Hochschule Bremerhaven

Unterlagen zur Lehrveranstaltung

Digital- und Mikroprozessortechnik
[ET–DMT]

- Teil 1: Digitaltechnik
  - Zahlendarstellungen
  - Boolsche Logik
- Teil 2: Sequentielle Logik / getaktete Systeme
  - Speicher
  - Zustands-Automaten
- Teil 2: Mikroprozessortechnik
  - CPU-Architekturen
  - Aufbau von Mikrocontrollern
  - Instruktionssatz und Assemblerprogrammierung
  - Steuerungen und Regelungen mit Mikrocontrollern

Revision: V1.3a (major)

Datum: April 2020

Prof. Dr.-Ing. Kai Müller
Hochschule Bremerhaven
Institut für Automatisierungs- und Elektrotechnik
An der Karlstadt 8

D–27568 Bremerhaven

Tel: +49 471 48 23 – 415
FAX: +49 471 48 23 – 418

E–Mail: kmueller@hs–bremerhaven.de

I Einleitung

I.I Umdruck zur Vorlesung

Über die Homepage der Vorlesung <http://www1.hs-bremerhaven.de/kmueller/> werden aktualisierte oder korrigierte Unterlagen im Verlauf der Vorlesung zur Verfügung gestellt.

I.II Digital- und Mikroprozessortechnik

Die Digitaltechnik hat nicht nur die Entwicklung der Computer ermöglicht, sondern sie beeinflusst wie kaum eine andere Technologie die Entwicklung von Produkten und Anlagen. Beispielsweise werden in einigen PKWs über 60 Mikroprozessoren eingesetzt, die Funktionen wie Motormanagement, ABS, ESP usw. übernehmen. Dieser sogenannte Embedded-Markt hat einen besonders großen Zuwachs zu verzeichnen; der Anteil der Digitaltechnik in vielen Produkten nimmt ständig zu. Die Veranstaltung Digitaltechnik umfasst eine Einführung in die binäre Logik und deren technischer Verwirklichung auf Gatterebene. In den letzten Jahren haben programmierbare Bauteile stark an Bedeutung gewonnen. In der Vorlesung und den Übungen soll dieser Entwicklung Rechnung getragen werden. Am Beispiel eines Mikrocontrollers und einem modernen Entwicklungssystem soll das wichtige Gebiet der Mikroprozessoren und Mikrocontroller berücksichtigt werden.

Folgende Themen sollen behandelt werden:

- Zahlensysteme (dezimal, binär, hexadezimal)
- Logische Funktionen
- Digitale Schaltungen
- Sequentielle Schaltungen (Schaltwerke)
- CPU, Mikrocontroller (am Beispiel PicoBlaze)

Ich wünsche allen Hörern der Veranstaltung “Digital- und Mikroprozessortechnik” viel Freude an einem faszinierenden Fachgebiet, das wie kaum ein anderer Bereich unser tägliches Leben bestimmt.

Bremerhaven, März 2018

Kai Müller
<kmueller@hs-bremerhaven.de>
Tel: (0471) 4823 – 415

II Inhalt

Teil 1

1 Analoge und digitale Systeme

Unter System verstehen wir Einrichtung oder Anlage, die aufgrund einer Einwirkung (Eingangssignal) eine Wirkung an seinem Ausgang als Folge des Eingangssignals erzeugt.

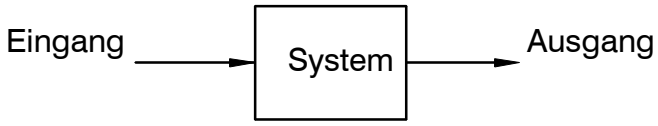


Bild 1.1: System

Unter einem analogen System verstehen wir ein System, bei dem die Signale jeden beliebigen Wert innerhalb eines *kontinuierlichen* Wertebereichs annehmen können und diese Werte auch von *technischer Bedeutung* sind (Beispiel: Temperatur eines Backofens).

Bei *Digitale Systemen* nehmen die Signale auch kontinuierliche Werte an. Jedoch interessieren hier nur zwei Werte:

> Bei einem digitalen System interessieren nur zwei Werte oder Zustände der Ein- und Ausgangssignale: 0 oder 1, bzw. TRUE oder FALSE, ON oder OFF bzw. LOW oder HIGH.

Häufig werden bei digitalen Systemen wieder kontinuierliche Signale benötigt (z.B. CD-Player). Diese Aufgabe übernehmen Digital-Analog-Umsetzer (DAC) bzw. Analog-Digital-Umsetzer (ADC).

> Man bezeichnet die analoge Ein-/Ausgangssignale auch als koniuierliche Signale.
> Bei einem digitalen System spricht man auch von *diskreten* Signale, die nur die diskreten Werte 0 bzw. 1 annehmen.

Mit anderen Worten: Wir tun so, als wenn bei einem digitalen System nur zwei verschiedene Werte auftreten.

Die Vorteile der Digitaltechnik gegenüber einer Verwirklichung mit analogen Schaltungen sind jedoch so immens, dass immer mehr analoge Funktionen durch digitale Systeme ersetzt werden. Beispiele:

Langspielplatte → CD-Player
Kamera mit Film → Digitalkamera

Darüber hinaus können mit digitalen Systemen neue Funktionen verwirklicht werden (Computer, ABS / ESP sowie digitale Motorsteuerung bei Kraftfahrzeugen, digitale Nachrichtenübermittlung bei Mobiltelefonen oder im Fernseh- und Rundfunkbereich).

Wichtige Vorteile der Digitaltechnik gegenüber analogen Ausführungen:

- Hohe Flexibilität durch Programmierbarkeit
- Möglichkeit, beliebig komplexe Funktionen zu realisieren
- Störungsfreie Übertragung und Signalverarbeitung (DAB bzw. DVB Rundfunk)

2 Zahlensysteme

Polyadische Zahlen

Gebräuchliche Zahlensysteme sind polyadisch aufgebaut (im Gegensatz z.B. zu den römischen Zahlen), d.h. man bezieht die Zahl auf eine *Basis* **B**.

Eine positive Zahl mit N Stellen hat folgenden Wert

$$n = \sum_{i=0}^{N-1} b_i B^i = b_{N-1} B^{N-1} + b_{N-2} B^{N-2} + \ldots + b_2 B^2 + b_1 B + b_0 . \quad (1.1)$$

$b_i =$ Stellenwert der Stelle $(0 \le b_i < B)$
$B^i =$ Stellenfaktor der Stelle

N nennt man auch die *Wortbreite* einer Zahl.

Ist die Basis bekannt, wird die Zahl n in der Form

$$n = b_{N-1} b_{N-2} \ldots b_1 b_0 \quad (1.2)$$

geschrieben. Oft schreibt man die Basis als Index hinter die Zahl ($132_{10} = 132$ im Dezimalsystem). Bei Dezimalzahlen lässt man den Index 10 aber fast immer weg.

Im Alltag verwenden wir das arabische Zahlensystem mit der Basis $B = 10$ (**Dezimalzahlensystem**).
Beispiel:

$$n = 132 = 1 \cdot 10^2 + 3 \cdot 10 + 2 \quad (1.3)$$

Häufig – insbesondere in Programmen – ist das Horner-Schema zur Berechnung und zur Konversion zwischen verschiedenen Zahlensystemen sinnvoll:

Horner-Schema: $n = \Big(\Big(...\Big(\big(b_{N-1}B + b_{N-2}\big)B + b_{N-3}\Big)B... + b_2\Big)B + b_1\Big)B + b_0$

2.1 Dual-, Oktal- und Hexadezimalzahlen

Das dezimale Zahlensystem ist für digitale Anwendungen ungeeignet, da es in der Digitaltechnik nur zwei Zustände, d.h. 0 und 1 gibt. Das Dualzahlensystem besitzt die kleinstmögliche Basis (2). Jede Stelle hat nur die Werte 0 der 1 und wird als

Bit (binary digit) $b_i \in \{0, 1\}$

bezeichnet.

Oktal und Hexadezimalzahlen lassen sich aus dem Dualzahlensystem ableiten.

2.1.1 Dualzahlen

Die Dualzahl

$$n = (b_{N-1}\, b_{N-2}...b_2\, b_1\, b_0)_2 \tag{1.4}$$

hat somit den Wert

$$n = b_{N-1}\, 2^{N-1} + b_{N-2} 2^{N-2} + ... + b_2\, 2^2 + b_1 2 + b_0\,. \tag{1.5}$$

Beispiel:

$$(1\,1\,1\,0\,1\,0\,1)_2 = 2^6 + 2^5 + 2^4 + 2^2 + 1 = (117)_{10}$$

Das äußerst rechte Bit nennt man ***LSB*** (least significant bit).

Das höchstwertige Bit nennt man ***MSB*** (most significant bit).

2.1.2 Oktalzahlen

Basis $B = 8$
Zeichenvorrat: $b_i \in \{0, 1, ... , 7\}$

Oktalzahlen lassen sich leicht aus Dualzahlen bilden, indem immer drei Dualstellen zu einer Oktalstelle zusammengefasst werden. Umgekehrt kann jede Oktalstelle leicht durch drei Dualstellen ersetzt werden.

Beispiel: Zahl 117_{10}

$165_8 = 1\,8^2 + 6\,8 + 5 = 64 + 48 + 5 = 117_{10}$

Umwandlung in Dualzahl:

$165_8 = (001\ \ 110\ \ 101)_2 = 1110101_2$

In vielen Programmiersprachen werden Oktalzahlen als 0oXXX geschrieben, d.h. im Fall der Zahl 165_8 schreibt man "0o165".

2.1.3 Hexadezimalzahlen

In der Digitaltechnik sehr verbreitet sind Hexadezimalzahlen, da sie sich leicht in Dualzahlen konvertieren lassen. Dadurch, dass jeweils vier Dualstellen eine Hexadezimalstelle bilden, eignen sich Hexadezimalzahlen gut für die meisten Wortbreiten der Computer-Rechenwerke (8 Bit, 16 Bit, 32 Bit bzw. 64 Bit).

Basis $B = 16$
Zeichenvorrat: $b_i \in \{0, 1, \ldots, 7, 8, 9, A, B, C, D, E, F\}$

Die A, B, C, D, E, F im Hexadezimalsystem haben die Wertigkeiten 10, 11, 12, 13, 14, 15 im Dezimalsystem (es wird ja ein Zeichenvorrat von B Zeichen benötigt).

Beispiel: Zahl 117_{10}

$75_{16} = 7\,16 + 5 = 112 + 5 = 117_{10}$

Umwandlung in Dualzahl:

$75_8 = (0111\ \ 0101)_2 = 1110101_2$

In vielen Programmiersprachen werden Hexadezimalzahlen als 0xXXX geschrieben, d.h. im Fall der Zahl 75_{16} schreibt man "0x75".

Weit verbreitet ist auch an angestelltes "H" als Kennung, z.B. $75H = 75_{16}$.

2.1.4 BDC-Zahlen (binary coded decimal)

Ein Beispiel für ein nicht-polyadisches Zahlensystem ist der BCD-Code, die gelegentlich für die Ausgabe von Zahlen an Computern oder Messeinrichtungen verwendet wird.

BCD-Zahlen entstehen durch Kodierung jeder einzelnen dezimalen Stelle in das Dualzahlensystem. Für den Zahlenbereich 0..9 werden 4 Dualstellen benötigt.

Beispiel: Zahl 117_{10} im als BCD-Zahl

$117_{10} = (0001\ 0001\ 0111)_{\mathrm{BCD}}$

2.2 Konversion von Dezimalzahlen in Dual, Oktal und Hexadezimalzahlen

Der Algorithmus zur Konversion wird aus dem Hornerschema zur Berechnung Dezimalzahlen aus anderen Zahlensystemen deutlich (s. Abschnitt 2).

$$n = \Big(\Big(...\Big(\big(b_{N-1}B + b_{N-2}\big)B + b_{N-3}\big)B... + b_2\Big)B + b_1\Big)B + b_0 \qquad (1.6)$$

Dividiert man (1.6) ganzzahlig durch B, so erhält man als Rest den Wert b_0. Der Quotient ist dann

$$\Big(...\Big(\big(b_{N-1}B + b_{N-2}\big)B + b_{N-3}\big)B... + b_2\Big)B + b_1\,. \qquad (1.7)$$

Die weitere Division durch B liefert als Rest die Stelle b_1 im Zielsystem.

Die Division im Dezimalsystem wird so lange fortgesetzt, bis der Quotient null wird. Die Reste bilden jeweils die Stellen im Zielsystem.

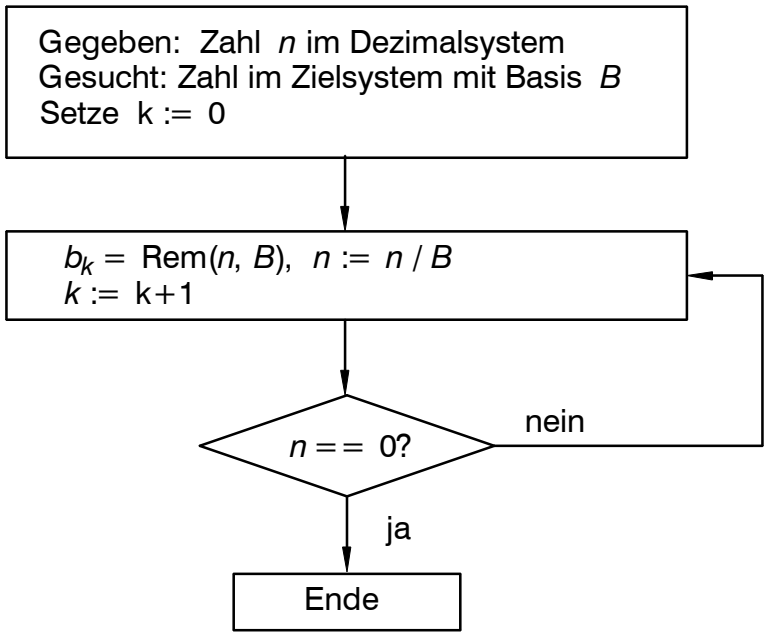


Bild 1.2: Algorithmus zur Berechnung von Zahlen im beliebigen Zahlensystem aus Dezimalzahlen

2.2.1 Aufgaben

1. Konversion von 102_{10} in das Dualsystem [B = 2]

k n b_k (Rest im Dualsystem)

0	102 / 2 = 51	0
1	51 / 2 = 25	1
2	25 / 2 = 12	1
3	12 / 2 = 6	0
4	6 / 2 = 3	0
5	3 / 2 = 1	1
6	1 / 2 = 0	1

[Ende, da $n = 0$ ist.]

Schreibt man die Reste in die Reihenfolge $b_k \ldots b_0$, so erhält man
$n = 1100110_2$.

---

2. Konversion von 606_{10} in das Hexadezimalsystem [B = 16]

k	n	b_k (Rest im Hexadezimalsystem)
0	606 / 16 = 37	E
1	37 / 16 = 2	5
2	2 / 16 = 0	2

[Ende, da $n = 0$ ist.]

Schreibt man die Reste in die Reihenfolge $b_k \ldots b_0$, so erhält man
$n = 25E_{16}$.

Eine Umsetzung in das Dualzahlensystem kann stellenweise erfolgen. Mit
$E_{10} = 1110_2$, $5_{10} = 0101_2$, $2_{10} = 0010_2$ folgt
$n = 10\,0101\,1110_2$

3 Arithmetische Operationen nichtdezimaler Zahlen (nur Ganzzahlarithmetik)

Arithmetische Operationen im nichtdezimalen Zahlensystem erfolgen nach den gleichen, bekannten Rechenregeln, jedoch sind die Berechnungen ungewohnt.

Besonders einfache Regeln gelten für die Dualzahlen, da der Wertebereich der einzelnen Stellen nur 0 und 1 umfasst. Wie bei jeder Addition kann ein Überlauf in die nächste Stelle erforderlich werden. Dieser Überlauf wird "*Carry*" genannt. Die bitweise Addition kann in folgender Tabelle dargestellt werden:

Einstellige duale Addition

Eingänge			Ausgänge	
$carry_{in}$	x_k	y_k	$x_k + y_k$	$carry_{out}$
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

Die entsprechende Tabelle für das Dezimalsystem wäre wesentlich umfangreicher.

