem Format ausgerichtet **(aligned)** sein. Das heißt, dass ein Datum, das s Byte lang ist an einer Speicheradresse steht, die kongruent 0 mod s ist. Beispiele für ausgerichtete und nicht ausgerichtete Daten sind in der folgenden Tabelle:



**Ausrichtung auf Wortgrenzen:**

| Adressiertes Objekt | Ausgerichtet auf Byte | Nicht ausgerichtet auf Byte |
|---|---|---|
| Byte | 0,1,2,3,4,5,6,7 | Nie |
| Half word | 0,2,4,6 | 1,3,5,7 |
| Word | 0,4 | 1,2,3,5,6,7 |
| Double word | 0 | 1,2,3,4,5,6,7 |

Warum Speicherausrichtung?

Speicher sind von der Hardware der DRAM- oder heute auch SDRAM-Bausteine her auf die Ausrichtung auf Wortgrenzen hin optimiert. Deshalb ist bei den Maschinen, die keine Ausrichtung fordern, der Zugriff auf nicht ausgerichtete Daten langsamer als auf ausgerichtete Daten. Zum Beispiel müssen bei einem 32-Bit Zugriff zwei 32-Bit Daten über den Datenbus gelesen werden, und von diesen müssen per Maskierung die richtigen 32 Bit ermittelt werden, die dann den gesuchten Operanden ausmachen.

Um diesen Nachteil dem Programmierer zu ersparen, erzwingen moderne Prozessoren die Ausrichtung auf Wortgrenzen.

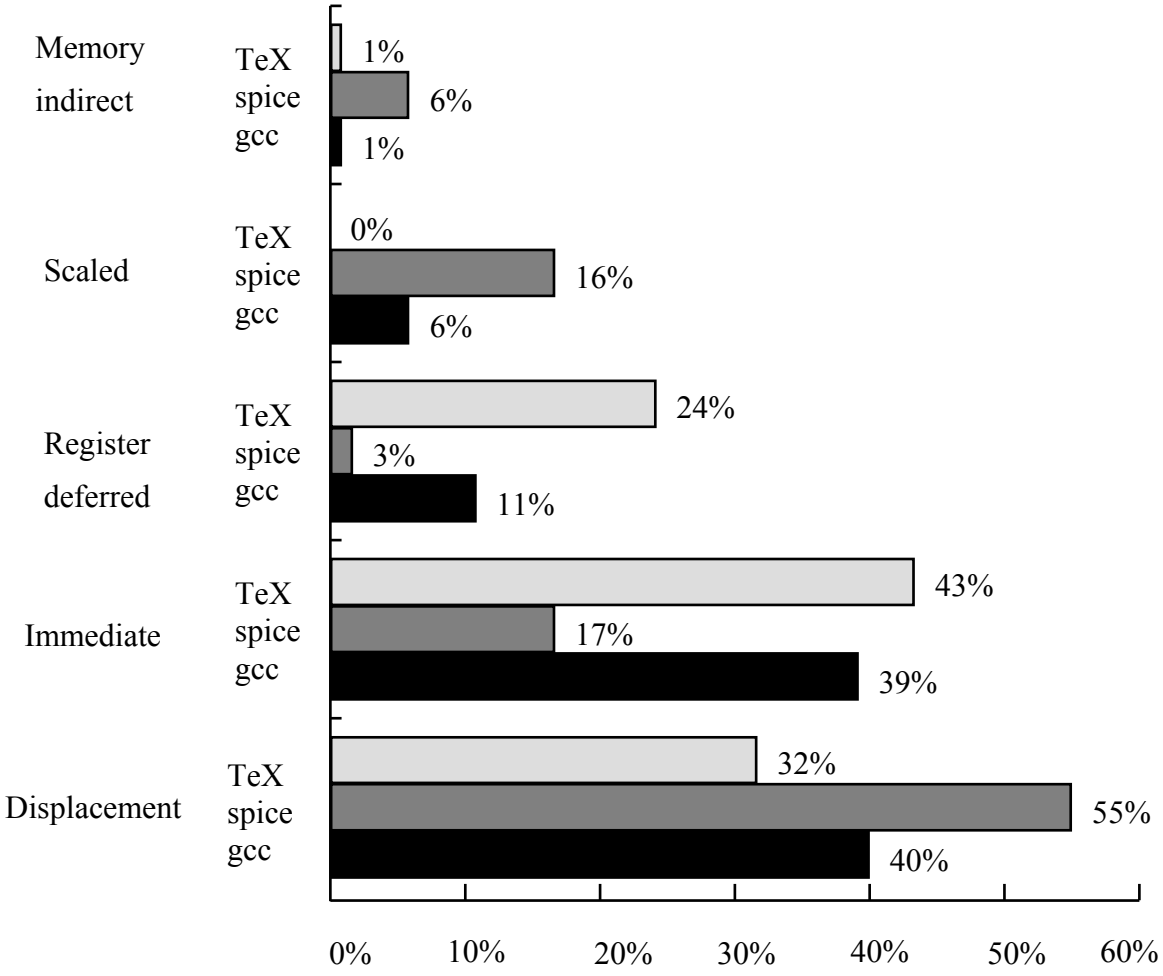
### 7.3.4. Adressierungsarten

Wir wissen jetzt, welche Bytes wir bekommen, wenn wir eine Adresse im Speicher zugreifen. Dieser Abschnitt soll die Möglichkeiten aufzeigen, wie Speicheradressen im Befehl anzugeben sind.

**Definition:** Die *effektive Adresse* ist die aktuelle Speicheradresse, die durch die jeweilige Adressierungsart angesprochen wird.

Die folgende Tabelle zeigt die in heutigen Rechnern verwendeten Adressierungsarten.

| **Adressierungsart** | **Beispiel** | **Bedeutung** | **Anwendung** |
|---|---|---|---|
| Register | Add R4 , R3 | Regs [R4] ←Regs [R4]+Regs[R3] | Wert ist im Register. |
| Immediate | Add R4 , #3 | Regs [R4] ←Regs [R4]+3 | Operand ist eine Konstante |
| Displacement | Add R4 , 100(R1) | Regs [R4] ←Regs [R4]+Mem [100+Regs[R1]] | Lokale Variable |
| Register deferred or indirect | Add R4 , (R1) | Regs [R4] ←Regs [R4]+Mem [Regs[R1]] | Register dient als Pointer. |
| Direct or absolute | Add R1 , (1001) | Regs [R1] ←Regs [R1]+Mem [1001] | Manchmal nützlich für Zugriff auf statische Daten |
| Indexed | Add R3 , (R1 + R2) | Regs [R3] ←Regs [R3]+Mem [Regs[R1]+Regs[R2]] | Nützlich für array-Addressierung: R1=base of array; R2=index amount |
| Memory indirect | Add R1 , @(R3) | Regs [R1] ←Regs [R1]+Mem[Mem[Regs[R3]]] | Wenn R3 die Adresse eines Pointers *p* enthält, dann bekommen wir **p*. |
| Autoincrement | Add R1 , (R2)+ | Regs [R1] ←Regs [R1]+ Mem[Regs[R2]]Regs [R2] | Nützlich für arrays mit Schleifen. R2 ziegt auf den array-Anfang; jeder Zugriff erhöht R2 um die Größe d eines array Elements |
| Autodecrement | Add R1 , -(R2) | Regs [R2] ←Regs [R2]-*d*<br>Regs [R1] ←Regs [R1]+<br>Mem[Regs [R2] ] | Genauso wie Autoincrement |
| Scaled | Add R1 , 100 (R2)[R3] | Regs [R1] ←Regs [R1]+ Mem[100 + Regs [R2] + Regs | Indizierung von Feldern mit Datentypen der Länge d |

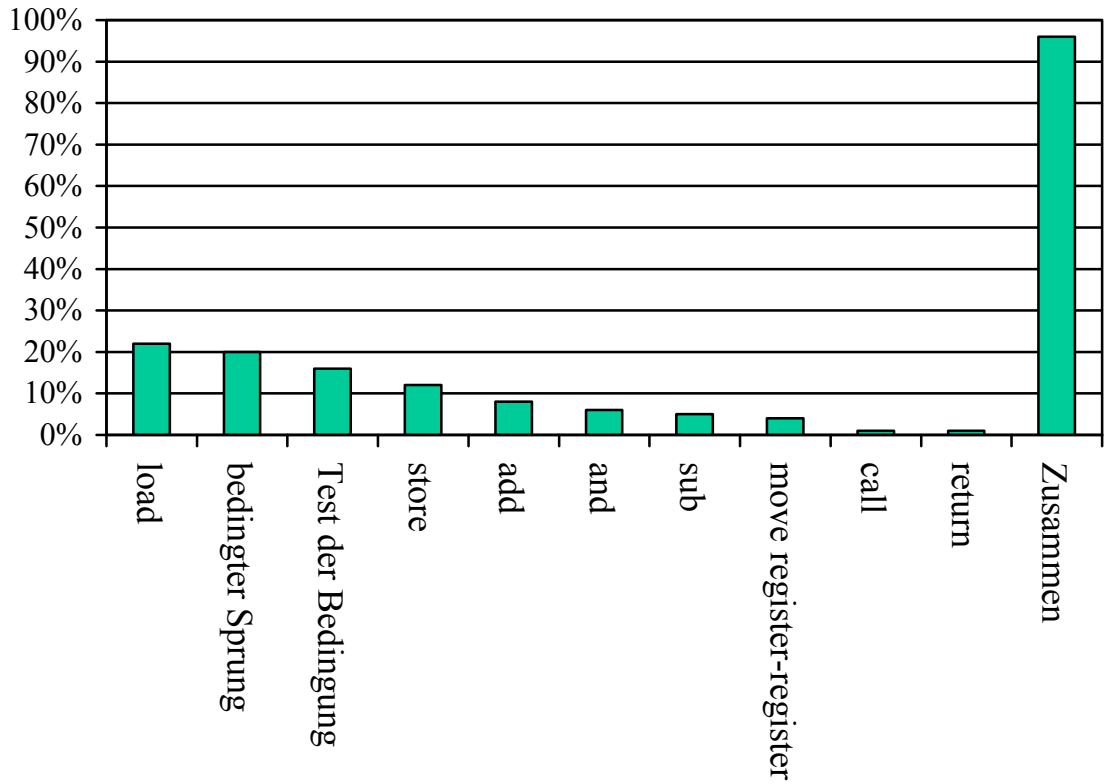



Häufigkeit der Adressierungsarten im Spec 92 Benchmark

**Operationen in einem Befehlssatz**

Die folgende Tabelle zeigt die Verteilung der 10 häufigsten Instruktionen im SPECint92 gemessen an IBM-kompatiblen Maschinen.