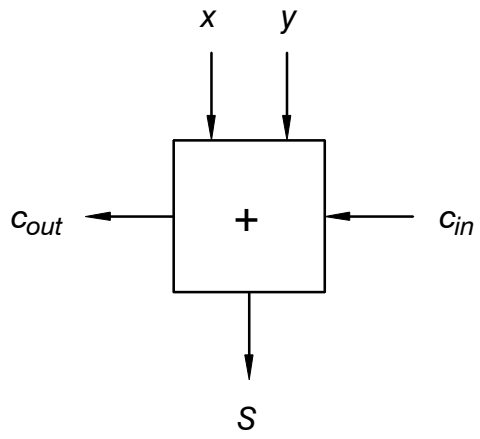


Bild 1.3: Symbol für die einstellige Addition

Beispiel: Addition von 110100111_2 und 101110_2

```
   1 1 0 1 0 0 1 1 1                  (423)10
+        1 0 1 1 1 0                   (46)10
--------------------
       1 0 1 1 1 0 0   (Carry Bits)
--------------------
   1 1 1 0 1 0 1 0 1                  (469)10
```

Bei der Subtraktion kann ein Unterlauf (*Borrow*) auftreten. In der Tabelle für die einstellige duale Subtraktion sind die Rechenregeln zusammengefasst

Einstellige duale Subtraktion

Eingänge			Ausgänge	
$borrow_{in}$	x_k	y_k	$x_k - y_k$	$borrow_{out}$
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	1

Beispiel: Subtraktion von 110100111_2 minus 101110_2

```
   1 1 0 1 0 0 1 1 1                  (423)10
-        1 0 1 1 1 0                   (46)10
--------------------
-  0 1 1 1 1 0 0 0 0   (Borrow Bits)
--------------------
   1 0 1 1 1 1 0 0 1                  (377)10
```

Die Differenz wird oft benutzt, um Zahlen zu vergleichen.

Ist das letzte Borrow “1”, so ist $Y > X$, anderenfalls ist $Y \leq X$.

3.1 Vorzeichenbehaftete Zahlen und Komplementdarstellung

Die zuvor beschriebene Subtraktion ist in der Digitaltechnik unüblich, da man verschiedene Rechenwerke für Addition und Subtraktion benötigt. Außerdem möchte man das Vorzeichen der Zahl nicht gesondert betrachten (ganze Zahlen). Im folgenden werden wir die vorzeichenbehaftete Subtraktion auch mit einem Addierwerk ausführen.

3.1.1 Negative Zahlen

Eine gesonderte Speicherung des Vorzeichens wäre aufwendig und würde zusätzliche Logik erfordern. Es wird deshalb bei vorzeichenbehafteten Zahlen das *MSB* als Vorzeichen interpretiert.

MSB = 0 → positive Zahl, MSB = 1 → negative Zahl

Diese Darstellung wird als “signed magnitude” bezeichnet (= Betrag + Vorzeichenbit). Wird nur das MSB als Vorzeichen interpretiert, kann man zwar ganze Zahlen darstellen, man muss dennoch, Addition und Subtraktion unterschiedlich handhaben.

Damit Zahlen eindeutig vorzeichenbehaftet darstellbar sind, muss man sich auf eine *feste Wortlänge* einigen (z.B. 32 Bit).

3.1.2 Komplementdarstellung ganzer Zahlen (Basiskomplement)

Die Darstellung negativer Zahlen als Komplement ist komplizierter als das bloße Ändern des Vorzeichenbits, jedoch wird die Rechnung in der Komplementdarstellung besonders einfach.

Komplement: negative Zahl = C − positive Zahl

Es existieren zwei Verfahren: das Einer-Komplement und das Zweier-Komplement.

Einer-Komplement: $C_1 = 2^N - 1$, (allgemein $C_1 = B^N - 1$)

Zweier-Komplement: $C_2 = 2^N$, (allgemein $C_2 = B^N$)

Beispiel: Bildung des Einer- und des Zweier-Komplements (8 Bit Wortbreite)

Einer-Komplement von 00110110

```
  0 1 1 1 1 1 1 1 1    (C1 = 2^8 - 1 = 255)
-   0 0 1 1 0 1 1 0    (54)
-------------------
    1 1 0 0 1 0 0 1    (Einer-Komplement)
```

Man erkennt, dass die Berechnung sich in der Umkehrung aller Bits erschöpft. Es muss also gar keine Subtraktion durchgeführt werden.

Zweier-Komplement von 00110110

```
  1 0 0 0 0 0 0 0 0    (C2 = 2^8 = 256)
-   0 0 1 1 0 1 1 0    (54)
-------------------
    1 1 0 0 1 0 1 0    (Zweier-Komplement)
```

Auch zur Bestimmung des Zweier-Komplements ist keine Subtraktion nötig. Man erkennt dies, wenn man C_2 als $(C_2-1)+1 = C_1+1$ schreibt. Damit wird

$C_2{-}x = C_1{+}1{-}x = (C_1{-}\mathrm{x}){+}1$. Der Ausgruck in Klammern ist aber das Einer-Komplement. Es müssen somit alle Stellen der ursprünglichen Zahl invertiert werden and anschließend muss eine 1 addiert werden.

Zweier-Komplement: Invertieren aller Stellen; anschließend Addtion von 1.

Zweier-Komplement von 00110110 (schnelles Verfahren)

```
    0 0 1 1 0 1 1 0    (54)
------------------
    1 1 0 0 1 0 0 1    (Inversion aller Stellen)
+   0 0 0 0 0 0 0 1    (Addition von 1)
------------------
    1 1 0 0 1 0 1 0    (Zweier-Komplement)
```

Vorzeichenbehaftete Zahlen mit 3 Bit Wortbreite (N = 2, N+1 Stellen)

Einer-Komplement

Dezimal	−3	−2	−1	−0	0	1	2	3
Dual	100	101	110	111	000	001	010	011

Zweier-Komplement

Dezimal	−4	−3	−2	−1	0	1	2	3
Dual	100	101	110	111	000	001	010	011

Man erkennt, dass im Einer-Komplement die Null zweimal auftritt (als $+0$ und -0). Die Zahlenbereiche sind

Einer-Komplement: $\mathrm{n} \in \{-2^N{+}1, ..., 2^N{-}1\}$

Zweier-Komplement: $\mathrm{n} \in \{-2^N, ..., 2^N{-}1\}$, asymmetrischer Zahlenbereich

Das Zweier-Komplement bildet eine *monotone* Zahlenfolge, d.h. durch Addition von 1 erhält man die nächstfolgende Zahl. Wir werden im folgenden deshalb nur das Zweier-Komplement verwenden, das ein Standard bei der Berechnung von ganzen Zahlen auf nahezu allen Computern ist.

3.1.3 Werte der Bits bei Beträgen und vorzeichenbehafteten Zahlen

Werden negative Zahlen nicht benötigt, so wird auch kein Vorzeichenbit verwendet. Bei einer N+ 1-stelligen Zahl haben damit die einzelnen Bits folgende Bedeutung.

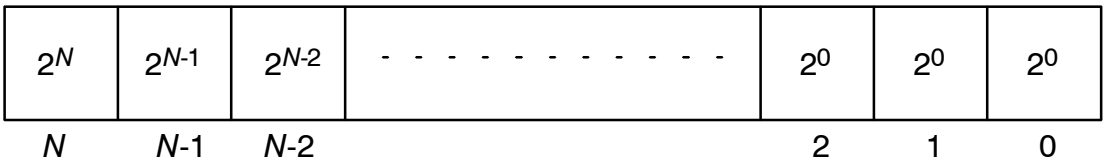


Bild 1.4: Betragszahlen (nicht-negative Zahlen)

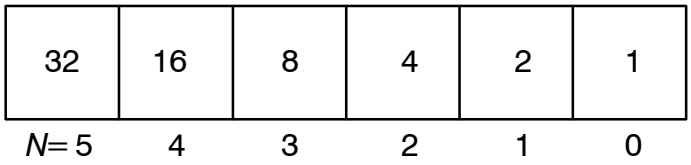


Bild 1.5: Betragszahl (Beispiel $N = 5$, 6 Bits)

Vorzeichenbehaftete Zahlen (ganze Zahlen) werden überwiegend aus numerischen Gründen im Zweier-Komplement dargestellt. Die Bedeutung der einzelnen Bits zeigt Bild 1.6.

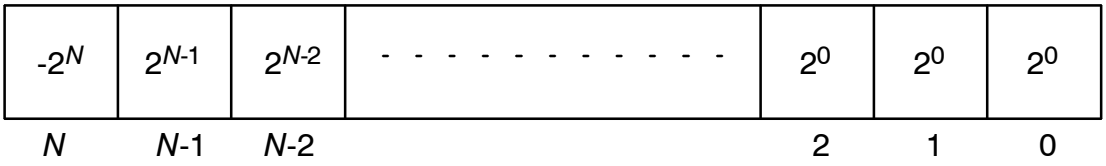


Bild 1.6: Ganze Zahlen (Zahlen mit Vorzeichen)

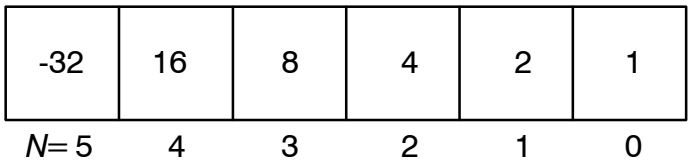


Bild 1.7: Ganze Zahlen (Beispiel $N = 5$, 6 Bits)

Die Bedeutung des MSB bei vorzeichenbehafteten Zahlen im Zweier-Komplement ergibt sich natürlich aus $C = 2^N$ (Wert -32 im MSB bei 6 Bits).

Der Zahlenbereich für Vorzeichenzahlen beträgt somit

$$-2^N \leq x \leq 2^{N-1} \tag{1.8}$$

bzw. $-32 \leq x \leq 31$ für 6-Bit-Zahlen.

3.1.4 Subtraktion und Addition von Zahlen in Komplementdarstellung

Da die Wortlänge fest ist, gibt es auch einen festen Wertebereich, in dem das Ergebnis liegen muss. Die Addition bzw. Subtraktion betragsgroßer Zahlen können leicht außerhalb des zulässigen Wertebereichs liegen.

Jede Subtraktion kann als Addition des Komplements geschrieben werden

$$x - y = x - y + C - C = x + (C - y) - C = x + \bar{y} - C . \tag{1.9}$$

Die Zahl y bezeichnet man als Komplement bezüglich C. Die Zahl C ist beim Zweier-Komplement

$$C = 2^N \tag{1.10}$$

und liegt damit außerhalb des erlaubten Zahlenbereiches. Die Subtraktion von C in (1.9) kann aber entfallen, da C im erlaubten Bereich der Stellen nur Nullen enthält. Damit ändern die Subtraktion von C das Ergebnis (im erlaubten Stellenbereich) nicht.

Beispiele (Wortlänge 4 Bit)

Bei 4 Bit Wortlänge $(N = 4)$ beträgt der Zahlenbereich
$n \in \{-2^{N-1}, ..., 2^{N-1}-1\} = \{-8, ..., 7\}$

1. $2 + 3 = 5$

```
  0 0 1 0   (2)
+ 0 0 1 1   (3)
-----------
  0 1 0 1   (5)
```

2. $2 - 3 = 2 + (-3) = -1$

   Hierzu wird die Zahl -3 als Zweier-Komplement geschrieben

```
  0 0 1 1   (3)
-----------
  1 1 0 0   (Inversion der Stellen)
+ 0 0 0 1   (Addition von 1)
-----------
  1 1 0 1   (-3)
```

   Jetzt wird die Addition von $2 + (-3)$ ausgeführt

```
  0 0 1 0   (2)
+ 1 1 0 1   (-3)
-----------
  1 1 1 1   (-1)
```

3. $-2 - 3 = (-2) + (-3) = -5$

```
    1 1 1 0   (-2)
+   1 1 0 1   (-3)
  -----------
(1) 1 0 1 1   (-5),   Carry Bit wird ignoriert
```

4. $4 - 7 = 4 + (-7) = -3$

```
    0 1 0 0   (4)
+   1 0 0 1   (−7)
  ---------
    1 1 0 1   (−3)
```

5. $-3 - 6 = (-3) + (-6) = -9$?

```
    1 1 0 1   (−3)
+   1 0 1 0   (−6)
  ---------
  1 0 1 1 1   (+7)    (Überlauf, Overflow)
```

Ein Überlauf tritt auf, wenn bei der Addition von Zahlen mit gleichem Vorzeichen ein anderes Vorzeichen im Ergebnis auftritt. Die Addition von Zahlen mit unterschiedlichem Vorzeichen führt niemals zum Überlauf.

Mikroprozessoren besitzen oft ein besonderes Bit (Flag), um einen Überlauf nach einer arithmetischen Operation anzuzeigen.

3.1.5 Grafische Darstellung der Zahlen mit 4 Bit Wortlänge

Additions- und Subtraktionsoperationen lassen sich auch grafisch anhand eines sogenannten Zahlenrings darstellen

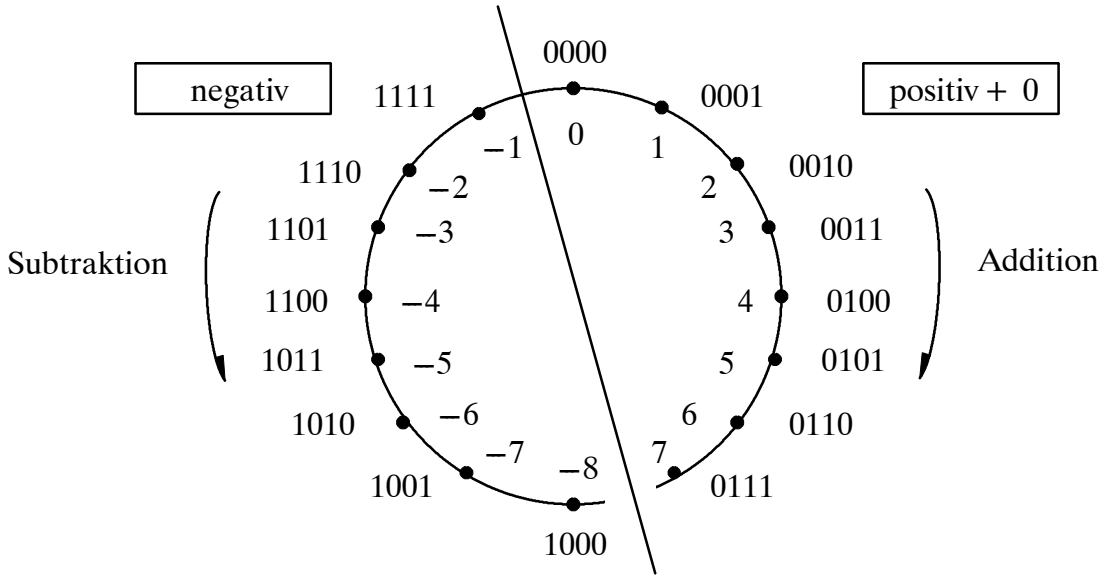


Bild 1.8: Zahlenring für 4 Bit Zweier-Komplement-Zahlen

3.2 Multiplikation von Dualzahlen

Jede Multiplikation kann auf eine Folge von Additionen zurückgeführt werden. Im Fall der Dualzahlen ist dies besonders einfach, da immer nur mit 0 bzw 1 multipliziert werden muss.

Einstellige duale Multiplikation

Eingänge		Ausgang
x_k	y_k	$x_k \times y_k$
0	0	0
0	1	0
1	0	0
1	1	1

Multiplikation von 3 mit 5

```
      0 0 1 1 x 0 1 0 1    3 x 5
-----------------------
      0 0 1 1
    0 0 0 0
  0 0 1 1
-----------------------
  0 0 1 1 1 1              15
```

Es ist bei umfangreichen Berechnungen günstiger, die Berechnungen iterativ mit Hilfe von Zwischensummen und Linksverschieben der Ergebnisse durchzuführen:

```
0 0 1 1 x 0 1 0 1      3 x 5
-----------------------
          0 0 0 0      Zwischenergebnis = 0
+         0 0 1 1
----------------
          0 0 1 1      Zwischenergebnis = 3
+       0 0 0 0        Multiplikand um 1 nach links verschoben
----------------
        0 0 0 1 1      Zwischenergebnis = 3
+     0 0 1 1          Multiplikand um 2 nach links verschoben
----------------
      0 0 1 1 1 1      Zwischenergebnis = 15
+   0 0 0 0            Multiplikand um 3 nach links verschoben
-----------------------
    0 0 0 1 1 1 1      Endergebnis = 15
```

Die Wortbreite des Ergebnisses ist Summe der Wortbreiten der Produkte.

3.2.1 Multiplikation von Zweier-Komplement-Zahlen

Auch die Multiplikation vorzeichenbehafteter Zahlen kann als Folge von Additionen erfolgen. Besondere Beachtung muss das MSB des Multiplikators erfahren, da es sich nicht um einen Faktor wie bei den übrigen Stellen handelt, sondern das Vorzeichen abbildet.