**Befehlshäufigkeit:**




**Faustregel:**

Die am häufigsten genutzten Befehle sind die einfachsten Befehle eines Befehlssatzes!

### 7.3.5. Sprungbefehle

Es gibt vier prinzipiell verschiedene Sprungbefehle:

| | |
|---|---|
| **Conditional branch** | **Bedingte Verzweigung** |
| **Jumps** | **Unbedingter Sprung** |
| **Procedure Calls** | **Aufruf** |
| **Procedure Returns** | **Rückkehr aus Prozeduren** |

### 7.3.6. Sprungbefehle

Die Zieladresse für Sprünge muss im Befehl codiert sein. (Ausnahme Returns, weil da das Ziel zur Compilezeit noch nicht bekannt ist). Die einfachste Art ist ein Displacement (offset), das zum **PC (Program Counter)** dazuaddiert wird. Solche Sprünge werden **PC-relativ** genannt.

PC relative Branches und Jumps haben den Vorteil, dass das Sprungziel meist in der Nähe des gegenwärtigen PC Inhalts ist, und deshalb das Displacement nur wenige Bits braucht.

Außerdem hat die Technik, Sprungziele PC-relativ anzugeben, den Vorteil, dass das Programm ablauffähig ist, egal wohin es geladen wird. Man nennt diese Eigenschaft **position independence**.

Für die PC-relative Adressierung ist es sinnvoll, die Länge des erforderlichen Displacements zu kennen.

### 7.3.7. Zusammenfassung:

Wichtigste message ist: **Die einfachsten Operationen sind die wichtigsten. Sie müssen schnell sein.**

**Load, store, add, subtract, move register register, and, shift, compare equal, compare not equal, branch, jump, call und return.**

Die **Sprungentfernung** von der aktuellen Stelle sollte mindestens **8 Bit**, (12 Bit erreicht bereits fast 100% der Fälle) codierbar sein.

Wir brauchen **PC-relative und Register-indirekte Adressierung für Sprünge**.

### 7.3.8. Befehlsformate und Codierung

Für die Länge der Befehlsworte ist die Anzahl der Register und die Adressierungsarten entscheidend. Entscheidender als die Länge des opcodes, da sie öfter in einem Befehl auftauchen können.

Der Architekt muss zwischen folgenden Zielen balancieren:

1. So viele Register und so viele Adressierungsarten wie möglich zu haben.
2. Das Befehlsformat so kompakt wie möglich zu machen.
3. Die Längen der Befehlsformate einfach zugreifbar zu machen.

Letzteres bedeutet insbesondere: Vielfaches eines Bytes. Moderne Architekturen entscheiden sich in der Regel für ein festes Befehlsformat, wobei Einfachheit für die Implementierung gewonnen wird, aber das Optimum an durchschnittlicher Codelänge nicht erreicht wird.

Ein festes Befehlsformat bedeutet nur wenige Adressierungsmodi.

Die Länge der 80x86-Befehle können von 1 bis 5 Byte variieren.

Die der MIPS, Alpha, i860, PowerPC sind fest 4 Byte.

### 7.3.9. Zusammenfassung Codierung

Da wir an der Optimierung in Hinsicht auf Performance interessiert sind, (und nicht vorrangig an Optimierung in Hinsicht auf Code-Dichte, entscheiden wir uns für ein festes Instruktionsformat mit 32 Bit.

## *7.4. Vorgaben DLX*

- GPR-Architektur, load-store
- Adressierung: Displacement, Immediate, Indirect
- schnelle einfache Befehle (load, store, add, ...)
- 8-Bit, 16-Bit, 32-Bit Integer
- feste Länge des Befehlsformats, wenige Formate
- Mindestens 16 GPRs

### 7.4.1. Register

Der Prozessor hat **32 GPRs.**

**Jedes Register ist 32-Bit lang.**

**Sie werden mit R0,..,R31 bezeichnet.**

**R0 hat den Wert 0 und ist nicht beschreibbar** (Schreiben auf R0 bewirkt nichts)

**R31 übernimmt die Rücksprungadresse bei Jump and Link-Sprüngen**

### 7.4.2. Datentypen

**8-Bit Bytes.**

**16-Bit Halbworte.**

**32-Bit Worte.**

**Für Integers. All diese entweder als unsigned Integer oder im 2-er Komplement.**

Laden von Bytes und Halbworten kann wahlweise mit führenden Nullen (unsigned) oder mit Replikation der Vorzeichenstelle (2-er Komplement) geschehen.

Adressierungsarten

**Displacement und Immediate**

**Durch geschickte Benutzung von R0 und 0 können damit vier Adressierungsarten realisiert werden:**

**Displacement:** **Load R1, 1000(R2);**

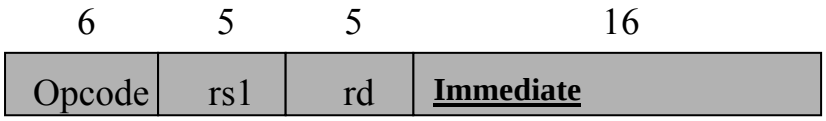
**Immeditate:** **Load R1, #1000;**

**Indirect:** **Load R1, 0(R2);**

**Direct:** **Load R1, 1000(R0);**

### 7.4.3. Befehlsformate:

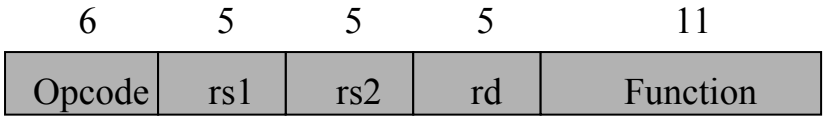
I - Befehl



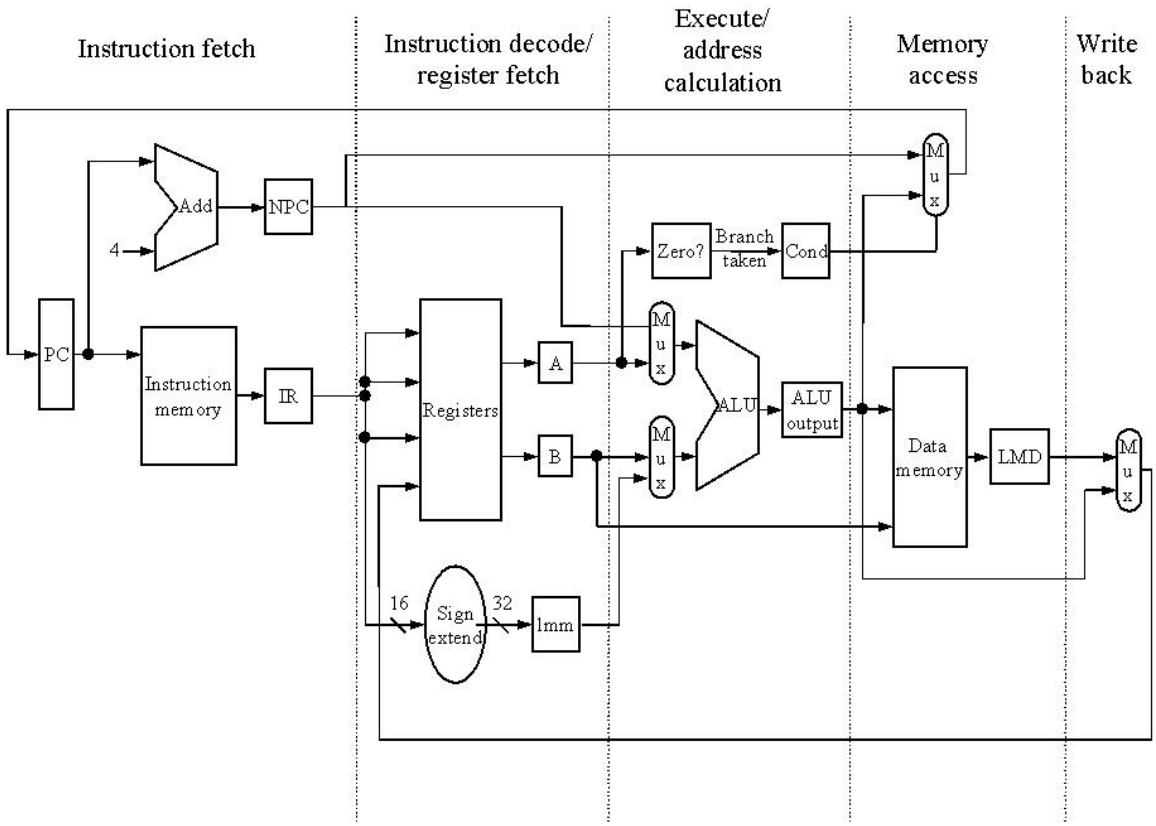

Loads und Stores von Bytes, Worten, Halbworten
Alle Immediate-Befehle (rd ← rs1 op immediate)

Bedingte Verzweigungen (rs1 : register, rd unbenutzt)
Jump register, Jump and link register
(rd = 0, rs1 = destination, immediate = 0)

R - Befehl




Register-Register ALU Operationen: rd ← rs1 func rs2
Function sagt, was gemacht werden soll: Add, Sub, ...
Read/write auf Spezialregistern und moves




Die Syntax der PC-relativen Sprungbefehle ist etwas irreführend, da der als Operand eingegebene Parameter als Displacement zum PC zu verstehen ist.