Bei der Schiebeoperation muss jeweils das Vorzeichenbit kopiert werden.

Multiplikation von (–3) x (–5)

```
1 1 0 1 x 1 0 1 1      (-3) x (-5)
----------------------
          0 0 0 0      Zwischenergebnis = 0
+         1 1 0 1
-----------------
        1 1 1 0 1      Zwischenergebnis = -3 + Vorzeichenerw.
+       1 1 0 1        Multiplikand um 1 nach links verschoben
-----------------
      1 1 0 1 1 1      Zwischenergebnis = -9
+     0 0 0 0          Multiplikand um 2 nach links verschoben
-----------------
    1 1 1 0 1 1 1      Zwischenergebnis = -9 + Vorzeichenerw.
+   0 0 1 1            Mult. um 3 verschoben und negiert
----------------------
(1) 0 0 0 1 1 1 1      Endergebnis = 15,  Carry  wird ignoriert
```

3.3 Division von Dualzahlen

Jede Division kann auf eine Folge von Subtraktionen zurückgeführt werden. Zu beachten ist, dass eine Division durch null unzulässig ist.

Einstellige duale Division

Eingänge		Ausgang
x_k	y_k	x_k / y_k
0	0	nicht definiert
0	1	0
1	0	nicht definiert
1	1	1

Eine ganzzahlige Division x / y liefert den Quotienten und den (ganzzahligen) Rest (Remainder)

$$x = Q \cdot y + R . \tag{1.11}$$

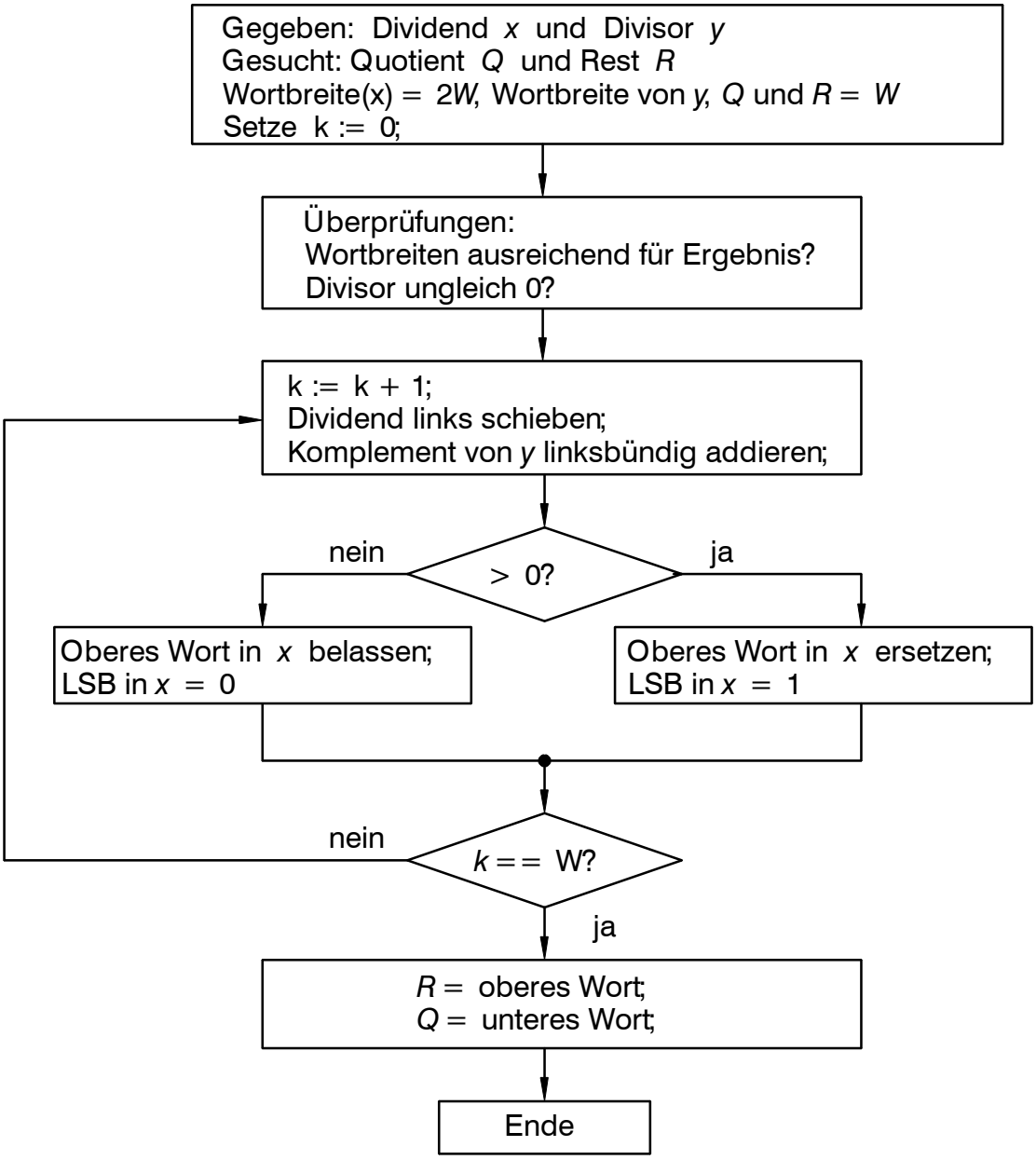


Bild 1.9: Algorithmus zur Division (positiver) Zahlen im Dualsystem (Restoring-Algorithmus)

Das Verfahren beruht auf einer Folge bedingter Subtraktionen des des Divisors vom Dividenden. Viele Mikroprozessoren und DSPs (*Digital Signal Processor*) unterstützen diesen Algorithmus durch spezielle Befehle (*conditional subtract*), sofern die Division nicht Teil des Assembler-Befehlssatzes ist.

Beispiel: Division von 44 durch 7: 0 1 1 1, Zweier-Komplement von 7: 1 0 0 1

```
     0 0 1 0  1 1 0 0    (44)
--------------------
     0 1 0 1  1 0 0 x    1. Linksshift Dividend
+    1 0 0 1             Addition des Komplements von 7
 --------------------
     1 1 1 0             Ergebnis negativ => LSB = 0
--------------------
     0 1 0 1  1 0 0 0    => Restore, LSB = 0
     1 0 1 1  0 0 0 x    2. Linksshift
+    1 0 0 1
--------------------
     0 1 0 0             Ergebnis positiv => LSB = 1
--------------------
     0 1 0 0  0 0 0 1    => Ersetzen, LSB = 1
     1 0 0 0  0 0 1 x    3. Linksshift
+    1 0 0 1
--------------------
     0 0 0 1             Ergebnis positiv  => LSB = 1
--------------------
     0 0 0 1  0 0 1 1    => Ersetzen, LSB = 1
     0 0 1 0  0 1 1 x    4. Linksshift
     1 0 0 1
--------------------
     1 0 1 1             Ergebnis negativ  => LSB = 0
--------------------
     0 0 1 0  0 1 1 0    Endergebnis => R = 2, Q = 6
====================
```

Die beschriebene Division verwendet positive Zahlen (Beträge). Für ganze Zahlen muss entsprechend den elementaren Rechenregeln zusätzlich das Vorzeichen interpretiert werden.

Es wird ersichtlich, dass eine CPU für die Division erheblich mehr Zeit benötigt als beispielsweise für eine Addition.

Die Division wird benötigt, um höhere mathematische Funktionen wie beispielsweise die Wurzelfunktion zu berechnen zu können. Die (positive) Wurzel einer Zahl y

$$x = \sqrt{y} \tag{1.12}$$

kann iterativ gemäß

$$k_{k+1} = \frac{1}{2}\left(x_k + \frac{y}{x_k}\right) \tag{1.13}$$

bestimmt werden. Für x_0 kann der Startwert $x_0 = 1$ verwendet werden.

4 Gleitkommazahlen (Floating Point Numbers), IEEE Single Precision Floating

Als Beispiel für Gleitkommazahlen sei das gebräuchliche IEEE Single Precision Floating Format aufgezeigt (ANSI/IEEE Standard 754-1985).

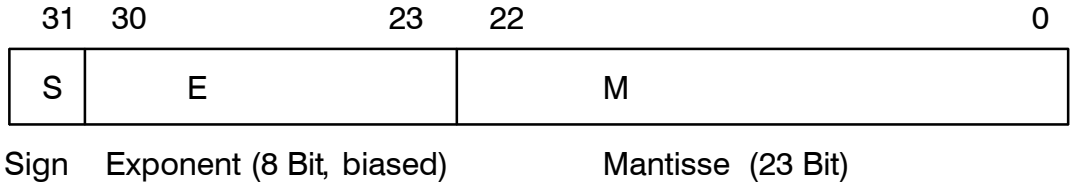


Bild 1.10: 32 Bit IEEE Floating Format (einfache Genauigkeit)

Der Wert der Gleitkommazahl berechnet sich nach der Formel

$$F = (1 - 2S) \cdot 2^{E-127} \cdot 1.\mathrm{M}\,. \tag{1.14}$$

Der Exponent hat einen Offset (bias) von 127, d.h. der Wert $E = 127$ bedeutet $2^0 = 1$. Die Mantisse hat eine "implizite" 1 vor dem Komma (sofern $M \neq 0$).

Beispiel: Gleitkommazahl $x = -6{,}5$

```
1  1 0 0 0 0 0 0 1  1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 =
0xC0D0000  (Hexadezimal)
Sign = 1  =>  negative Zahl
Exponent = 129  => 2^2
Mantisse = 1.0 + 0.5 + 0.125 = 1.625
```

$$x = (-1) \cdot 2^2 \cdot 1.625 = -6.5 \tag{1.15}$$

$E = 0$ und $E = 255$ haben eine besondere Bedeutung:

$E = 255$ und $F = 0$ und $S = 1 \rightarrow x = -\infty$
$E = 255$ und $F = 0$ und $S = 0 \rightarrow x = +\infty$
$E = 0$ und $F = 0$ und $S = 1 \rightarrow x = -0$
$E = 0$ und $F = 0$ und $S = 0 \rightarrow x = +0$

Durch weitere Regeln werden sogenannte *denormalisierte Zahlen* oder *NANs* (not a number) dargestellt.

Mit einfacher Genauigkeit (32 Bit) lassen sich etwa 7 Dezimalstellen genau darstellen.

Gleitkomma-Arithmetik erfolgt in Software (sehr langsam) oder bei größeren Mikroprozessoren in Hardware. Die Gleitkomma-Hardware hat in den letzten Jahre enorme Fortschritte gemacht. Insbesondere bei modernen RISC-CPUs (*Reduced*

Instruction Set Computer) erreicht man die gleiche Geschwindigkeit wie bei Ganzzahl-Arithmetik (Integer).

5 Weitere Kodierungen für Zahlen und Zeichen

5.1 Gray-Code

In einigen technischen Anwendungen ist es wichtig, dass beim Übergang von einer Zahl zu einer Anderen sich jeweils nur ein Bit ändert (z.B. bei Winkelsensoren in der Automatisierungstechnik).

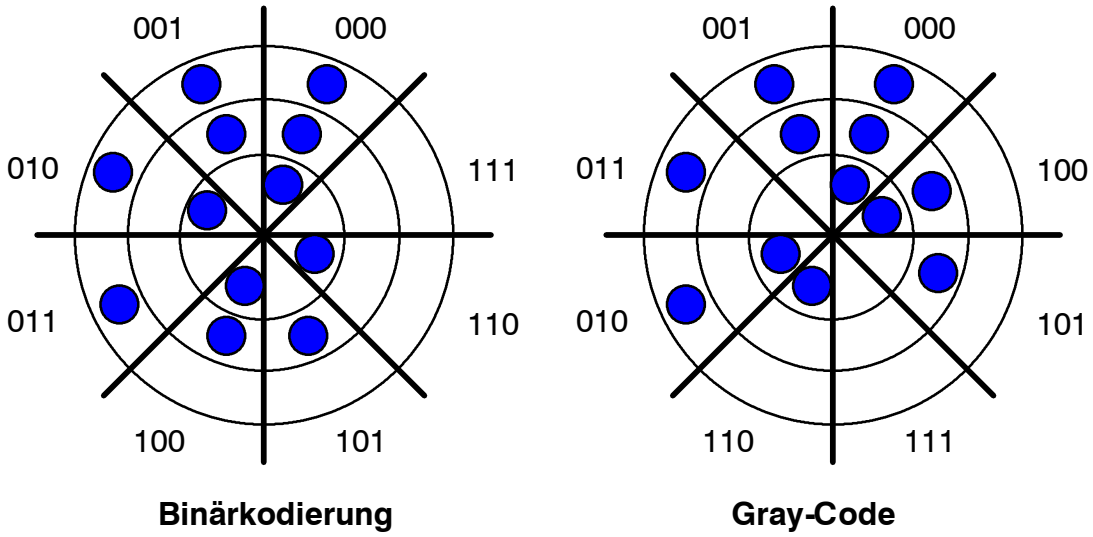


Bild 1.11: Binär- und Gray-kodierte Winkelsensoren

Man erkennt, dass sich von Sektor zu Sektor eines Winkeldecoders nur jeweils ein Bit ändert.

> Der Gray-Code vermeidet Lesefehler an den Übergängen von einer Zahl zur jeweils angrenzenden Zahl.

Bezeichnet man die drei Bits für den Gray-Code mit $g_2\, g_1\, g_0$ sowie die Binärbits mit $b_2\, b_1\, b_0$, so gelten folgende Regeln für die Umwandlungen.

Umwandlung Binär-Code in Gray-Code:

g2 = b2 (keine Änderung)
g1 = b1 XOR b2
g0 = b0 XOR b1

Mit Ausnahme des obersten Bits wird also immer das Binär-Bit mit seinem Vorgänger XOR verknüpft.

Umwandlung Gray-Code in Binär-Code:

b2 = g2 (keine Änderung)
b1 = g1 XOR b2
b0 = g0 XOR b1

Mit Ausnahme des obersten Bits wird immer das Gray-Bit mit den Vorgänger des gebildeten Binär-Bits XOR verknüpft.

5.2 Zeichensätze (Character Encoding)

In digitalen Systemen werden Buchstaben, Zahlen und Sonderzeichen als Zahl dargestellt. Für einfache Systeme genügen 8 Bit Zahlen (**= 1 Byte**, 256 Zeichen). Ein gebräuchlicher Zeichensatz ist der ASCII-Zeichensatz (*American Standard Code for Information Interchange*). Für komplexe internationale Zeichendarstellung verwendet man neuerdings den sogenannten UNICODE (hier werden Zeichen durch mehrere Bytes dargestellt.

Ein 8-Bit breites Wort bezeichnet man als Byte. Den Wert eines Bytes gibt man of im Hexadezimalsystem an (2 Stellen).

Tabelle der ASCII-Kodierung von Zeichen und Sonderzeichen (nicht druckbare Zeichen)

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

Beispiel: Zeichen “m”

Das Zeichen “m” wird durch die Zahl $6D_{16} = 0110\ 1101_2$ repräsentiert.

Die ASCII-Tabelle nutzt das Byte nicht vollständig aus, da nur die Zahlen 0..127 standardisiert sind. Die übrigen Zahlen (128..255) werden für nicht-standardisierte Zeichensätze verwendet.

5.3 Unicode (UTF)

Die Erweiterung der ASCII-Kodierung auf beliebige Zeichen ist UTF (Unicode Transformation Format). Das häufigste Format ist UTF-8 (insbesondere in HTML). Wird der 7-Bit-ASCII Code auf 8 Bit erweitert (MSB = 0), so ist ASCII mit UTF kompatibel.

Alle anderen Zeichen erfordern mehr Bytes (Multibyte Encoding). Das erste Byte beginnt dann mit einer Anzahl von ’1’-Bits, die die Anzahl der Bytes für ein Zeichen festlegt (maximal 6 x ’1’ ist zulässig). Alle folgenden Bytes müssen dann mit “10” beginnen. Damit ist UTF eindeutig kodiert. Die übrigen Bits enthalten die eigentliche Information über das darzustellende Zeichen.