Tatsächlich müsste die erste Zeile heißen:

**J offset bedeutet PC <--- PC+4 + offset mit $-2^{15}$ <= offset < $+2^{15}$**

Das würde aber heißen, dass man in Assemblerprogrammen die Displacements bei relativen Adressen explizit angeben muss. Dies macht die Wartung eines solchen Programms unglaublich schwierig. Daher erlaubt man, dass man Namen für die effektiven Adressen einführt, die man wie Marken ins Assemblerprogramm schreibt. Ein Compiler verwendet natürlich die tatsächlichen Offsets, aber für den Leser ist eine solches mit Marken geschrieben Programm leichter verständlich.

| Instruction type/opcode | Instruction meaning |
|---|---|
| **Data transfers** | **Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR** |
| LB,LBU,SB | Load byte, load byte unsigned, store byte |
| LH, LHU, SH | Load half word, load half word unsigned, store half word |
| LW, SW | Load word, store word (to/from integer registers) |
| **Arithmetic/logical** | **Operations on integer or logical data in GPRs; signed arithmetic trap on overflow** |
| ADD, ADDI, ADDU, ADDUI | Add, add immediate (all immediates are 16 bits); signed and unsigned |
| SUB, SUBI, SUBU, SUBUI | Subtract, subtract immediate; signed and unsigned |
| MULT,MULTU | Multiply, signed and unsigned; all operations take and yield 32-bit values |
| AND,ANDI | And, and immediate |
| OR,ORI,XOR,XORI<br>LHI | Or, or immediate, exclusive or, exclusive or immediate |
| LHI | Load high immediate – loads upper half of register with immediate |
| SLL, SRL, SRA, SLLI, SRLI, SRAI | Shifts: both immediate (S_l) and variable form (S_); shifts are shift left logical, right logical, right arithmetic |
| S_, S_ I | Set conditional: ”_ ” may be LT, GT, LE, GE, EQ, NE |
| **Control** | **Conditional branches and jumps; PC-relative or through register** |
| BEQZ,BNEZ | Branch GPR equal/not equal to zero; 16-bit offset from PC+4 |
| J, JR | Jumps: 16-bit offset from PC+4 (J) or target in register (JR) |
| JAL, JALR | Jump and link: save PC+4 in R31, target is PC-relative (JAL) or a register (JALR) |
| TRAP | Transfer to operating system at a vectored address |
| RFE | Return to user code from an exception; restore user mode |

**Beispiel:**

Wir wollen zwei 100-elementige Vektoren A und B miteinander Multiplizieren, wobei A und B aus doppeltgenauen Gleitkommazahlen bestehen und A an der Adresse 5000 steht und B an der Adresse 5800 steht. Das Ergebnis soll an die Stelle 6600 gespeichert werden. Das heißt, wir wollen folgenden Algorithmus implementieren:

```
C = 0;
for (i=0; i<100; i++)
  { C = C + a[i]*b[i]; }
```

**Wie sieht das als Assemblerprogramm aus?**

```
        SUBD F0, F0, F0          //    F0 = 0; Variable für C
        ADDI R3, R0, #5000       //    R3 = 5000;
        ADDI R4, R0, #5800       //    R4 = 5800;
        ADD  R5, R0, R4          //    R5 = 5800;
Loop    LD   F2, 0(R3)           //    Lade A[i]
        LD   F4, 0(R4)           //    Lade B[i]
        MULTD F6, F2, F4         //    A[i] * B[i]
        ADDDF0, F0, F6           //    Aufaddieren auf C
        ADDI R3, R3, #8          //    Hochzählen von i
        ADDI R4, R4, #8          //    Hochzählen von i
        SEQ  R6, R3, R5          //    Abbruchbedingung
        BEQZ R6, Loop            //    Schleife
        SD   6600(R0), F0        //    Speichern des Ergebnisses
```

```
function Fakultät(n:integer): integer;
var k:integer;
begin
if n=0 then Fakultät := 1
else
  begin
  k:=n;
  n:=n-1;
```

```
  Fakultät := k*Fakultät(n)
  end
end;
```

R1: n
R2: Stackpointer
R3: k
R4: Fakultät
R31: Rücksprungadresse

**Hauptprogramm:**

```
            ADDI  R1, R0, #2
            JAL   Fakultät          // R1 Fakultät wird in R4 berechnet
rück1       SW    2000(R0), R4

            BNEZ  R1, weiter        // Wenn R1=0 ist, soll R4 auf 1
            ADDI  R4, R0, #1        // gesetzt werden.
            J     return            // danach Rücksprung
weiter      ADD   R3, R1, R0        // k := n
            SUBI  R1, R1, #1        // n := n-1
            JAL   Fakultät          // rekursiver Aufruf
rück2       MULT  R4, R3, R4        // Fakultät := Fakultät* k
Fakultät    ADDI  R2, R2, #4        //
            SW    1000(R2), R1      //   Retten der Register
            ADDI  R2, R2, #4        //   auf den Stack
            SW    1000(R2), R3      //
            ADDI  R2, R2, #4        //
            SW    1000(R2), R31     //

Return LW   R31, 1000(R2)           //
            SUBI  R2, R2, #4        //
            LW    R3, 1000(R2)      //   Zurückholen der
            SUBI  R2, R2, #4        //   Register vom Stack
```

```
            LW    R1, 1000(R2)      //
            SUBI  R2, R2, #4        //
            JR    R31               //  Rücksprung nach rück2

Fakultät    ADDI  R2, R2, #4        //
            SW    1000(R2), R1      //  Retten der Register
            ADDI  R2, R2, #4        //  auf den Stack
            SW    1000(R2), R3      //
            ADDI  R2, R2, #4        //
            SW    1000(R2), R31     //

            BNEZ  R1, weiter        //  Wenn R1=0 ist, soll R4 auf 1
            ADDI  R4, R0, #1        //  gesetzt werden.
            J     return            //  danach Rücksprung
weiter      ADD   R3, R1, R0        //  k := n
            SUBI  R1, R1, #1        //  n := n-1
            JAL   Fakultät          //  rekursiver Aufruf
rück2       MULT  R4, R3, R4        //  Fakultät := Fakultät* k

return      LW    R31, 1000(R2)     //
            SUBI  R2, R2, #4        //
            LW    R3, 1000(R2)      //  Zurückholen der
            SUBI  R2, R2, #4        //  Register vom Stack
            LW    R1, 1000(R2)      //
            SUBI  R2, R2, #4        //
            JR    R31               //  Rücksprung
```

# 8. Pipelining

## *8.1. Pipelining : Implementierungstechnik*

Vielfältig angewendet in der Rechnerarchitektur.

Macht CPUs schnell.

Wie Fließbandverarbeitung.

Hintereinanderausführung von **Stufen der Pipeline, pipe stages,** pipe segments.

Vergleich Autoproduktion: Maximiert den Durchsatz, d.h. so viele Autos wie möglich werden pro Stunde produziert. Und das, obwohl die Produktion eines Autos viele Stunden dauern kann.

Hier: **So viele Instruktionen wie möglich sollen in einer Zeiteinheit ausgeführt werden.**

### 8.1.1. Durchsatz

Da die Stufen der Pipeline hintereinander hängen, muss die Weitergabe der Autos (der Instruktionen) synchron also **getaktet** stattfinden. Sonst würde es irgendwo einen Stau geben.

Der Takt für diese Weitergabe wird bei uns der Maschinentakt sein.

**Die Länge des Maschinentaktzyklus ist daher bestimmt von der maximalen Verarbeitungszeit der Einheiten in der Pipeline für eine Berechnung.**

Der Entwerfer der Pipeline sollte daher anstreben, alle Stufen so zu gestalten, dass die Verarbeitung innerhalb der Stufen etwa gleichlang dauert.
Im Idealfall ist die durchschnittliche Ausführungszeit für eine Instruktion auf einer gepipelineten Maschine

$$\frac{\textit{Zeit für die nicht gepipelinete Verarbeitung}}{\textit{Anzahl der Stufen}}$$

**Dadurch ist der speedup durch pipelining höchstens gleich der Anzahl der Stufen.**

Wie bei der Autoproduktion, wo mit einer n-stufigen Pipeline n mal so viele Autos gefertigt werden können.