1 Byte (ASCII):

```
0xxxxxxx
```

2 Bytes:

```
110xxxxx 10xxxxxx
```

3 Bytes:

```
1110xxxx 10xxxxxx 10xxxxxx
```

4 Bytes:

```
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

5 Bytes:

```
111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
```

6 Bytes:

```
1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
```

UTF ist bisher auf 4 Bytes beschränkt (5 oder 6 Bytes stellen Ausnahmen dar). Bei 4 Bytes besteht somit eine Netto-Information von 21 Bits zur Verfügung (= Anzahl ’x’). Zeichen dürfen nur in dem jeweils kürzesten Format übertragen werden. Beispielsweise muss das Zeichen ’a’ durch den ASCII-Code 0x16 (= 1 Byte) übertragen werden.

6 Fehlererkennung in Dualzahlen

Ein Fehler in digitalen Systemen kann durch physikalische Einflüsse entstehen, beispielsweise durch Einstreuung von elektromagnetischer Störungen oder durch Hardware-Defekte. Man ist deshalb bestrebt, diese Fehler zu korrigieren oder zumindest zu erkennen.

Man unterscheidet *einfache* und *mehrfache* Fehler. Im Fall mehrfacher Fehler sind mehrere Bits in einem Wort fehlerhaft. Da Einfachfehler sehr viel wahrscheinlicher sind, verwendet man oft Verfahren zur Korrektur bzw. Erkennung dieser Fehlerart.

6.1 Minimale Distanz

Die minimale Distanz gibt an, wieviele Bits von einem gültigen Wort zu einem benachbarten gültigen Wort geändert werden müssen. Dies lässt sich grafisch an einem 3-Bit Wort aufzeigen.

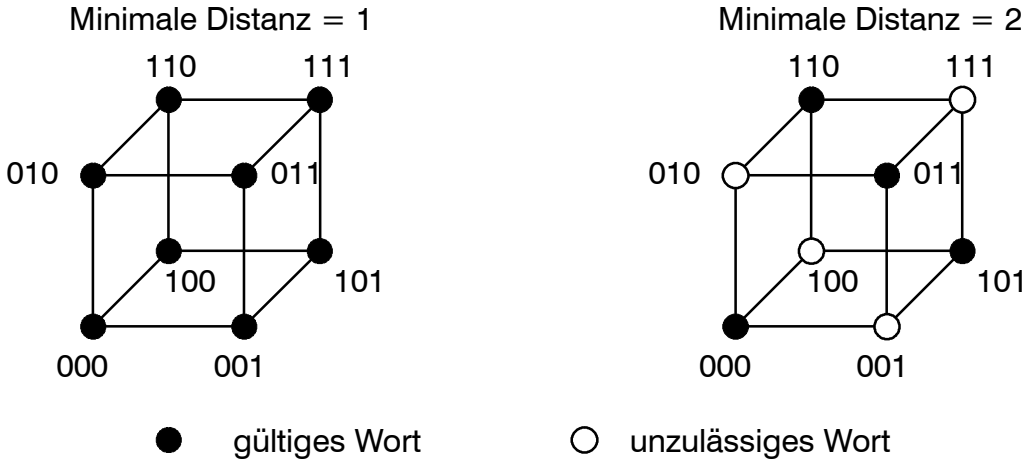


Bild 1.12: Grafische Darstellung der minimalen Distanz

Durch Änderung jeweils eines Bits kommt man von einem Punkt des linken Würfels zu einem angrenzenden Punkt. Wenn sich nur ein Bit ändert, kann man also nicht sagen, ob das Wort gültig ist oder ob ein Fehler aufgetreten ist.

In dem Rechten Würfel müssen sich jeweils zwei Bits ändern, um ein gültiges Wort zu erzeugen; ändert sich nur ein Bit, so muss ein Fehler aufgetreten sein.

Eine Einzelfehlererkennung erfordert eine minimale Distanz von mindesten 2.

Man kann diesen Sachverhalt auch als *Redundanz* auffassen. Für 2^n Worte benötigt man mindestens $n+1$ Bit, um Einzelfehler erkennen zu können.

6.1.1 Parity Bit

Die einfachste Art, eine minimale Distanz von 2 zu erzeugen, ist das Anfügen eines sogenannten Paritäts-Bits (parity bit). Hier wird ein Bit angefügt, das abhängig von einer geraden oder ungeraden Anzahl Bits mit einer “1” in einem Wort (Informations-Bits) gesetzt wird. Entsprechend unterscheidet man zwischen ungerader und gerader Parität.

Minimale Distanz = 2, gerade und ungerade Parität für 3 Informations-Bits

Informations-Bits	gerade Parität	ungerade Parität
0 0 0	0 0 0 0	0 0 0 1
0 0 1	0 0 1 1	0 0 1 0
0 1 0	0 1 0 1	0 1 0 0
0 1 1	0 1 1 0	0 1 1 1
1 0 0	1 0 0 1	1 0 0 0
1 0 1	1 0 1 0	1 0 1 1
1 1 0	1 1 0 0	1 1 0 1
1 1 1	1 1 1 1	1 1 1 0

Mit Hilfe des zusätzlichen Paritäts-Bits lassen sich in diesem Beispiel also Einzelfehler in einem Wort mit 3 Information-Bits erkennen.

Durch Anfügen weiterer Bits lassen sich mehrere Fehler erkennen oder Einzelfehler korrigieren (z.B. Distance-4 Hamming Code, CRC-Code).

6.1.2 Fehlerkorrektur

Viele digitale Datenübertragungen wären ohne Fehlerkorrekturverfahren nicht zuverlässig nutzbar (Ethernet, Festplatten, CD, DVD usw.). Fehler können durch elektromagnetische Störungen (Funk, Nähe zu hochspannungsführenden Leitern) oder schlicht durch mechanische Schäden wie Kratzer auf der DVD verursacht werden.

Zur Behebung unvermeidlicher Fehler werden Korrekturverfahren (ECC = Error Checking and Correction) eingesetzt. Dies soll am Beispiel einer Korrektur eines einzelnen Bitfehlers gezeigt werden. Zum Einsatz kommt häufig das CRC-Verfahren (Cyclic Redundancy Check).

Es sollen 8 Bits übertragen werden. Die Stellen der einzelnen Bits werden abweichen von der üblichen Notation mit 1...8 angegeben.

Bit-Position	8	7	6	5	4	3	2	1
Werte (Beispiel)	**1**	**1**	**0**	**0**	**1**	**0**	**0**	**1**

Für die Datenübertragung müssen natürlich Korrekturbits eingefügt werden. Es werden so viele Korrekturbits benötigt, dass jede mögliche Bit-Position adressiert werden kann. In diesem Fall sind 4 Korrektur-Bits die richtige Lösung, da dann 12 Bits übertragen werden müssen und jede Bit-Position über 4 Bits adressierbar ist. Die Korrekturbits müssen an den Stellen eingefügt werden, die eine 2er-Potenz sind (1, 2, 4, 8). Die Datenbits verschieben sich entsprechend nach links.

12	11	10	9	8	7	6	5	4	3	2	1
1	**1**	**0**	**0**	$\mathbf{c_4}$	**1**	**0**	**0**	$\mathbf{c_3}$	**1**	$\mathbf{c_2}$	$\mathbf{c_1}$

Die Werte der Korrektur-Bits $c_4...c_1$ müssen nun ermittelt werden. Hierzu werden alle Adressen, unter denen eine 1 übertragen werden bitweise über XOR verknüpft. Eine 1 wird an den Stellen 3, 7 und 11 übertragen. Somit lautet die XOR Verknüpfung.

```
        0011    ( 3)
xor     0111    ( 7)
xor     1011    (11)
xor     1100    (12)
-------------------
        0011    = c4 c3 c2 c1
```

Diese Bits werden nun eingefügt und ergeben das zu übertragende Datenpaket.

				$\mathbf{c_4}$				$\mathbf{c_3}$		$\mathbf{c_2}$	$\mathbf{c_1}$
12	11	10	9	8	7	6	5	4	3	2	1
1	**1**	**0**	**0**	***0***	**1**	**0**	**0**	***0***	**1**	***1***	***1***

Wir nehmen zunächst an, dass keine Fehler auftritt. Der Empfänger bildet dann die XOR-Verknüpfung aller Bit-Positionen, an denen eine 1 auftritt (ohne Korrektur-Bits!) sowie mit den Korrektur-Bits als letzte Position:

```
        0011    ( 3)
xor     0111    ( 7)
xor     1011    (11)
xor     1100    (12)
xor     0011    (Korrektur-Bits)
-------------------
        0000    = Ergebnis 0 (keine Fehler)
```

Nun soll ein Fehler auftreten. Anstelle von “110001000111” erhält der Empfänger “110001010111”, d.h. die Bit-Position 5 wechselt von 0 auf 1:

				c_4				c_3		c_2	c_1
12	11	10	9	8	7	6	5	4	3	2	1
1	**1**	**0**	**0**	***0***	**1**	**0**	**1**	***0***	**1**	***1***	***1***

Der Empfänger überprüft in gleicher Weise das empfangene Datenpaket:

```
        0011    ( 3)
xor     0101    ( 5)
xor     0111    ( 7)
xor     1011    (11)
xor     1100    (12)
xor     0011    (Korrektur-Bits)
--------------------
        0101    = Ergebnis 5 (Bit 5 ist falsch)
```

Die Vorgehensweise funktioniert auf für die Korrekturbits. Treten zwei Bitfehler auf, kann zwar eine Fehler erkannt werden; eine Korrektur ist dann aber nicht mehr möglich.

Wir nehmen nun an dass auch Bit Nummer 12 falsch wird (0 anstatt 1):

				c_4				c_3		c_2	c_1
12	11	10	9	8	7	6	5	4	3	2	1
1	**0**	**0**	**0**	***0***	**1**	**0**	**1**	***0***	**1**	***1***	***1***

Die Überprüfung durch den Empfänger ergibt:

```
        0011    ( 3)
xor     0101    ( 5)
xor     0111    ( 7)
xor     1100    (12)
xor     0011    (Korrektur-Bits)
--------------------
        1110    = Ergebnis 14 (falsch,
                  keine Korrektur möglich)
```

7 Digitale Schaltungen

Elektronische Digitalschaltungen bestehen überwiegend aus TTL- (Transistor-Transistor Logic) oder CMOS- (Complementary Metal-Oxyde Semiconductor) Bauelementen. Die

logischen Zustände “0” und “1” werden durch kleine bzw. große Spannungen (relativ zur Betriebsspannung von ca. 5 V) abgebildet. Man spricht in diesem Zusammenhang auch von *HIGH* und *LOW*. Man drückt dadurch aus, dass die *logischen Zustände* durch *physikalische Spannungen* repräsentiert werden.

Zuordnung von “0” zu “LOW” und “1” zu “HIGH” = positive Logik

Zuordnung von “0” zu “HIGH” und “1” zu “LOW” = negative Logik

7.1 Verwirklichung digitaler Funktionen

Eine logische Funktion kann nicht nur elektronisch realisiert werden. In der folgenden Tabelle sind gebräuchliche Technologien zusammengestellt.

Technologie	**Bit = 0**	**Bit = 1**
Pneumatik	niedriger Druck	hoher Druck
Relais	Anker abgefallen	Anker angezogen
CMOS	0–1,5V	3,5–5,0V
TTL	0–0,8V	2,0–5,0V
Lichtleiter	kein Licht	Licht an
Dynamischer Speicher	Kondensator entladen	Kondensator geladen
Biploar ROM	Sicherung zerstört	Sicherung intakt
Festplatte/ Magnetband	Flussrichtung Nord	Flussrichtung Süd
CDROM (Read Only)	keine Vertiefung (no Pit)	Vertiefung (Pit)
CD/RW (rewritable)	polykristalline Struktur	monokristalline Struktur

7.2 Logische Signale und Funktionen (Gatter)

Unabhängig von der physikalischen Verwirklichung entwirft man oft ein digitales System als *logisches* System mit *logischen* Signalen, d.h. “0”-“1”-Signalen. Wir ignorieren also das elektrische Verhalten (bei CMOS- oder TTL-Schaltungen) und beschränken uns auf die *diskreten* Signale 0 und 1. Bevor man also eine Schaltung aufbaut, wird zunächst das logische Verhalten entworfen.

Bild 2.1: Logische Schaltung mit drei Eingängen und zwei Ausgängen

Die Funktion der Schaltung wird durch die sogenannte *Wahrheitstabelle* vollständig beschrieben.

Die Wahrheitstabelle beschreibt vollständig eine *kombinatorische* logische Schaltung.

Unter *kombinatorischer* Schaltung versteht man eine Schaltung, bei der die Ausgangsgrößen ausschließlich von den Eingangsgrößen abhängen.

Schaltungen mit internen Speicherelementen heißen sequentielle Schaltungen.

Die sequentiellen Schaltungen sind wesentlich leistungsfähiger. Jeder Computer ist eine sequentielle Schaltung. Eine sequentielle Schaltung enthält aber immer kombinatorische Logik. Es ist deshalb sinnvoll, sich zunächst mit kombinatorischer Logik zu befassen.

Die Schaltung in Bild 2.1 könnte z.B. die einstellige Addition aus Abschnitt 3 sein.

Wahrheitstabelle für eine einstellige duale Addition

Eingänge			Ausgänge	
$x = \text{carry}_{in}$	$y = x_k$	$z = y_k$	$f = x_k + y_k$	$g = \text{carry}_{out}$
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

Beliebig komplexe Schaltungen lassen sich aus wenigen elementaren Logikfunktionen (=Gatter) aufbauen.

7.3 Elementare Gatter

Es werden jeweils die gültigen IEC-Symbole aufgelistet als auch die veralteten amerikanischen Schaltzeichen. Alte deutsche Symbole werden weggelassen. Die alten

amerikanischen Symbole sind auch heute weit verbreitet, insbesondere bei ECAD-Software (Electronic Computer Aided Design).

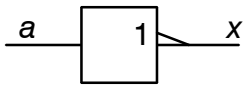
Variablen mit den Zuständen 0 und 1 bezeichnet man als boolsche Variablen.

Die Rechenreglen mit boolschen Variablen heißen boolsche Algebra.

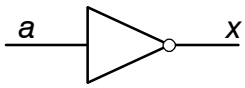
Die folgenden Gatter werden nur mit der minimalen Anzahl von Eingangsgrößen gezeichnet. Der Erweiterung auf mehrere Eingangsgrößen ist jedoch einfach.
Übung: Zeichnen Sie Gatter mit jeweils drei Eingangsgrößen und geben Sie die entsprechende Wahrheitstabelle an.

7.3.1 Inverter (Negation)

Symbol



amerik. Symbol



Wahrheitstabelle

a	x
0	1
1	0

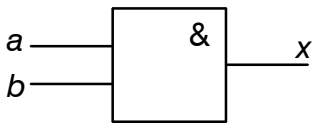
Boolsche Funktion

DIN-Schreibweise: $x = \overline{y}$ (x = nicht a)

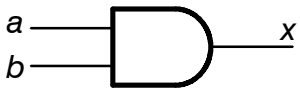
amerik. Schreibweise: $x = a'$ (x = not a)

7.3.2 Konjunktion (AND)

Symbol



amerik. Symbol



Wahrheitstabelle

a	b	x
0	0	0
0	1	0
1	0	0
1	1	1

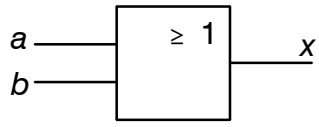
Boolsche Funktion

DIN-Schreibweise: $x = a \wedge b$ (x = a und b)

amerik. Schreibweise: $x = a \cdot b$ ($x = a$ and b)

7.3.3 Disjunktion (OR)

Symbol



amerik. Symbol

Wahrheitstabelle

a	*b*	*x*
0	0	0
0	1	1
1	0	1
1	1	1

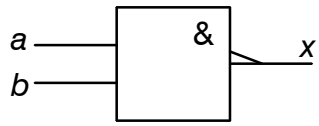
Boolsche Funktion

DIN-Schreibweise: $x = a \vee b$ (x = *a* oder *b*)

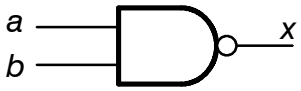
amerik. Schreibweise: $x = a + b$ (*x* = *a* or *b*)

7.3.4 NAND

Symbol



amerik. Symbol



Wahrheitstabelle

a	*b*	*x*
0	0	1
0	1	1
1	0	1
1	1	0

Boolsche Funktion

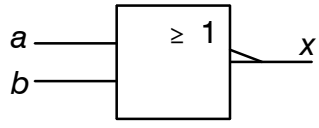
DIN-Schreibweise: $x = \overline{a \wedge b}$

amerik. Schreibweise: $x = (a \cdot b)'$ (*x* = *a* nand *b*)

Mit NAND-Gattern lassen sich beliebige logische Funktionen realisieren.

7.3.5 NOR

Symbol



amerik. Symbol

Wahrheitstabelle

a	*b*	*x*
0	0	1
0	1	0
1	0	0
1	1	0

Boolsche Funktion

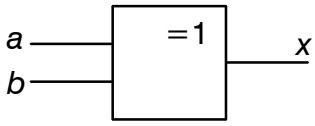
DIN-Schreibweise: $x = \overline{a \vee b}$

amerik. Schreibweise: $x = (a + b)'$ (*x* = *a* nor *b*)

Mit NOR-Gattern lassen sich beliebige logische Funktionen realisieren.

7.3.6 Antivalenz (exklusiv ODER, XOR)

Symbol



amerik. Symbol

Wahrheitstabelle

a	b	x
0	0	0
0	1	1
1	0	1
1	1	0

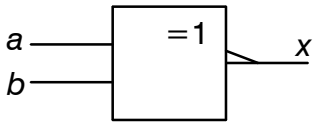
Boolsche Funktion

DIN-Schreibweise: $x = a \oplus b = (\bar{a} \wedge b) \vee (a \wedge \bar{b})$

amerk. Schreibweise: $x = (a \oplus b)$ $(x = a \text{ xor } b)$

7.3.7 Equivalenz (XNOR)

Symbol



amerik. Symbol

Wahrheitstabelle

a	b	x
0	0	1
0	1	0
1	0	0
1	1	1

Boolsche Funktion

DIN-Schreibweise: $x = \overline{a \oplus b} = (\bar{a} \wedge \bar{b}) \vee (a \wedge b)$

amerik. Schreibweise: $x = (a \oplus b)'$ $(x = a \text{ xnor } b)$

7.4 Vorrangregeln für boolsche Algebra

Leider sind die Vorrangregeln für Operationen international nicht einheitlich (die amerikanischen Regeln unterscheidet sich von von DIN 66000). Dies folgt aus der amerikanischen Schreibweise für UND (•) und ODER (+), die der UND-Verknüpfung eine höhere Priorität (Multiplikationszeichen) als der ODER-Verknüpfung (Summenzeichen) einräumt. Die deutsche DIN 66000 sieht UND und ODER als gleichwertig an. In der DIN 660000 müssen also Klammern gesetzt werden, die im Amerikanischen entfallen können.

Wir wollen uns in diesem Skript der amerikanischen Schreibweise anschließen, um bei den späteren Software-Übungen nicht umlernen zu müssen.

Prioritätsregeln

1.	Negation	a'
2.	Konjunktion (UND)	$a \cdot b$

3.	Adjunktion (ODER)	$a + b$
4.	NAND	$(a \cdot b)'$
5.	NOR	$(a + b)'$
6.	Äquivalenz	$a \equiv b$
7.	Antivalenz (XOR)	$a \oplus b$

Im Zweifel kann durch Klammernsetzung immer eine eindeutige Funktion angegeben werden.

7.5 Funktionen für eine Variable (UNARY OPERATORS)

Eingang

$a = 1 \quad 0$

Ausgang x	Symbol	Bezeichnung
0 0	$x = 0$	Konstante 0
1 0	$x = a$	Identität
0 1	$x = a'$	Negation
1 1	$x = 1$	Konstante 1

Bild 2.2: Unary Operators

7.6 Funktionen für zwei Variablen (BINARY OPERATORS)

Eingänge

a = 1 0 1 0
b = 1 1 0 0

Ausgang x	Symbol	Bezeichnung	Grundverknüpfung
0 0 0 0	$x = 0$	Konstante 0	
0 0 0 1	$x = (a + b)'$	NOR	•
0 0 1 0	$x = a \cdot b'$	Inhibition	
0 0 1 1	$x = b'$	Negation (b)	
0 1 0 0	$x = a' \cdot b$	Inhibition	
0 1 0 1	$x = a'$	Negation (a)	
0 1 1 0	$x = a \oplus b$	XOR (Antivalenz)	•
0 1 1 1	$x = (a \cdot b)'$	NAND	•
1 0 0 0	$x = a \cdot b$	UND (Konjunktion	•
1 0 0 1	$x = a \equiv b$	XNOR (Äquivalenz)	•
1 0 1 0	$x = a$	Identität (a)	
1 0 1 1	$x = a + b'$	Implikation	
1 1 0 0	$x = b$	Identität (b)	
1 1 0 1	$x = a' + b$	Implikation	
1 1 1 0	$x = a + b$	ODER (Disjunktion)	•
1 1 1 1	$x = 1$	Konstante 1	

Bild 2.3: Binary Operators

7.7 Übung: Aufstellen einer Wahrheitstabelle

Geben Sie für folgende Schaltung die Wahrheitstabelle an.

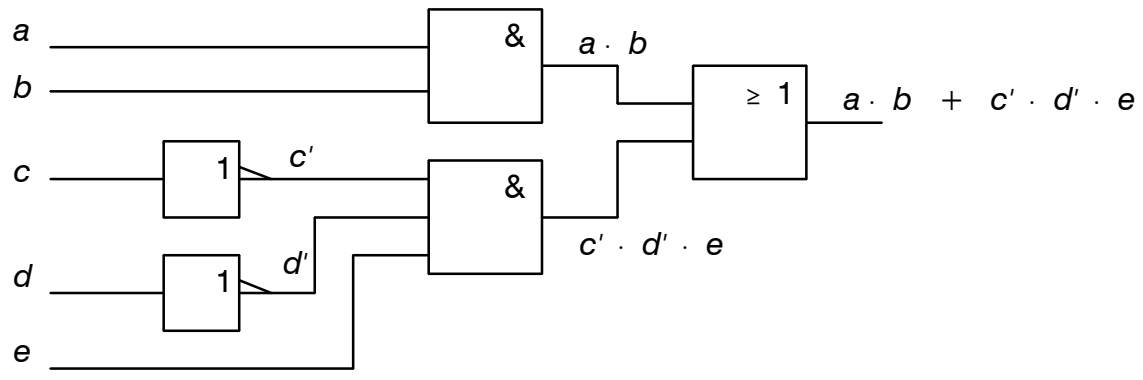


Bild 2.4: Schaltung aus einzelnen Gattern

a	b	c	d	e	$a \cdot b$	$c' \cdot d' \cdot e$	$a \cdot b + c' \cdot d' \cdot e$
0	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	1	1	0	0	0
0	0	1	0	0	0	0	0
0	0	1	0	1	0	0	0
0	0	1	1	0	0	0	0
0	0	1	1	1	0	0	0
0	1	0	0	0	0	0	0
0	1	0	0	1	0	1	1
0	1	0	1	0	0	0	0
0	1	0	1	1	0	0	0
0	1	1	0	0	0	0	0
0	1	1	0	1	0	0	0
0	1	1	1	0	0	0	0