**Durch Pipelining benötigt man**

- **weniger Maschinentakte für einen Befehl oder**
- **Weniger Zeit für einen Taktzyklus oder**
- **beides**

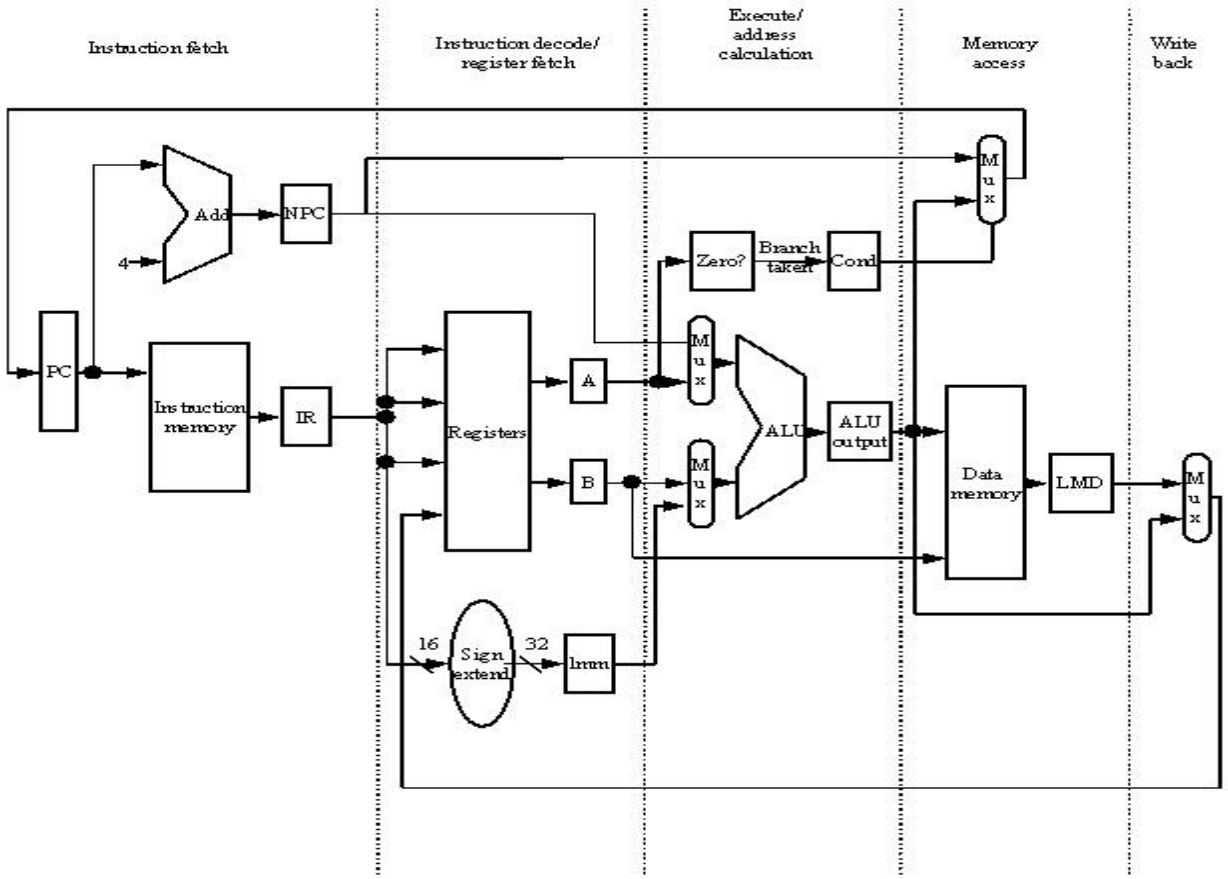
Wenn die Ursprungsmaschine mehrere Takte für die Ausführung einer Instruktion brauchte, gilt ersteres.

Wenn die Ursprungsmaschine einen langen Takt für die Ausführung einer Instruktion brauchte, wird durch Pipelining die Taktzykluszeit verringert.
Pipelining ist **für den Benutzer** des Rechners **unsichtbar.**

### 8.1.2. Die DLX-Pipeline

Wir werden die Probleme des Pipelining an der DLX studieren, weil sie einfach ist und alle wesentlichen Merkmale eines modernen Prozessors aufweist. Wir kennen bereits eine nicht gepipelinete Version der DLX aus:

- nicht optimal, aber so, dass sie sich leicht in eine gepipelinete Version umbauen lässt.
- die Ausführung jeder Instruktion dauert 5 Taktzyklen



1. Instruction Fetch Zyklus: (IF):

**IR <-- Mem[PC]**

**NPC <-- PC + 4**

Hole den Wert aus dem Speicher, wobei die Adresse im gegenwärtigen PC ist und speichere ihn im IR (Instruction-Register). Erhöhe den PC um 4, denn dort steht die nächste Instruktion. Speichere diesen Wert in NPC (neuer PC).

2. Instruction decode / Register fetch Zyklus (ID):

**A <-- Regs[$IR_{6..10}$]**

**B <-- Regs[$IR_{11..15}$]**

**IMM <-- $((IR_{16})^{16}$##$IR_{16..31}$**

Dekodiere die Instruktion in IR und hole die Werte aus den adressierten Registern. Die Werte der Register werden zunächst temporär in A und B gelatcht für die Benutzung in den folgenden Taktzyklen. Die unteren 16 Bit von IR werden sign-extended und ebenso im temporären Register IMM gespeichert.

Dekodierung und Register lesen werden in einem Zyklus durchgeführt. Das ist möglich, weil die Adressbits ja an festen Positionen stehen 6..10, 11..15, 16..31. Es kann sein, daß wir ein Regsiter lesen, das wir gar nicht brauchen. Das schadet aber nichts, es wird sonst einfach nicht benutzt.

Der Immediate Operand wird um 16 Kopien des Vorzeichenbits ergänzt und ebenfalls gelesen, falls er im nachfolgenden Takt gebraucht wird.

3. Execution / effective adress Zyklus (EX):

Hier können vier verschiedene Operationen ausgeführt werden, abhängig vom DLX-Befehlstyp:

**Befehl mit Speicherzugriff (load/store):**

**ALUoutput <-- A + IMM**

Die effektive Adresse wird ausgerechnet und im temporären Register ALUoutput gespeichert.

**Register-Register ALU-Befehl:**

**ALUoutput <-- A func B**

Die ALU wendet die Operation func (die in Bits 0..5 und 22..32 des Opcodes spezifiziert ist) auf A und B an und speichert das Ergebnis im temporären Register ALUoutput.

**Register-Immediate ALU-Befehl:**

**ALUoutput <-- A op IMM**

Die ALU wendet die Operation op (die in Bits 0..5 des Opcodes spezifiziert ist) auf A und B an und speichert das Ergebnis im temporären Register ALUoutput.

**Verzweigungs-Befehl:**

**ALUoutput <-- NPC + IMM**

**Cond <-- (A op 0)**

Die ALU berechnet die Adresse des Sprungziels relativ zum PC. Cond ist ein temporäres Register, in dem das Ergebnis der Bedingung gespeichert wird. Op ist im Opcode spezifiziert und ist z.B. gleich bei BEQZ oder ungleich bei BNEZ.

Wegen der load/store Architektur kommt es nie vor, dass gleichzeitig eine Speicheradresse für einen Datenzugriff berechnet und eine arithmetische Operation ausgeführt werden muss. Daher sind die Phasen Execution und Effective Adress im selben Zyklus möglich.

4. Memory access / branch completion Zyklus (MEM):

Die in diesem Zyklus aktiven Befehle sind load, store und branches:

**Lade Befehl (load):**

**LMD <-- Mem[ALUoutput]**

Das Wort, adressiert mit ALUoutput wird ins temporäre Register LMD geschrieben.

**Speicher Befehl (store):**

**Mem[ALUoutput] <-- B**

Der Wert aus B wird im Speicher unter der Adresse in ALUoutput gespeichert.

**Verzweigungs Befehl (branch):**

**if cond then**

**PC <-- ALUoutput**

**else PC <-- NPC**

Wenn die Verzweigung ausgeführt wird, wird der PC auf das Sprungziel, das in ALUoutput gespeichert ist gesetzt, sonst auf den NPC. Ob verzweigt wird, ist im Execute Zyklus nach cond geschrieben worden.

5. Write back Zyklus (WB):

**Register-Register ALU Befehl:**

**Regs[$IR_{16..20}$] <-- ALUoutput**

**Register-Immediate ALU Befehl:**

**Regs[$IR_{11..15}$] <-- ALUoutput**

**Lade Befehl:**

**Regs[$IR_{11..15}$] <-- LMD**

Das Ergebnis wird ins Register geschrieben. Es kommt entweder aus dem ALUoutput oder aus dem Speicher (LMD). Das Zielregister kann an zwei Stellen im Befehl codiert sein, abhängig vom Opcode.

**Erklärung zum Datenpfad-Bild**

Am Anfang oder am Ende jedes Zyklus steht jedes Ergebnis in einem Speicher, entweder einem GPR oder in einer Hauptspeicherstelle oder in einem temporären Register (LMD, IMM, A, B, NPC, ALUoutput, Cond). Die temporären Register halten ihre Werte nur

während der aktuellen Instruktion, während alle anderen Speicher sichtbare Teile des Prozessorzustands sind.
Wir können die Maschine jetzt so umbauen, daß wir fast ohne Veränderung in jedem Takt die Ausführung einer Instruktion beginnen können. Das wird mit Pipelining bezeichnet. Das resultiert in einem Ausführungsmuster wie auf der folgenden Folie

### *8.2. Ausführung der Befehle in der Pipeline*

| **Befehl** | **Takt** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
| ***Befehl i*** | **IF** | **ID** | **EX** | **MEM** | **WB** | | | | |
| ***Befehl i+1*** | | **IF** | **ID** | **EX** | **MEM** | **WB** | | | |
| ***Befehl i+2*** | | | **IF** | **ID** | **EX** | **MEM** | **WB** | | |
| ***Befehl i+3*** | | | | **IF** | **ID** | **EX** | **MEM** | **WB** | |
| ***Befehl i+4*** | | | | | **IF** | **ID** | **EX** | **MEM** | **WB** |

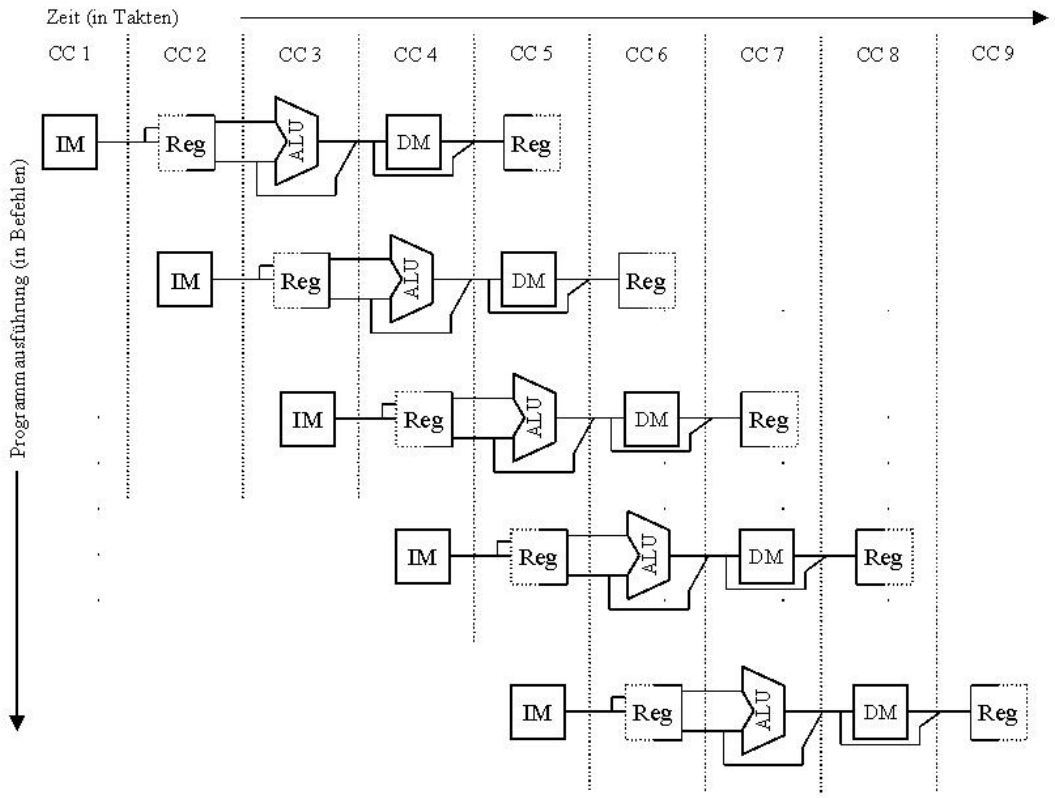
**Pipelining ist nicht so einfach wie es hier zunächst erscheint. Im folgenden wollen wir die Probleme behandeln, die mit Pipelining verbunden sind.**

Unterschiedliche Operationen müssen zum Zeitpunkt i unterschiedliche Teile der Hardware nutzen. Um das zu überprüfen, benutzen wir eine vereinfachte Darstellung des DLX-Datenpfades, den wir entsprechend der Pipeline zu sich selbst verschieben.

An vier Stellen tauchen Schwierigkeiten auf:

1. **Im IF Zyklus und im MEM Zyklus wird auf den Speicher zugegriffen**. Wäre dies physikalisch derselbe Speicher, so könnten nicht in einem Taktzyklus beide Zugriffe stattfinden. Daher verwenden wir zwei verschieden Caches, einen **Daten-Cache**, auf den im **MEM Zyklus** zugegriffen wird und einen **Befehls-Cache**, den wir im **IF-Zyklus** benutzen. Nebenbei: der Speicher muss in der gepipelineten Version fünf mal so viele Daten liefern wie in der einfachen Version. Der dadurch entstehende Engpass zum Speicher ist der Preis für die höhere Performance.

2. **Die Register werden im ID und im WB-Zyklus benutzt**.




3. **PC.** In der vereinfachten Darstellung des Datenpfades haben wir den PC nicht drin.

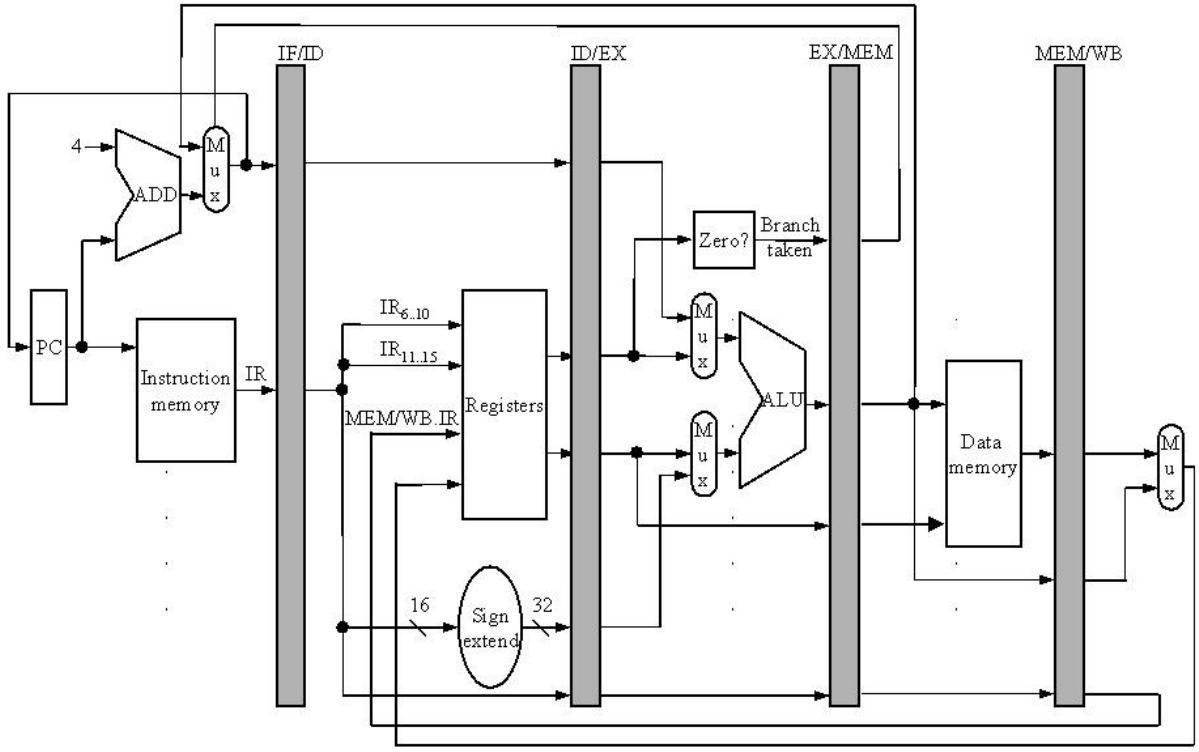
   **PC muss in jedem Takt inkrementiert werden, und zwar in der IF-Phase.**

4. **Problem Verzweigung** verändert PC aber erst in MEM-Phase. Aber das Inkrementieren passiert bereits in IF-Phase. Dies erfordert eine spezielle Behandlung von Verzweigungsbefehlen, die wir hier nicht behandeln wollen. Wir sagen stattdessen: Wenn dieser Fall eintritt, muss die Pipeline für einige Takte unterbrochen (gestaut) werden.

**Folgende Voraussetzungen müssen für einen reibungslosen Ablauf gegeben sein:**

- **Jede Stufe ist in jedem Takt aktiv.**
- **Jede Kombination von gleichzeitig aktiven Stufen muss möglich sein.**
- **Werte, die zwischen zwei Stufen der Pipeline weitergereicht werden, müssen in Registern gespeichert werden (gelatcht).**

Die Register sind auf der nächsten Folie zu sehen. Sie werden mit den Stufen bezeichnet, zwischen denen sie liegen.




Alle temporären Register aus unserem ersten Entwurf können jetzt in diesen Registern mit aufgenommen werden.

Die Pipeline-Register halten aber Daten und Kontrollinformation.

**Beispiel:** Die IR-Information muss mit den Stufen der Pipeline weiterwandern, damit zum Beispiel in der WB-Phase das Ergebnis in das richtige Register geschrieben wird, das zu dem alten Befehl gehört.
Jeder Schaltvorgang passiert in einem Takt, wobei die Eingaben aus dem Register vor der entsprechenden Phase genommen werden und die Ausgaben in die Register nach der Phase geschrieben werden.