0	1	1	1	1	0	0	0
1	0	0	0	0	0	0	0
1	0	0	0	1	0	1	1
1	0	0	1	0	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	0	0	0	0
1	0	1	0	1	0	0	0
1	0	1	1	0	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	0	1	0	1
1	1	0	0	1	1	1	1
1	1	0	1	0	1	0	1
1	1	0	1	1	1	0	1
1	1	1	0	0	1	0	1
1	1	1	0	1	1	0	1
1	1	1	1	0	1	0	1
1	1	1	1	1	1	0	1

7.8 Rechenregeln für eine Variable und eine Konstante

Die folgenden Rechenregeln ermöglichen oft eine erhebliche Vereinfachung von logischen Schaltungen.

$$x + 0 = x \tag{2.1}$$

$$x + 1 = 1 \tag{2.2}$$

$$x \cdot 0 = 0 \tag{2.3}$$

$$x \cdot 1 = x \tag{2.4}$$

$$x \cdot x = x \tag{2.5}$$

$$x + x = x \tag{2.6}$$

$$x + x' = 1 \tag{2.7}$$

$$x \cdot x' = 0 \tag{2.8}$$

$$x'' = x \tag{2.9}$$

7.9 Verwirklichung einfacher Gatterfunktionen durch mechanische Schalter

Zum Verständnis von Logikschaltungen ist es hilfreich, sich eine mechanische Realisierung vorzustellen. In vielen Industrieanwendungen werden auch einfache logische Funktionen durch Schalter (Öffner und Schließer) verwirklicht.

Bild 2.5: Öffner und Schließer

Durch Drücken des Tasters wird der Schließer geschlossen (Kontakt hergestellt); durch Drücken des Tasters auf dem Öffner wird der Kontakt geöffnet.

Eingänge: Taster nicht gedrückt = 0
Taster gedrückt = 1

Ausgänge: Lampe aus = 0
Lampe leuchtet = 1

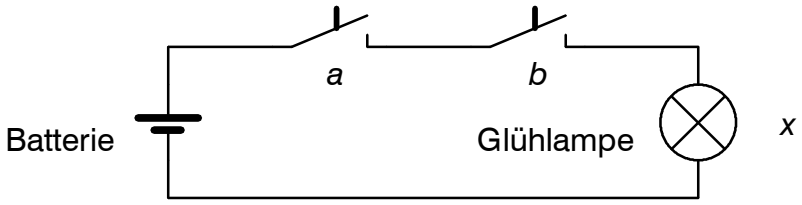


Bild 2.6: UND-Gatter

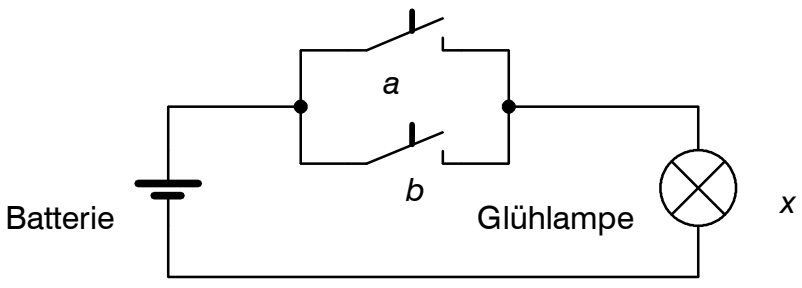


Bild 2.7: ODER-Gatter

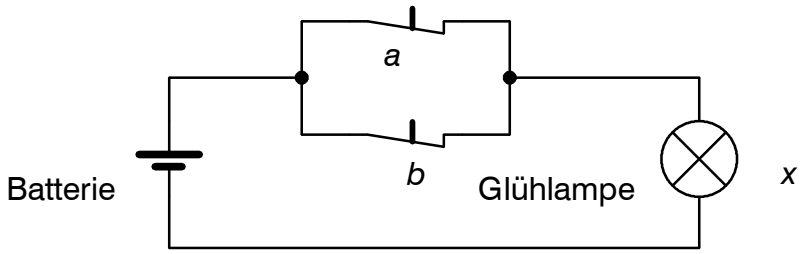


Bild 2.8: NAND-Gatter

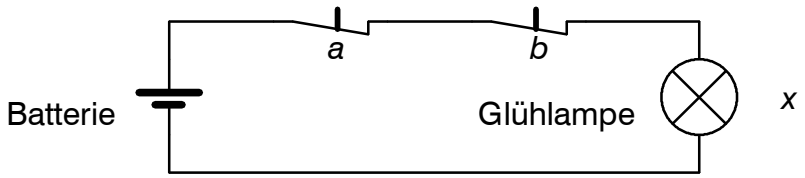


Bild 2.9: NOR-Gatter

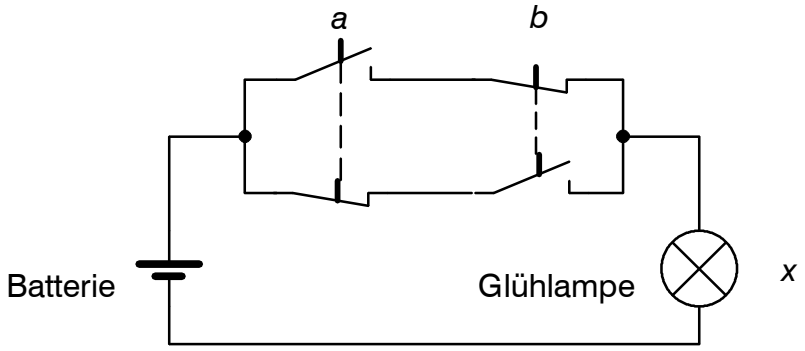


Bild 2.10: XOR-Gatter

Übungen: Verwirklichen andere Gatterfunktionen mit mechanischen Schaltern

8 Rechenregeln für mehrere Variablen (wichtig!)

Die Rechenregeln werden benötigt, um digitale Schaltungen zu vereinfachen, d.h. mit minimalen Anzahl von Gatterfunktionen verwirklichen zu können.

8.1 Kommutativgesetz (Vertauschen von Operanden)

Konjunktion

$$x_1 \cdot x_2 = x_2 \cdot x_1 \tag{3.1}$$

Disjunktion

$$x_1 + x_2 = x_2 + x_1 \tag{3.2}$$

8.2 Assoziativgesetz (Zusammenfassen von Operanden)

Konjunktion

$$x_1 \cdot x_2 \cdot x_3 = x_1 \cdot (x_2 \cdot x_3) \tag{3.3}$$

Disjunktion

$$x_1 + x_2 + x_3 = x_1 + (x_2 + x_3) \tag{3.4}$$

8.3 Distributivgesetz (Verteilen von Operanden)

Konjunktion

$$x_1 \cdot (x_2 + x_3) = (x_1 \cdot x_2) + (x_1 \cdot x_3) \tag{3.5}$$

Disjunktion

$$x_1 + (x_2 \cdot x_3) = (x_1 + x_2) \cdot (x_1 + x_3) \tag{3.6}$$

8.4 DeMorgansche Gesetze (Negationsregeln)

1. DeMorgansches Gesetz

$$(x_1 \cdot x_2 \cdot x_3)' = x_1' + x_2' + x_3' \tag{3.7}$$

2. DeMorgansches Gesetz

$$(x_1 + x_2 + x_3)' = x_1' \cdot x_2' \cdot x_3' \tag{3.8}$$

Beide DeMorgansche Gesetze fasst man zum DeMorganschen Theorem zusammen:

> Die DeMorganschen Gesetze lassen sich leicht merken:
> Inversion des Ausgangs ≡ Inversion aller Eingänge und Vertauschen von UND und ODER.

8.5 Kürzungsregeln

$$x_1 + (x_1 \cdot x_2) = x_1 \tag{3.9}$$

$$x_1 \cdot (x_1 + x_2) = x_1 \tag{3.10}$$

$$x_1 + (x_1' \cdot x_2) = x_1 + x_2 \tag{3.11}$$

$$x_1 \cdot (x_1' + x_2) = x_1 \cdot x_2 \tag{3.12}$$

$$(x_1 \cdot x_2) + (x_1 \cdot x_2') = x_1 \tag{3.13}$$

$$(x_1 + x_2) \cdot (x_1 + x_2') = x_1 \tag{3.14}$$

Übung:

Begründen sie Kürzungsregeln mit Hilfe der vorstehenden Regeln oder der Wahrheitstabelle.

9 Mehrstufige Logik

Komplexe Logikschaltungen werden durch Hintereinanderschaltung vieler Gatter verwirklicht. Die maximale Anzahlen von Gattern die ein Signal durchlaufen muss, bestimmt die “Stufigkeit” *n* (*n*-stufige Logik).

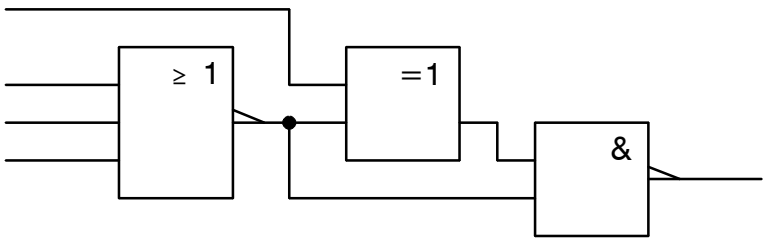


Bild 3.1: 3-stufige Logik

Es addieren sich die Durchlaufzeiten (= Propagation Delays) aller Gatter, d.h. es muss die Summe aller Gatterlaufzeiten abgewartet werden, bis der Ausgang gültig ist. Bei modernen Schaltkreistechnologien liegt die Gatterlaufzeit im Bereich von einigen Nanosekunden (milliardstel Sekunden). Je schneller eine Schaltung sein muss, desto weniger Stufen darf eine Schaltung enthalten.

10 Universelle Schaltungen mit NAND- oder NOR-Gattern

Diese Gattertypen finden sich sehr häufig, da mit NAND- oder NOR-Gattern beliebige kombinatorische Schaltungen aufbauen lassen. Diese Universalität folgt aus dem DeMorganschen Theorem.

10.1 Ungenutzte Eingänge

Abhängig von der Schaltkreistechnologie müssen ungenutzte Eingänge beschaltet werden, da anderenfalls Fehlfunktionen auftreten können. Als Grundregel sollte man berücksichtigen:

> Ungenutzte Eingänge sollten nicht “offen” belassen werden.

Beispiel: Inverter mit NAND- oder NOR-Gatter

Da mit NAND und NOR beliebige Funktionen verwirklicht werden können, sind natürlich auch Inverter möglich. Hierbei werden ein oder mehrere Eingänge der Gatter nicht benötigt, die aber in geeigneter Weise eindeutig beschaltet werden sollten.

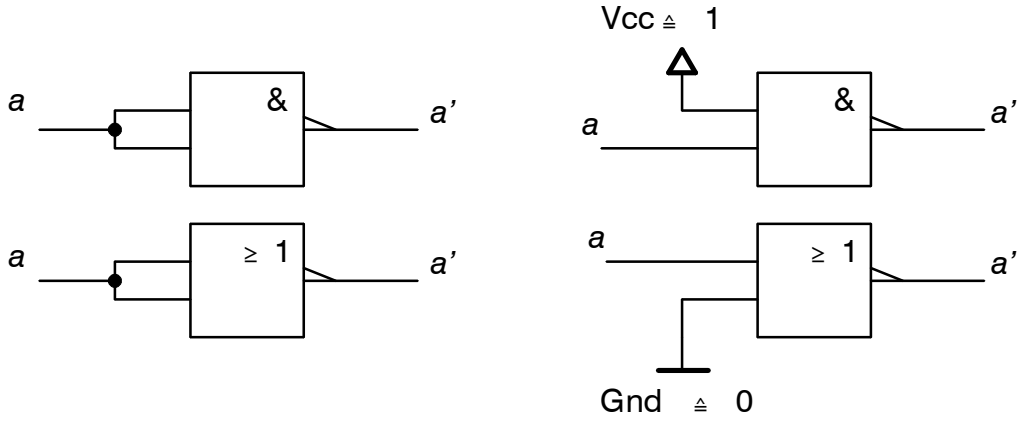


Bild 3.2: Inverter aus NAND- oder NOR-Gatter

Der Nachweis lässt sich aus den Rechenregeln 7.8 erbringen.

$$(a \cdot a)' = a' \tag{3.15}$$

$$(a + a)' = a' \tag{3.16}$$

$$(1 \cdot a)' = a' \tag{3.17}$$

$$(a + 0)' = a' \tag{3.18}$$

Auf diese Weise lassen sich Gatterschaltkreise, die oft mehrere unabhängige Gatter enthalten, besser ausnutzen.

10.2 AND- und OR-Funktion mit NAND und NOR

Die Verwirklichung von AND- und OR-Funktionen mit NAND und NOR ist einfach, da lediglich ein Inverter benötigt wird, der sich auch NAND und NOR bilden lässt (s. 10.1).

10.3 Verwirklichung von OR-Gattern mit NAND

Hier wird das DeMorgansche Theorem benötigt:

$$(x_1 \cdot x_2)' = x_1' + x_2' \,. \tag{3.19}$$

Es müssen also die *Eingänge* x_1 und x_2 invertiert werden und in ein NAND-Gatter gegeben werden.

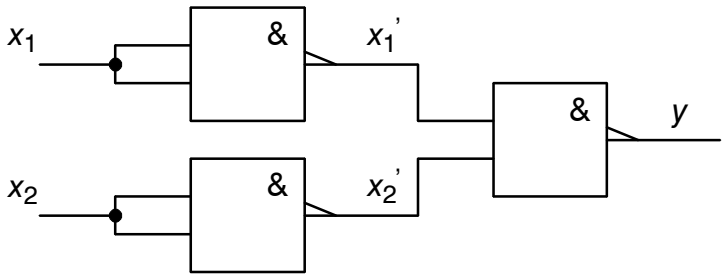


Bild 3.3: Verwirklichung der OR-Funktion mit NAND-Gattern

Beweis:

$$y = \left((x_1 \cdot x_1)' \cdot (x_2 \cdot x_2)'\right)' = (x_1' \cdot x_2')' = x_1 + x_2 \,. \tag{3.20}$$

10.4 Verwirklichung von AND-Gattern mit NOR

Hier wird ebenfalls das DeMorgansche Theorem angewandt:

$$(x_1 + x_2)' = x_1' \cdot x_2' \,. \tag{3.21}$$

Es müssen also die *Eingänge* x_1 und x_2 invertiert werden und in ein NOR-Gatter gegeben werden.

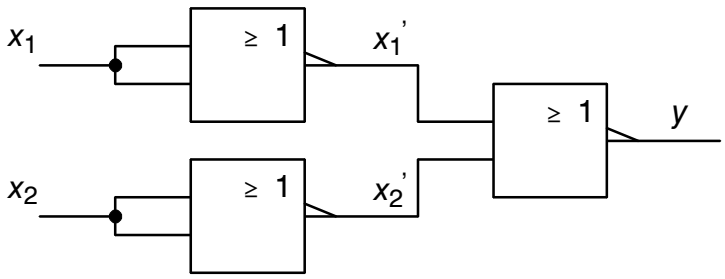


Bild 3.4: Verwirklichung der AND-Funktion mit NOR-Gattern

Beweis:

$$y = \left((x_1 + x_1)' + (x_2 + x_2)'\right)' = (x_1' + x_2')' = x_1 \cdot x_2 \,. \tag{3.22}$$

11 Normalformen

Die Normalformen sind der Schlüssel zur Verwirklichung komplexer Schaltungen mit einer minimalen Anzahl von Gattern. Viele *programmierbare* Logikbausteine basieren auf einer Darstellung von Logikfunktionen in Normalform. Die Normalformen – auch *kanonische Normalformen* genannt – lassen sich relativ leicht mit Computerprogrammen automatisch *minimieren* (= optimieren im Sinne einer minimalen Anzahl von Gattern).

Minterm: konjunktive Verknüpfung von Variablen (UND)

Maxterm: disjunktive Verknüpfung von Variablen (ODER)

Man unterscheidet zwei (völlig gleichwertige) Normalformen

Disjunktive Normalform (DNF): disjunktive Verknüpfung von Mintermen

Konjunktive Normalform (KNF): konjunktive Verknüpfung von Maxtermen

Aus unerklärlichen Gründen findet man die DNF häufiger. Wir werden deshalb ebenfalls die DNF-Darstellung bevorzugen. Beide Normalformen können jedoch über das DeMorgansche Theorem leicht ineinander überführt werden. Es gibt jedoch auch Gründe die eine oder andere Form zu wählen (Anzahl der Nullen oder Einsen in der Wahrheitstabelle).

x3	x2	x1	y	y'
0	0	0	0	1
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	0	1

Bild 3.5: Wahrheitstabelle für eine logische Funktion

Für wollen die DNF und KNF für die Wahrheitstabelle 3.5 aufstellen.

Die DNF berücksichtigt nur die Ausgänge mit “1”

Die disjunktive Zusammenfassung der “1”-Minterme liefert die DNF

$$y = x_3' \cdot x_2' \cdot x_1 \;+\; x_3' \cdot x_2 \cdot x_1' \;+\; x_3 \cdot x_2' \cdot x_1' \;. \qquad (3.23)$$

Die KNF berücksichtigt nur die Ausgänge mit “0”

Die konjunktive Zusammenfassung der “0”-Maxterme (Eingangsvariablen invertieren!) liefert die KNF

$$y = (x_3 + x_2 + x_1) \cdot (x_3 + x_2' + x_1') \cdot (x_3' + x_2 + x_1') \cdot (x_3' + x_2' + x_1) \cdot (x_3' + x_2' + x_1') . \quad (3.24)$$

Invertiert man (3.24) zweimal, so gilt die Gleichung natürlich weiterhin

$$y = \Big(\big((x_3 + x_2 + x_1) \cdot (x_3 + x_2' + x_1') \cdot (x_3' + x_2 + x_1') \cdot (x_3' + x_2' + x_1) \cdot (x_3' + x_2' + x_1')\big)' \Big)' . \quad (3.25)$$

Wendet man nun das DeMorgansche Theorem an