Die folgende Folie zeigt die Aktivitäten in den einzelnen Phasen unter dieser Sicht.

| Stufe | Was wird alles getan | | |
|---|---|---|---|
| IF | IF/ID.IR ← Mem[PC];<br>IF/ID.NPC,PC ← (if EX/MEM.cond {EX/MEM.ALU Output} else {PC+4}); | | |
| ID | ID/EX.A ← Regs[IF/ID.$IR_{6..10}$]; ID/EX.B ← Regs[IF/ID.$IR_{11..15}$];<br>ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR;<br>ID/EX.Imm ← $(IR_{16})^{16}$ ## $IR_{16..31}$; | | |
| | **ALU Befehl** | **Load oder store Befehl** | **Verzweigungsbefehl** |
| EX | EX/MEM.IR ← ID/EX.IR;<br>EX/MEM. ALUOutput ←<br>ID/EX. A func ID/EX. B;<br>or<br>EX/MEM.ALUOutput ←<br>ID/EX.A *op* ID/EX.Imm;<br>EX/MEM.cond ← 0; | EX/MEM.IR ← ID/EX.IR<br>EX/MEM.ALUOutput ←<br>ID/EX.A + ID/EX.Imm;<br><br><br><br>EX/MEM.cond ← 0;<br>EX/MEM.B ← ID/EX.B; | <br><br>EX/MEM.ALUOutput ←<br>ID/EX.NPC+ID.EX.Imm;<br><br><br><br>EX/MEM.cond ←<br>(ID/EX.A *op* 0); |
| MEM | MEM/WB.IR ← EX/MEM.IR;<br>MEM/WB.ALUOutput ←<br>EX/MEM.ALUOutput; | MEM/WB.IR ← EX/MEM.IR;<br>MEM/WB.LMD<br>Mem[EX/MEM.ALUOutput];<br>or<br>Mem[EX/MEM.ALUOutput] ←<br>EX/MEM.B; | |
| WB | Regs [MEM/WB. $IR_{16..20}$] ←<br>MEM/WB.ALUOutput;<br>or<br>Regs[MEM/WB.$IR_{11..15}$] ←<br>MEM/WB.ALUOutput; | Regs[MEM/WB.$IR_{11..15}$] ←<br>MEM/WB.LMD; | |

Die Aktivitäten in den ersten zwei Stufen sind nicht befehlsabhängig. Das muss auch so sein, weil wir den Befehl ja erst am Ende der zweiten Stufe interpretieren können.

Durch die festen Bit-Positionen der Operandenregister im IR-Feld ist die Dekodierung und das Register-Lesen in einer Phase möglich.

Um den Ablauf in dieser einfachen Pipeline zu steuern, ist die Steuerung der vier Multiplexer in dem Diagramm erforderlich:

**oberer ALU-input-Mux:** **Verzweigung oder nicht**

**unterer ALU-input-Mux:** **Register-Register-Befehl oder nicht**

**IF-Mux:** **EX/MEM.cond**

**WB-Mux:** **load oder ALU-Operation**

Es gibt einen fünften (nicht eingezeichneten Mux), der beim WB auswählt, wo im MEM/WB.IR die Adresse des Zielregisters steht, nämlich an Bits 16..20 bei einem Register-Register-ALU-Befehl und an Bits 11..15 bei einem Immediate- oder Load-Befehl.

### 8.2.1. Performance Verbesserung durch Pipelining

**Durchsatz wird verbessert - nicht Ausführungszeit einer Instruktion**

im Gegenteil, durch den **Pipeline-overhead**, die Tatsache, dass die Stufen **nie perfekt ausbalanciert** (Takt wird bestimmt durch die langsamste Stufe) sind und die zusätzlichen **Latche mit ihren Setup-Zeiten und Schaltzeiten** wird die **einzelne Instruktion meist langsamer**.

Aber insgesamt: **Programme laufen schneller, obwohl keine einzige Instruktion individuell schneller läuft.**

**Beispiel:** Ungepipelinete DLX: vier Zyklen für ALU- , branch- oder store-Operationen, fünf für load-Operationen. ALU 30%, branch 20%, store 10%, load 40%. 200 MHz Takt. Angenommen, durch den Pipeline-Overhead brauchen wir eine ns mehr pro Stufe. Welchen speedup erhalten wir durch die Pipeline?

**Durchschnittliche Ausführungszeit für einen Befehl =**

Zykluszeit * durchschnittliche Anzahl Zyklen pro Befehl =

5ns * ((30%+20%+10%)*4 + 40%*5) =

22 ns.

**In der Pipeline-Version muss der Takt mit der Zykluszeit der langsamsten Stufe laufen plus Overhead. Also 5ns+1ns.**

$$speedup = \frac{durchschnittliche\ ungepipelinete\ Ausführungszeit}{durchschnittliche\ gepipelinete\ Ausführungszeit}$$

$$= \frac{22ns}{6ns} = 3{,}67$$

Soweit würde die Pipeline gut für paarweise unabhängige Integer-Befehle funktionieren. In der Realität hängen Befehle aber voneinander ab. Dieses Problem werden wir im folgenden behandeln:

### 8.2.2. Pipeline Hazards

Es gibt Fälle, in denen die Ausführung einer Instruktion in der Pipeline nicht in dem für sie ursprünglichen Takt möglich ist. Diese werden **Hazards** genannt. Es gibt drei Typen:

1. **Strukturhazards** treten auf, wenn die Hardware die Kombination zweier Operationen, die gleichzeitig laufen sollen, nicht ausführen kann. Beispiel: Schreiben in den Speicher gleichzeitig mit IF bei nur einem Speicher.
2. **Datenhazards** treten auf wenn das Ergebnis einer Operation der Operand einer Nachfolgenden Operation ist, dies Ergebnis aber nicht rechtzeitig vorliegt.
3. **Steuerungshazards, Control hazards** treten auf bei Verzweigungen in der Pipeline oder anderen Operationen, die den PC verändern.

Hazards führen zu eine **Stau (stall)** der Pipeline. Pipeline-staus sind aufwendiger als z.B. Cache-miss-Staus, denn einige Operationen in der Pipe müssen weiterlaufen, andere müssen angehalten werden.

In der Pipeline müssen alle Instruktionen, die schon länger in der Pipeline sind als die gestaute, weiterlaufen, währen alle jüngeren ebenfalls gestaut werden müssen. Würden die älteren nicht weiterverarbeitet, so würde der Stau sich nicht abbauen lassen.

Als Folge werden keine neuen Instruktionen gefetcht, solange der Stau dauert.

### 8.2.3. Struktur Hazards

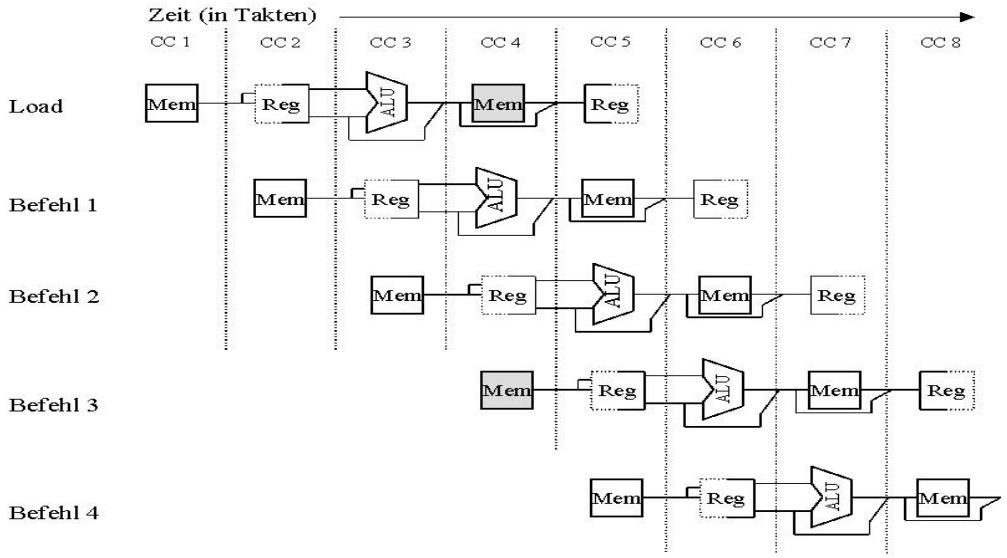
Die überlappende Arbeit an allen Phasen der Pipeline gleichzeitig erfordert, dass alle Resourcen oft genug vorhanden sind, so dass alle Kombinationen von Aufgaben in unterschiedlichen Stufen der Pipeline gleichzeitig vorkommen können. Sonst bekommen wir einen Struktur Hazard.

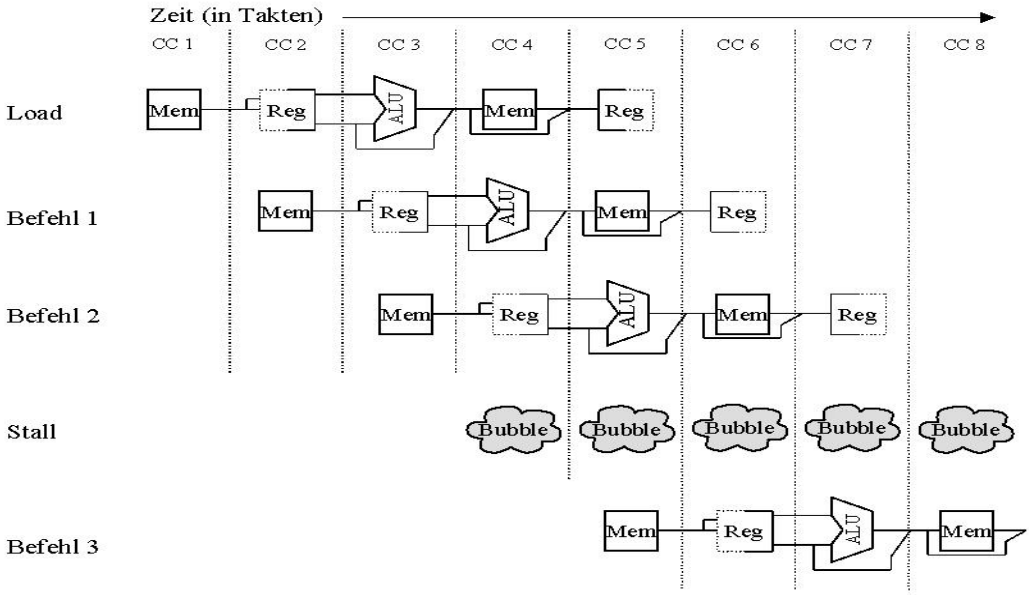
Typischer Fall: Eine Einheit ist nicht voll gepipelined: Dann können die folgenden Instruktionen, die diese Einheit nutzen, nicht mit der Geschwindigkeit 1 pro Takt abgearbeitet werden.

Ein anderer häufiger Fall ist der, dass z.B. der Registerfile nur einen Schreibvorgang pro Takt erlaubt, aber zwei aufeinanderfolgende Befehle in unterschiedlichen Phasen beide ein writeback in ein Register ausführen wollen.

Wenn ein Struktur Hazard erkannt wird, staut die Pipeline, was die CPI auf einen Wert > 1 erhöht.

Anderes Beispiel: Eine Maschine hat keinen getrennten Daten- und Instruktions-Cache. Wenn gleichzeitig auf ein Datum und eine Instruktion zugegriffen wird, entsteht ein Struktur Hazard. Die folgende Folie zeigt solch einen Fall. Die leer arbeitende Phase (durch den Stau verursacht) wird allgemein eine pipeline bubble (Blase) genannt, da sie durch die Pipeline wandert, ohne sinnvolle Arbeit zu tragen.






Zur Darstellung der Situation in einer Pipeline ziehen einige Entwerfer eine andere Form des Diagramms vor, das zwar nicht so anschaulich, aber dafür kompakter und genauso aussagefähig ist.

Ein Beispiel dafür zeigt die folgende Folie.

| Befehl | Takt 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| **Load Befehl** | **IF** | **ID** | **EX** | **MEM** | **WB** | | | | | |
| ***Befehl i+1*** | | **IF** | **ID** | **EX** | **MEM** | **WB** | | | | |
| ***Befehl i+2*** | | | **IF** | **ID** | **EX** | **MEM** | **WB** | | | |
| ***Befehl i+3*** | | | | **stall** | **IF** | **ID** | **EX** | **MEM** | **WB** | |
| ***Befehl i+4*** | | | | | | **IF** | **ID** | **EX** | **MEM** | **WB** |
| ***Befehl i+5*** | | | | | | | **IF** | **ID** | **EX** | **MEM** |
| ***Befehl i+6*** | | | | | | | | **IF** | **ID** | **EX** |

### 8.2.4. Daten Hazards

Betrachten wir folgendes Assembler-Programmstück

**ADD R1, R2, R3**

**SUB R4, R5, R1**

**AND R6, R1, R7**

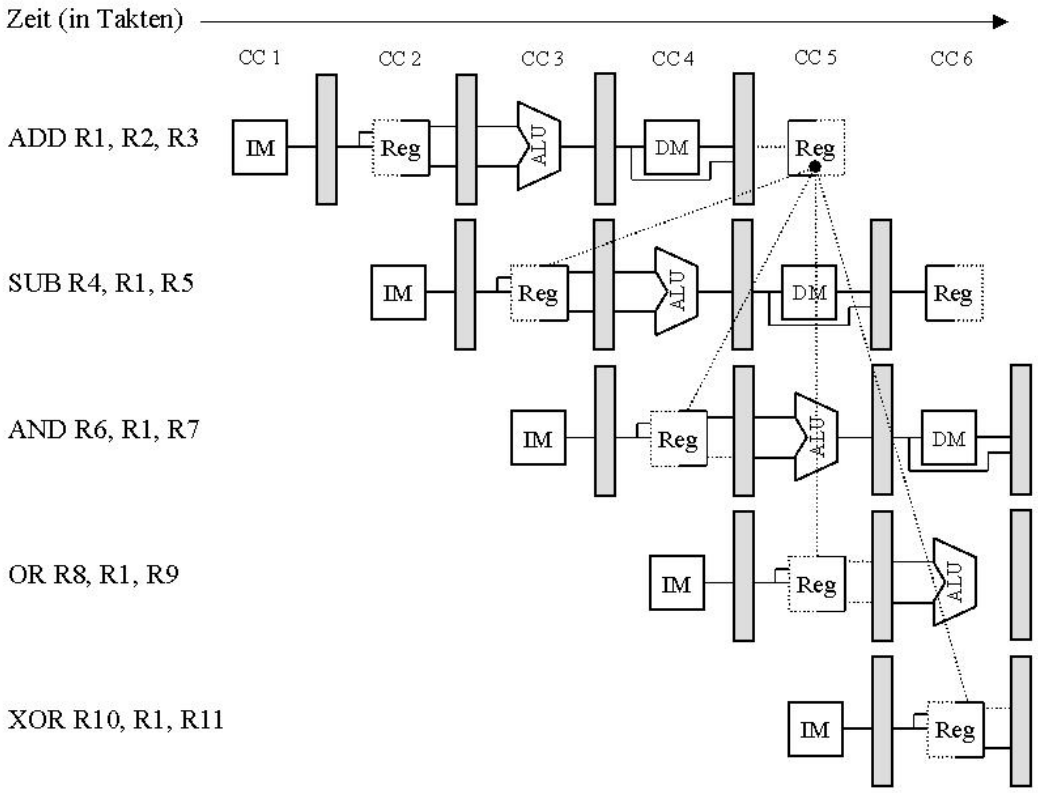
**OR R8, R1, R9**

**XOR R10, R1, R11**

Alle Befehle nach ADD brauchen das Ergebnis von ADD. Wie man auf der folgenden Folie sieht, schreibt ADD das Ergebnis aber erst in seiner WB-Phase nach R1.

Aber SUB liest R1 bereits in deren ID-Phase. Dies nennen wir einen **Daten Hazard.**

Wenn wir nichts dagegen unternehmen, wird SUB den alten Wert von R1 lesen. Oder noch schlimmer, wir wissen nicht, welchen Wert SUB bekommt, denn wenn ein Interrupt dazu führen würde, daß die Pipeline zwischen ADD und SUB unterbrochen würde, so würde ADD noch bis zur WB-Phase ausgeführt werden, und wenn der Prozess neu mit SUB gestartet würde, bekäme SUB sogar den richtigen Operanden aus R1.




Der AND-Befehl leidet genauso unter dem Hazard. Auch er bekommt den falschen Wert aus R1.

Das XOR arbeitet richtig, denn der Lesevorgeng des XOR ist zeitlich nach dem WB des ADD.

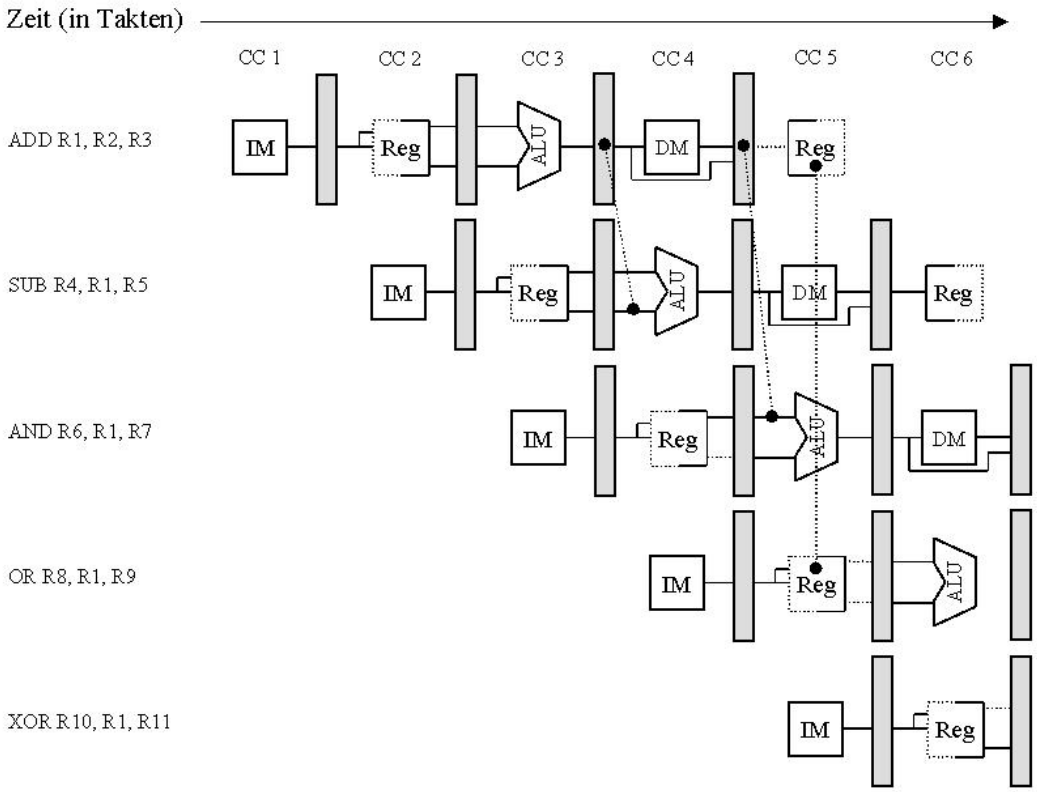
Das OR können wir dadurch sicher machen, dass in der Implementation dafür gesorgt wird, dass jeweils in der ersten Hälfte des Taktes ins Register geschrieben wird und in der zweiten Hälfte gelesen. Dies wird in der Zeichnung durch die halben Kästen angedeutet.