$$y = (x_3' \cdot x_2' \cdot x_1' + x_3' \cdot x_2 \cdot x_1 + x_3 \cdot x_2' \cdot x_1 + x_3 \cdot x_2 \cdot x_1')' , \quad (3.26)$$

so erkennt man, das man dies auch als DNF für y' auffassen kann (s. Bild 3.5)

$$y' = x_3' \cdot x_2' \cdot x_1' + x_3' \cdot x_2 \cdot x_1 + x_3 \cdot x_2' \cdot x_1 + x_3 \cdot x_2 \cdot x_1' . \quad (3.27)$$

KNF = DNF für y'

DNF und KNF beschreiben das Verhalten vollständig und richtig. In diesem Beispiel ist die DNF günstiger, da weniger Terme auftreten.

Wenig Einsen → DNF, wenig Nullen → KNF

Die obige Grundregel gilt für die Verwirklichung von y. Man kann jedoch y' realisieren (mit anschließender Inversion), so dass man im Prinzip mit einer Normalform auskommt.

Bei der Verwirklichung von y' finden sich weniger Nullen als Einsen, was nun eine Darstellung in KNF nahelegt:

$$y' = (x_3 + x_2 + x_1') \cdot (x_3 + x_2' + x_1) \cdot (x_3' + x_2 + x_1) . \quad (3.28)$$

Man benötigt lediglich einen Inverter am Ausgang, um wieder y erzeugen zu können. Die zugehörige Schaltung zeigt Bild 3.6.

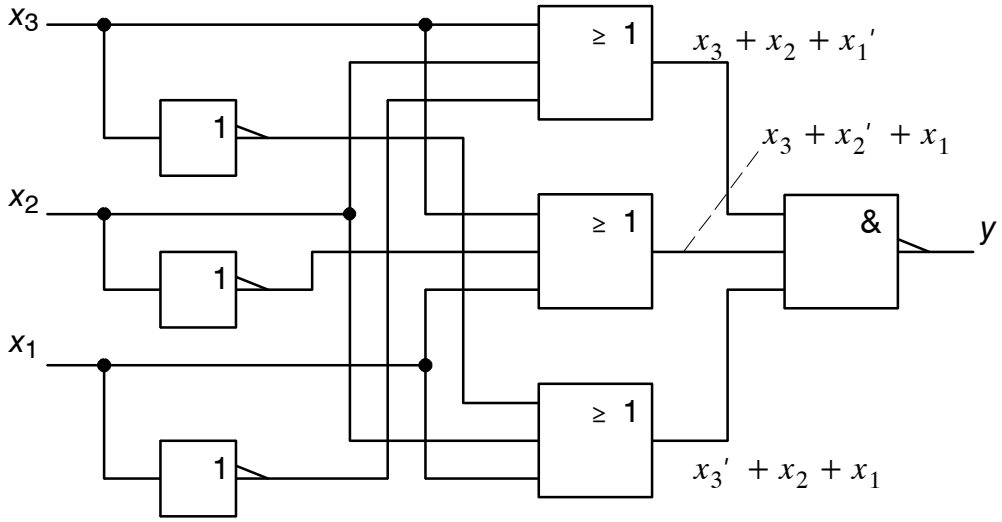


Bild 3.6: Logikschaltung zur Wahrheitstabelle 3.5 für *y'* (KNF)
(durch NAND-Gatter anstelle von AND entsteht *y*)

12 Minimieren (Optimieren) von Digitalschaltungen

Umfangreiche Digitalschaltung lassen sich algorithmisch am Rechner minimieren. Alle modernen Programme zur Synthese digitaler Schaltungen verfügen über leistungsfähige Algorithmen zur Minimierung des Schaltungsaufwandes (z.B. der Quine-McCluskey-Algorithmus). Kleinere Schaltungen können mit einem grafischen Verfahren, dem Karnaugh-Veitch-Diagramm (Karnaugh-Map) minimiert werden.

Karnaugh-Diagramm: grafische Minimierung von Digitalschaltungen mit bis zu vier Variablen

Im KV-Diagramm lässt sich das Distributivgesetz 8.3 grafisch zur Vereinfachung auswerten. Die Gleichung

$$y = x_1 \cdot x_2 + x_1 \cdot x_2' \tag{3.29}$$

kann durch "Ausklammern" auf die Form

$$y = x_1 \cdot \left(x_2 + x_2'\right) = x_1 \cdot 1 = x_1 \tag{3.30}$$

gebracht werden. Das Umformen boolscher Gleichungen ist unübersichtlich und führt folglich nicht mit Sicherheit auf eine minimale Realisierung.

12.1 KV-Diagramm für zwei Variablen

x2	x1	y
0	0	0
0	1	1
1	0	0
1	1	1

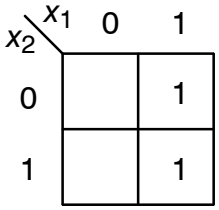


Bild 3.7: Wahrheitstabelle und zugehöriges Karnaugh-Diagramm für 2 Variablen x_1 und x_2

Im KV-Diagramm werden an den entsprechenden Stellen nur Einsen (für die DNF) bzw. nur Nullen (für die KNF) eingetragen. Die zusammenhängenden Gebiete werden markiert.

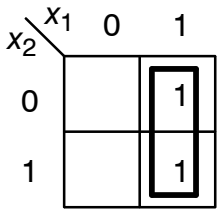


Bild 3.8: Markierung zusammenhängender Gebiete im KV-Diagramm

Bei jedem Übergang von einer "Zelle" des KV-Diagramm zur nächten "Zelle" ändert sich genau eine Variable. Jede "1" entspricht genau einem Minterm.

> Ändert sich eine Variable in einem zusammenhängenden Gebiet, so kann die betreffende Variable aus den Mintermen entfernt werden.

In Bild 3.8 ändert sich die Variable x_2 in dem zusammenhängenden Gebiet; folglich kann sie bei der Realisierung entfallen

$$y = x_1 \,. \tag{3.31}$$

Bei zwei Variablen lohnt sich das Zeichnen des KV-Diagramms meist nicht. Dies ändert sich bei 3 oder 4 Variablen. Eine größere Anzahl von Variablen würde dreidimensionale KV-Diagramme erfordern, die sich nicht mehr einfach zeichnen lassen.

12.2 KV-Diagramm für drei Variablen

Das KV-Diagramm für drei Variablen erschient etwas unmotiviert. Es muss jedoch eine Form gefunden werden, bei der vom Übergang von einer Zelle zur benachbarten Zelle sich jeweils nur eine Variable ändert

x3	x2	x1	y
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	0	0
1	1	1	1

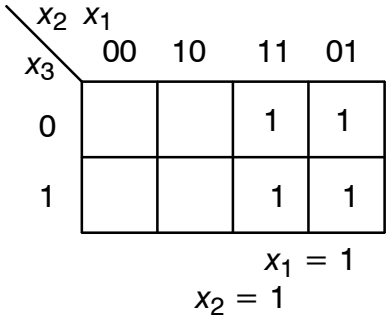


Bild 3.9: Wahrheitstabelle und zugehöriges Karnaugh-Diagramm für die Variablen x_1, x_2 und x_3

In Bild 3.9 erkennt man ein zusammenhängendes Gebiet aus vier Einsen. Dieses Gebiet ist in Bild 3.10 markiert.

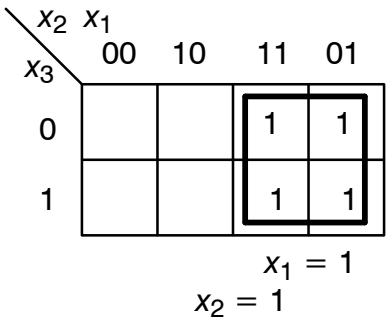


Bild 3.10: Markierung eines zusammenhängenden Gebiets

Es ändern sich die Variablen x_2 und x_3. Folglich brauchen diese Variablen auch nicht berücksichtigt werden; es gilt

$$y = x_1 \,. \tag{3.32}$$

(Nachweis z.B. anhand der Wahrheitstabelle.)

12.3 KV-Diagramm für vier Variablen

Logische Funktionen mit mehr als vier Variablen lassen sich in einem KV-Diagramm nicht mehr darstellen. Auch hier muss eine Form gefunden werden, bei der beim Übergang von einer Zelle zur nächsten Zelle sich nur jeweils eine Variable ändert. Eine mögliche Realisierung mit einem Beispiel zeigt das Bild 3.11.

x4	x3	x2	x1	y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

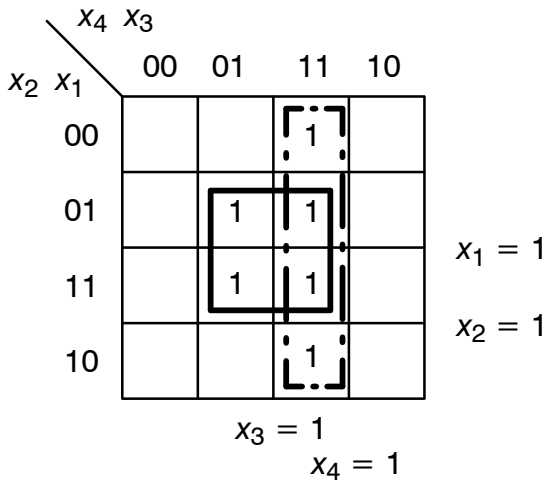


Bild 3.11: Wahrheitstabelle und zugehöriges Karnaugh-Diagramm für die Variablen x_1, x_2, x_2 und x_4

In Bild 3.11 sind die zusammenhängenden Gebiete markiert. Da es nur zwei zusammenhängende Gebiete gibt, lässt sich die logische Schaltung von ursprünglich 6 Mintermen (s. Wahrheitstabelle, 6 Einsen) auf 2 Minterme (Gebiete) reduzieren, was einer dramatischen Vereinfachung entspricht.

Das senkrecht markierte Gebiet beschreibt den Minterm

$$y_1 = x_3 \cdot x_4 \,, \tag{3.33}$$

da sich die Variablen x_1 und x_2 ändern. Entsprechend folgt für das quadratische Gebiet (hier ändern sich die Variablen x_2 und x_4)

$$y_2 = x_1 \cdot x_3 \,. \tag{3.34}$$

Die Disjunktion (OR) der Minterme beschreibt vollständig die logische Funktion

$$y = y_1 + y_2 = x_3 \cdot x_4 \;+\; x_1 \cdot x_3 \,. \tag{3.35}$$

Intuitiv findet man diese Lösung im allgemeinen nicht.

> Zusammenhängende Gebiete finden sich auch “über die Grenzen der Karnaugh-Diagramms” hinweg.

Beispiel

x4	x3	x2	x1	y
0	0	0	0	0
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	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

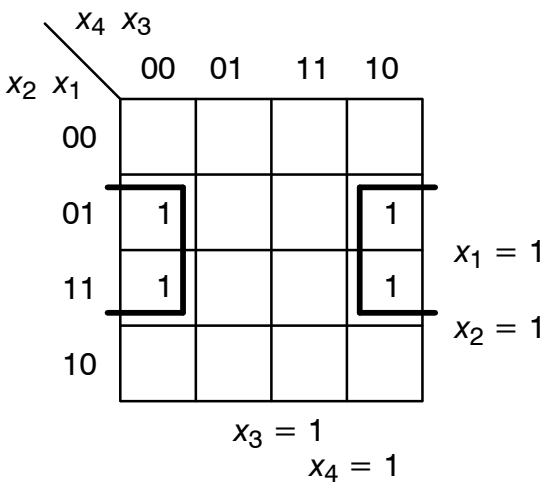


Bild 3.12: Beispiel zur Auffindung zusammenhängender Gebiete

Die markierten Bereiche bilden ein *zusammenhängendes* Gebiet im Sinne der Minimierung von Mintermen, da sich auch beim dem übergang vom rechten zum linken Rand nur ein Bit ändert. Da es sich also nur um ein zusammenhängendes Gebiet handelt, wird die logische Funktion also auch nur durch einen einzigen Minterm beschrieben

$$y = x_1 \cdot x_3' \tag{3.36}$$

(es ändern sich in dem Gebiet ja die Variablen x_2 und x_4). Die logische Funktion der Wahrheitstabelle nach Bild 3.12 kann also durch die folgende einfache Schaltung verwirklicht werden:

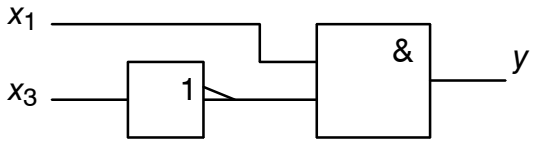


Bild 3.13: Schaltungstechnische Realisierung

Die gleichen Zusammenhänge gelten natürlich auch für den oberen und unteren Rand eines KV-Diagramms.

13 Minimale Anzahl von Produkttermen

Zur Verwirklichung von Schaltungen mit Gattern oder programmierbaren Bauelementen (PLDs = Porgrammable Logic Devices) ist es vorteilhaft, die minimale Funktion zu finden.

Hierzu kann das KV-Diagramm verwendet werden. In vielen Fällen bilden jedoch die Gebiete mit zusammenhängenden Einsen (oder Nullen) nicht die minimale Form.

Eine systematische Methode zum Auffinden einer minimalen Verwirklichung kann über die sogenannten *Essential Prime-Implicants* erfolgen.

> Distinguished 1-Cells sind Zellen im KV-Diagramm, die nur einem Gebiet zugeordnet sind.

> Essential Prime-Implicants sind Gebiete, die eine Distinguished 1-Cell umfassen.

Das folgende Beispiel soll die Vorgehensweise verdeutlichen.

x4	x3	x2	x1	y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

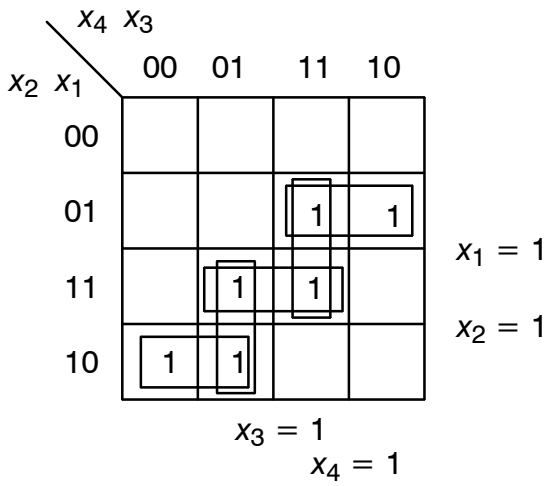


Bild 4.1: Beispiel zur minimalen Verwirklichung einer logischen Funktion

Es existieren 6 Gebiete, so dass die logische Funktion fehlerfrei als

$$y = x_4' \cdot x_2 \cdot x_1' + x_4' \cdot x_3 \cdot x_2 + x_3 \cdot x_2 \cdot x_1 + x_4 \cdot x_3 \cdot x_1 + x_4 \cdot x_2' \cdot x_1 \quad (4.1)$$

verwirklicht werden kann. Diese Lösung ist jedoch nicht minimal. Die Minimale Lösung kann über die Distinguished 1-Cells gefunden werden (in Bild 4.2) markiert.

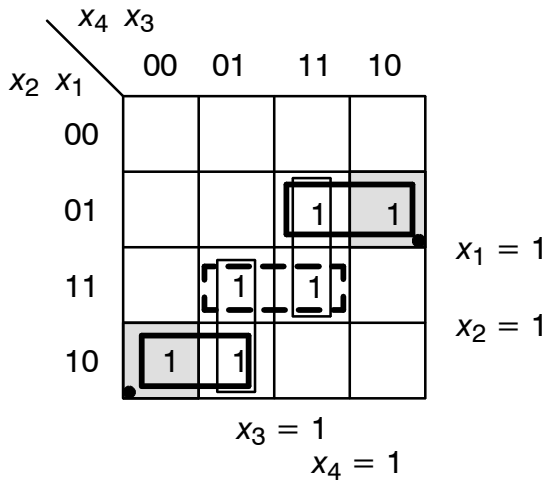


Bild 4.2: Distinguished 1-Cells und Prime-Implicants

Die zugehörigen Essential Prime-Implicants sind ebenfalls hervorghoben. Sie müssen auf jeden Fall realisiert werden, da die Prime-Implicants nicht durch andere Gebiete abgedeckt werden.

Nun müssen lediglich die Gebiete berücksichtigt werden, die die Essential Prime-Implicants nicht abgedeckt sind. In unserem Fall ist das der gestrichelt gekennzeichnete Bereich, d.h.

$$x_3 \cdot x_2 \cdot x_1 \,. \tag{4.2}$$

Die minimale Lösung besteht dann nur aus den 3 Mintermen