Wie vermeiden wir nun stalls bei SUB und AND?

Das Zauberwort heißt **forwarding**.

Das geht folgendermaßen:

1. **Das Ergebnis der ALU (aus EX/MEM) wird (auch) in den ALU-Eingang zurückgeführt.**
2. **2. Eine spezielle Forwarding Logik, die nur nach solchen Situationen sucht, wählt die zurückgeschrieben Version anstelle des Wertes aus dem Register als tatsächlichen Operanden aus und leitet ihn an die ALU.**




Beim Forwarding ist bemerkenswert: Wenn SUB selbst gestallt wird, benötigt der Befehl nicht die rückgeführte Version des Ergebnisses, sondern kann ganz normal aus R1 zugreifen.

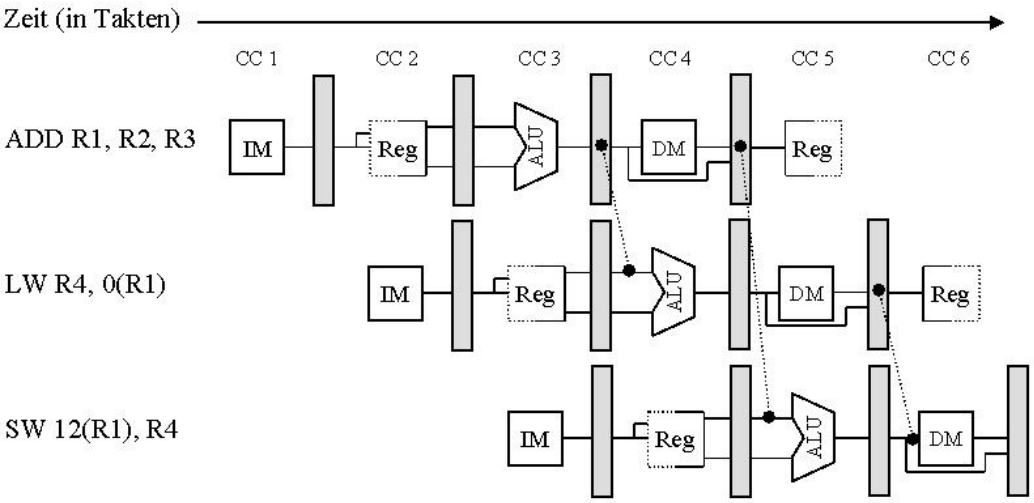
Das Beispiel zeigt auch, dass das neue Ergebnis auch für den übernächsten Befehl (AND) in rückgeführter Form zur Verfügung gehalten werden muss.

**Forwarding ist eine Technik der beschleunigten Weiterleitung von Ergebnissen an die richtige Verarbeitungseinheit, wenn der natürliche Fluss der Pipeline für einen solchen Transport nicht ausreicht.**

Ein anderes Beispiel:

```
ADD   R1, R2, R3
LW    R4, 0(R1)
SW    12(R1), R4
```

Um hier den Daten Hazard zu vermeiden, müssten wir R1 rechtzeitig zum ALU-Eingang bringen und R4 zum Speichereingang. Die folgende Folie zeigt die Forwarding-Pfade, die für dieses Problem erforderlich sind.




### 8.2.5. Daten Hazards, die Staus verursachen müssen

Es gibt Daten Hazards, die nicht durch forwarding zu entschärfen sind:

**LW R1, 0(R2)**

**SUB R4, R1, R3**

**AND R6, R1, R7**

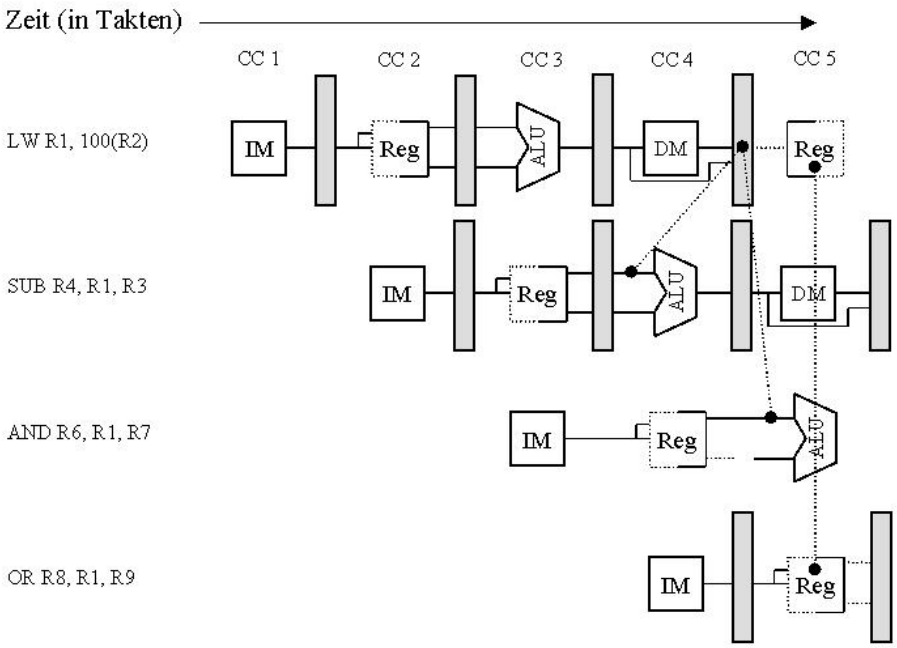
**OR R8, R1, R9**

Die folgende Folie zeigt das Problem:

Die LW-Instruktion hat den Wert für R1 erst nach deren MEM-Phase, aber bereits während ihrer MEM-Phase läuft die EX-Phase der SUB-Instruktion, die den Wert von R1 benötigt.

Ein forwarding-Pfad, der dies verhindern kann, müsste einen Zeitsprung rückwärts machen können, eine Fähigkeit, die selbst modernen Rechnerarchitekten noch versagt ist.

Wir können zwar für die AND und OR-Operation forwarden, aber nicht für SUB.
Für diesen Fall benötigen wir spezielle Hardwaremechanismen, genannt **Pipeline interlock**.




**Pipeline interlock** staut die Pipeline

**nur die Instruktionen ab der Verbraucherinstruktion für den Datenhazard werden verzögert.**

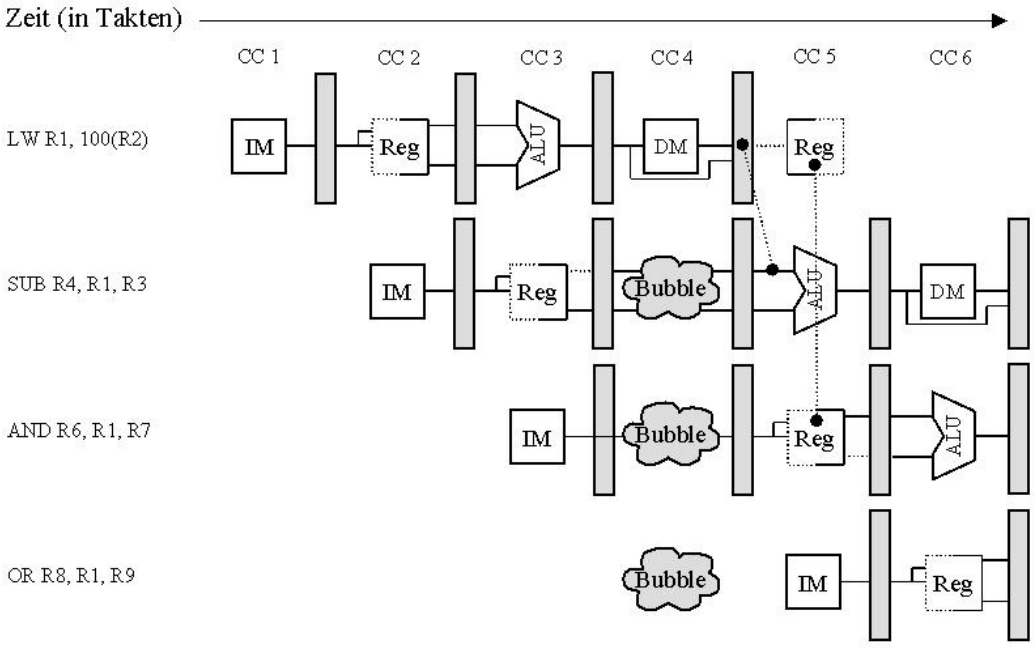
**die Quellinstruktion und alle davor müssen weiterbearbeitet werden**.

Dazwischen entsteht eine Pause, eine sogenannte **Pipeline-blase (bubble)**, in der nichts sinnvolles berechnet wird.

Im Beispiel sieht das so aus:

alles verzögert sich um einen Takt ab der Bubble.

In Takt vier wird keine Instruktion begonnen und also auch keine gefetcht.




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **LW R1,0(R1)** | IF | ID | EX | MEM | WB | | | | |
| **SUB R4,R1,R3** | | IF | ID | EX | MEM | WB | | | |
| **AND R6,R1,R7** | | | IF | ID | EX | MEM | WB | | |
| **OR R8,R1,R9** | | | | IF | ID | EX | MEM | WB | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **LW R1,0(R1)** | IF | ID | EX | MEM | WB | | | | |
| **SUB R4,R1,R3** | | IF | ID | **stall** | EX | MEM | WB | | |
| **AND R6,R1,R7** | | | IF | **stall** | ID | EX | MEM | WB | |
| **OR R8,R1,R9** | | | | **stall** | IF | ID | EX | MEM | WB |