$$y = x_4' \cdot x_2 \cdot x_1' \; + \; x_3 \cdot x_2 \cdot x_1 \; + \; x_4 \cdot x_2' \cdot x_1 \,. \tag{4.3}$$

13.1 Don't Cares

Können bei logischen Funktionen bestimmte Eingangskombinationen ausgeschlossen werden, sind die Ausgänge natürlich nicht definiert. Man spricht in diesem Fall von *Don't Cares*. Für die Minimierung können die Ausgänge dann beliebig als 0 oder 1 angenommen werden, um möglichst einfache Verwirklichungen zu erhalten. Im KV-Diagramm werden Don't Cares als "X" gekennzeichnet. Das folgende (Übungs-) Beispiel enthält Don't Cares.

13.2 Ansteuerung einer 7-Segment-Anzeige (Seven-Segment-Decoder)

In Anzeigen für Taschenrechner, Uhren oder Hinweistafeln werden Zahlen häufig mit Hilfe von 7-Segment-Anzeigen dargestellt. Hierzu wird ein BCD–7-Segment-Dekoder eingesetzt.

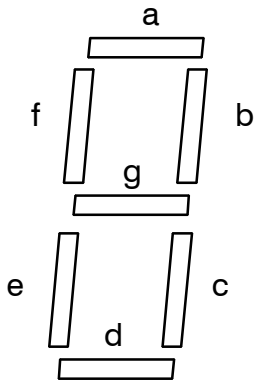


Bild 4.3: 7-Segment-Anzeige

Durch Ansteuerung der einzelnen Segmente *a-g* lassen sich gut lesbar die Zahlen 0-9 darstellen.

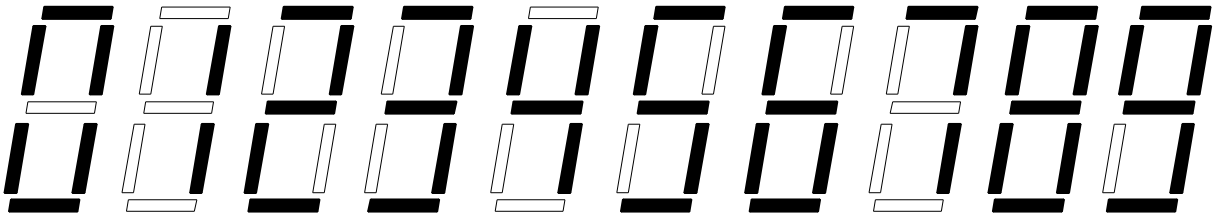


Bild 4.4: 7-Segment-Anzeige

Die Eingangsgrößen für die logische Funktion sind Zahlen im BCD-Format (4 Bit). Ausgänge sind die Ansteuersignale für die einzelnen Segmente. Die logische Funktion besitzt also 4 Eingangs- und 7 Ausgangsgrößen. Da nur die Eingangswerte “0000”–”1001” (0...9) für die Anzeige sinnvolle Eingangsgrößen darstellen, können die Werte “1010”–”1111” (10–15, also keine Ziffern) als “Don’t Care” aufgefasst werden.

Die minimale Verwirklichung aller boolschen Gleichung entspricht dem Innenleben des ICs 74xx49.

14 Hazards (static-0 and static-1 Hazards)

Hazards sind kurzfristig falsche Ausgänge, die durch Änderung von einer Eingangsvariablen verursacht werden. Ein static-0 oder static-1 Hazard ist die einfachste Form eines Hazards.

Ein static-0 Hazard erzeugt kurzzeitig eine “1”, obwohl die “0” der stationäre richtige Wert ist.

Ein static-1 Hazard erzeugt kurzzeitig eine “0”, obwohl die “1” der stationäre richtige Wert ist.

Wir wollen die Hazard-Problematic nur anreißen, indem wir die einfachste Form beschreiben. Hazards (ganauer: Timing-Hazards) entstehen aufgrund unterschiedlicher Signallaufzeiten in einem Gatternetzwerk. Hierzu betrachten wir folgendes Beispiel:

x3	x2	x1	y	
0	0	0	0	
0	0	1	1	
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	1	•
1	1	0	1	
1	1	1	1	•

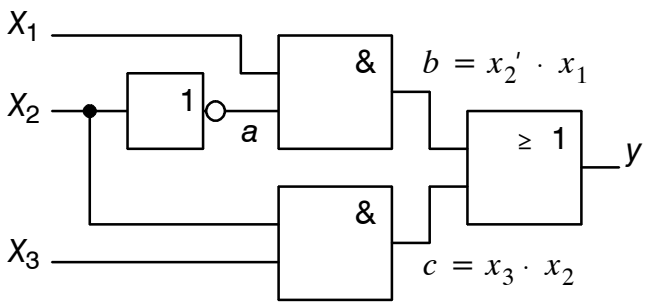


Bild 4.5: Enstehung eines Hazards

Beide markierten Einträge der Wahrheitstabelle liefern unabhängig von x_2 den Wert 1. Sehen wir uns den *dynamischen* Verlauf der Ausgangsgröße als Folge einer Änderung der Eingangsgröße x_2 von 1 auf 0 an, so ergibt sich das folgende Bild 4.6.

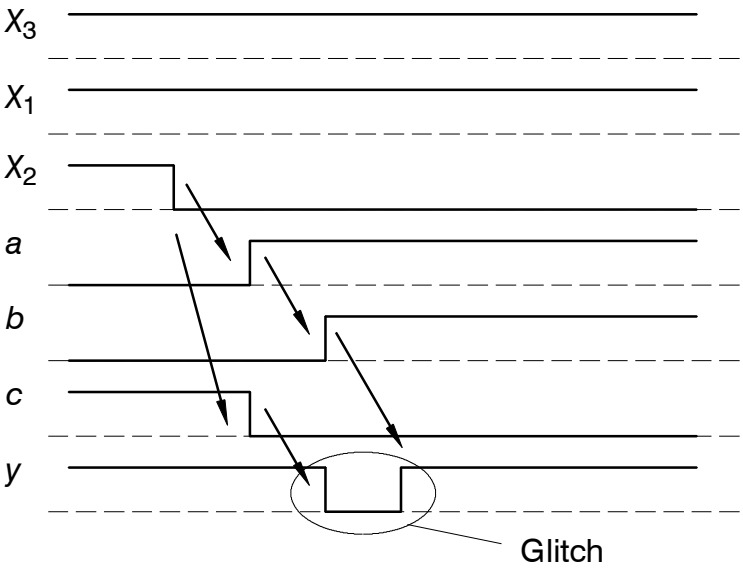


Bild 4.6: Timing-Analyse der Schaltung 4.5 (static-1 Hazard)

Unter das Annahme gleicher Gatterlaufzeiten zeigt das Signal y kurzzeitig einen falschen Wert (Breite: eine Gatterlaufzeit). Man bezeichnet diese typische Kurvenform als Glitch. Bei realen Schaltungen muss dieser Effekt berücksichtigt werden, indem das Ausgangssignal y erst nach der Summe aller Gatterlaufzeiten für weitere Operationen verwendet wird.

14.1 Vermeidung von Hazards (Glitches)

Betrachten wir das KV-Diagramm der Schaltung 4.7, so erkennen wir die zwei Minterme, die der Schaltung entsprechen.

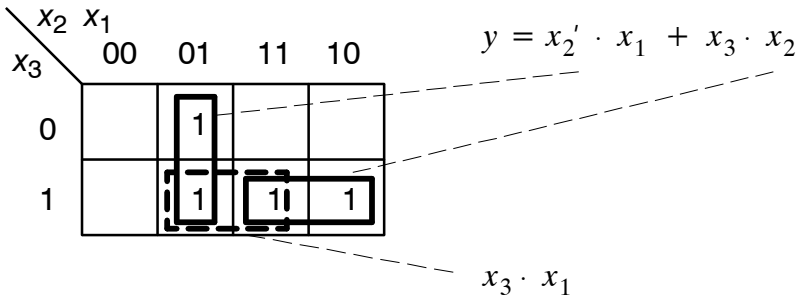


Bild 4.8: KV-Diagramm zu Bild 4.5

Der Hazard tritt beim Übergang von x_2: $1 \rightarrow 0$ auf. Wird nun ein zusätzlicher Minterm eingefügt, der diesen Übergang erfasst (gestrichelt gekennzeichnet), so kann kein Glitch mehr auftreten. Die Schaltung ist dann allerdings nicht mehr minimal.

Die boolsche Funktion lautet dann

$$y = x_2' \cdot x_1 + x_3 \cdot x_2 + x_3 \cdot x_1 . \tag{4.4}$$

Die um den dritten Minterm erweiterte Schaltung zeigt Bild 4.9.

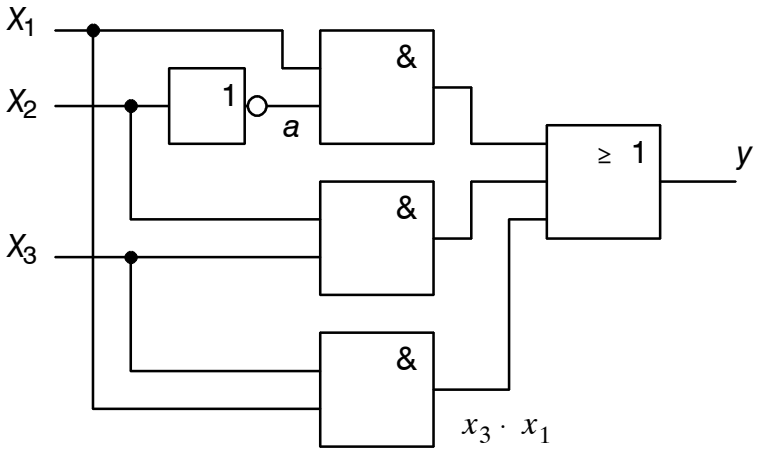


Bild 4.9: Vermeidung eines static-1 Hazards

15 Labore Digitale Schaltungen

Im Rahmen dieses Labors soll eine Aufgabe mit Gatterschaltungen in minimaler Form (DNF und KNF) gelöst werden. Nach der Minimierung der logischen Funktionen kann die Schaltung durch Verbinden von Gattern verwirklicht werden. Hierzu stehen genug Gatterfunktionen (AND, OR, NAND, NOR) und Verbindungsleitungen zur Verfügung.

Lab #01:

15.1 Labor #01a: Addierer (2 Bit)

Ein Addierer für unsigned und signed Integer-Arithmetik ist zu überprüfen und in Hardware (kombinatorische Gatter-Logik) aufzubauen. Da die zur Verfügung stehende Hardware-Schaltung nur über eine eingeschränkte Anzahl an Gattern verfügt, wird der Addierer auf 2 Bit beschränkt.

Bei einer 2-Bit-Addition kann das Ergebnis dreistellig werden. In diesem Fall kann ein Überlauf (Overflow) nicht auftreten. Das Blockschaltbild 3.14 zeigt die Struktur des Addierers.

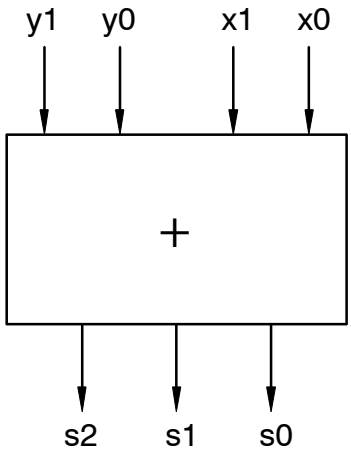


Bild 3.14: 2-Bit-Addierer

Eine Umsetzung des Addierers mit UND- und ODER-Gattern zeigt Bild 3.15

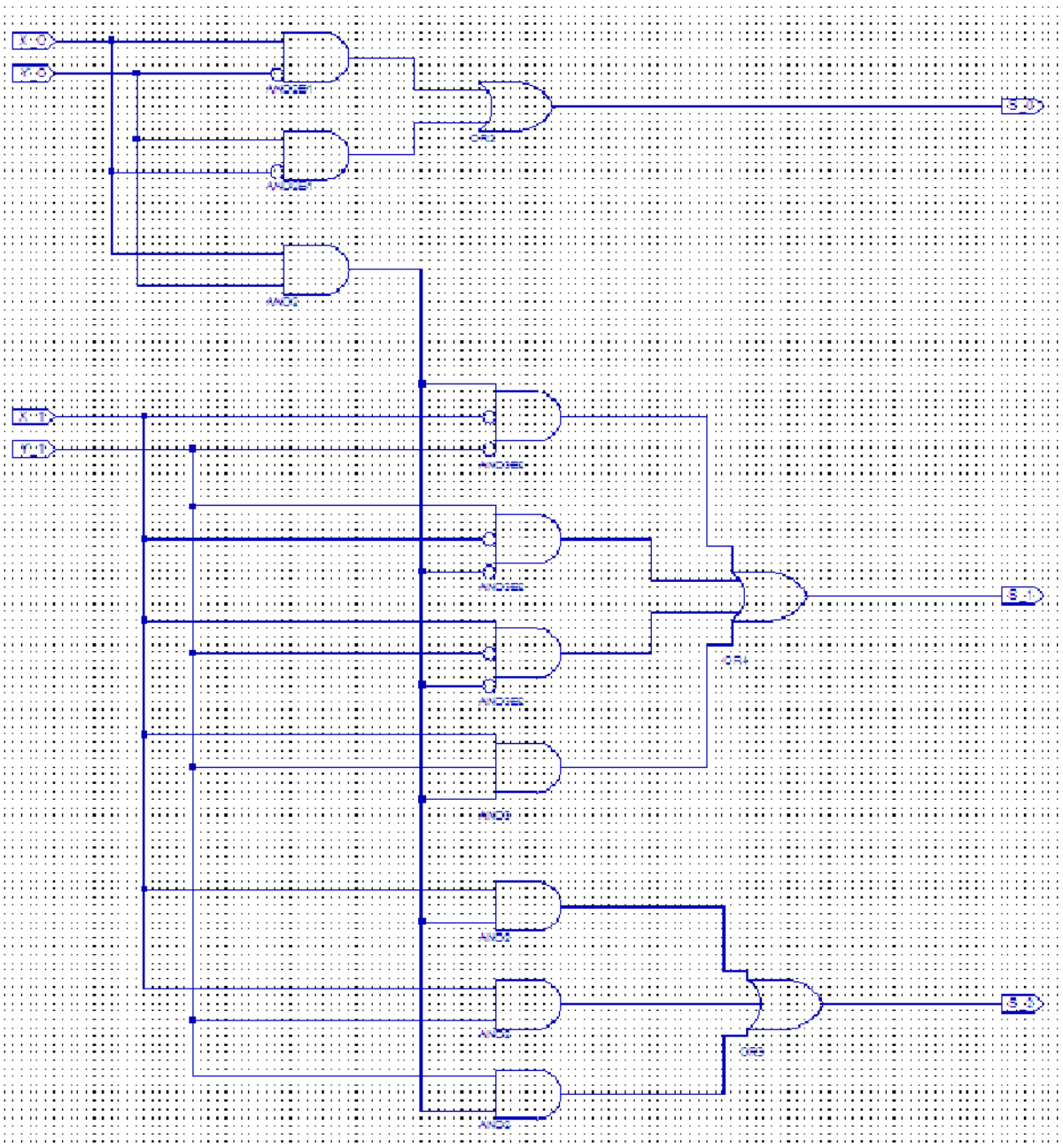
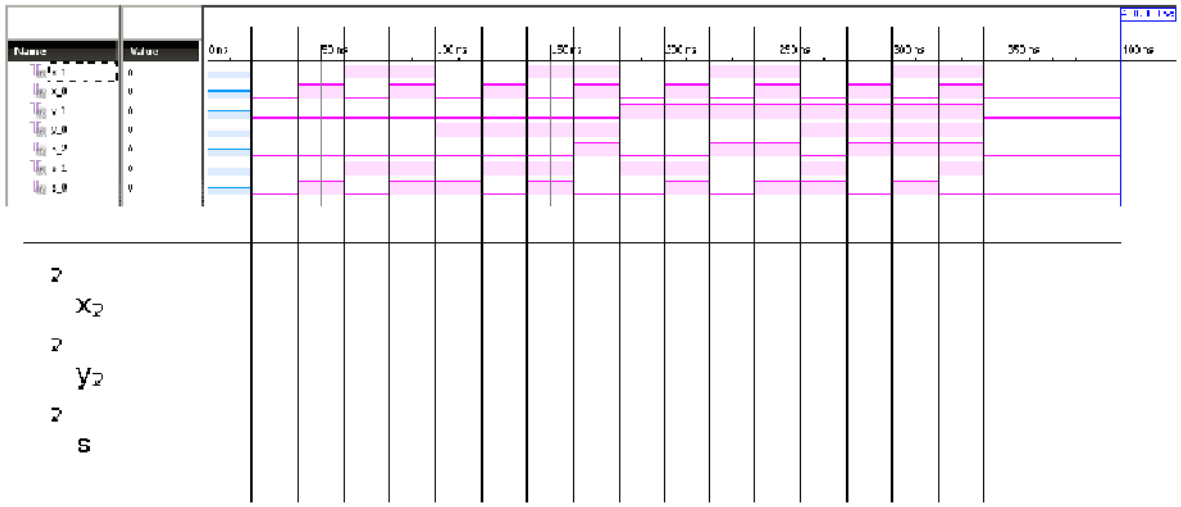


Bild 3.15: Addierer mit UND- und ODER-Gattern (2 Bit)

- Überprüfen Sie die Funktion der Schaltung durch Aufstellen der Wahrheitstabelle. Die Wahrheitstabelle soll aus der Schaltung hergeleitet werden.

y1	y0	x1	x0	s2	s1	s0	unsigned sum
0	0	0	0	0	0	0	0 + 0 = 0
0	0	0	1				
0	0	1	0				
0	0	1	1				
0	1	0	0				
0	1	0	1				
0	1	1	0				
0	1	1	1				
1	0	0	0				
1	0	0	1				
1	0	1	0				
1	0	1	1				
1	1	0	0				
1	1	0	1				
1	1	1	0				
1	1	1	1	1	1	0	3 + 3 = 6

▶ Ordnen Sie den Signalen Dezimalzahlen zu (unsigned).



- Bauen Sie die Schaltung auf und überprüfen Sie die Funktion.

Lab #02:

15.2 Labor #02: Bargraph

Eine Bargraph-Anzeige wird zur quasi-analogen Darstellung digitaler Zahlen eingesetzt. Eine solche Anzeige für 3-Bit-Zahlen soll in minimaler Form (in **d**isjunktiver **N**ormal**f**orm bzw. **k**onjunktiver **N**ormal**f**orm) verwirklicht werden. Die Ausgabe erfolgt mit LEDs (Light Emitting Diode), die unmittelbar von den Ausgängen einiger Gatter angesteuert werden können. Der Zusammenhang zwischen Ansteuerung der LEDs und der dualen Zahl zeigt Bild 2.11.

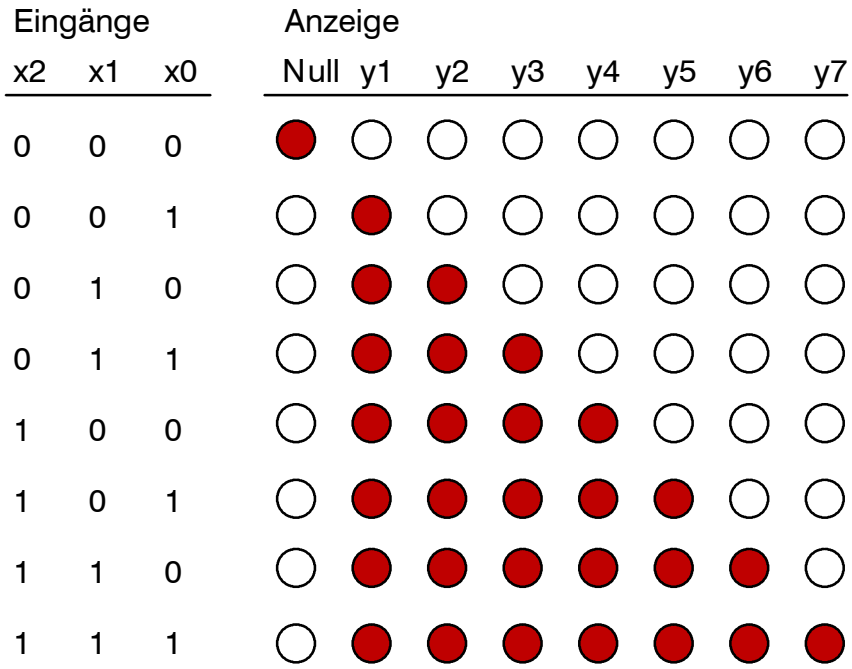
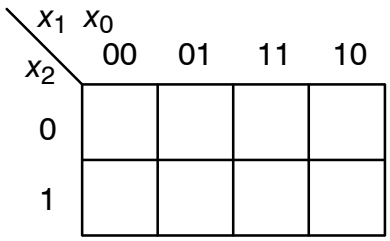


Bild 2.11: Bargraph für 3 Bit

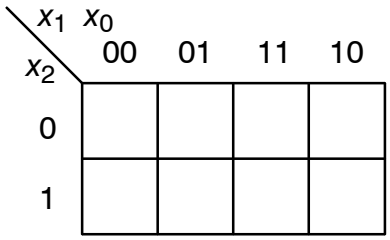
Jeder leuchtenden LED wird der logische Wert '1' zugeordnet.

- Tragen Sie die entsprechenden Nullen und Einsen in das Karnaugh-Diagramm ein (je ein Diagramm für jede Ausgangsgröße).
- Wählen Sie die DNF oder die KNF (wann ist was günstiger?).
- Kennzeichnen Sie die Gebiete, die Sie zu einem Minterm oder Maxterm zusammenfassen.
- Geben Sie die minimale boolsche Gleichung für den jeweiligen Ausgang an.
- Wenden Sie ggf. das DeMorgansche Theorem an, um möglichst einfache Gleichungen zu erhalten oder vorhandene Gatter möglichst gut auszunutzen.



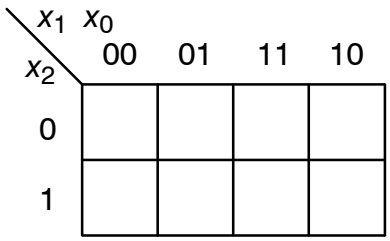
Null =

Bild 2.12: Karnaugh-Diagramm für Null



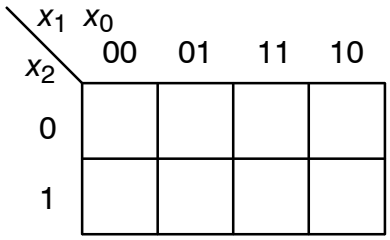
y_1 =

Bild 2.13: Karnaugh-Diagramm für y_1



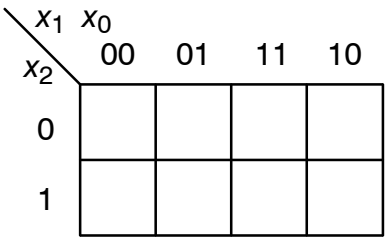
$y_2 =$

Bild 2.14: Karnaugh-Diagramm für y_2



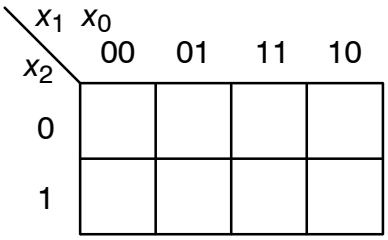
$y_3 =$

Bild 2.15: Karnaugh-Diagramm für y_3



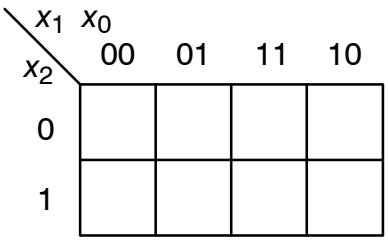
$y_4 =$

Bild 2.16: Karnaugh-Diagramm für y_4



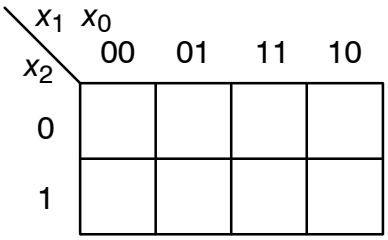
$y_5 =$

Bild 2.17: Karnaugh-Diagramm für y_5



$y_6 =$

Bild 2.18: Karnaugh-Diagramm für y_6



$y_7 =$

Bild 2.19: Karnaugh-Diagramm für y_7

▶ Bauen Sie die Schaltung auf und überprüfen Sie die Funktion.
Die Eingangsgrößen liefern Schalter. Es stehen jeweils die Signale x_k sowie die inversen Signale $\bar{x}_k$ zur Verfügung.
Die Ausgänge sind mit LEDs zu verbinden.

***

15.3 Lösung Labor #01 mit programmierbarer Logik

Die Lösung mit programmierbarer Logik (VHDL) zeigt das folgende (volständige) Programm. Der Wert '–' bedeutet "don't care" und ist notwendig, um nicht erforderlich Logik (latches) auszuschließen.

```
----------------------------------------------------------------
-- Company:          Univ. Bremerhaven
-- Engineer:         Kai Mueller
-- Create Date:      14:58:51 04/09/2011
-- Design Name:
-- Module Name:      bgtop - Behavioral
-- Revision 0.01 - File Created
--
----------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity bgtop is
    Port ( switch3 : in  STD_LOGIC_VECTOR (2 downto 0);
           led : out  STD_LOGIC_VECTOR (7 downto 0));
end bgtop;

architecture Behavioral of bgtop is
begin
    WITH switch3 SELECT
        led <=  "00000001" WHEN "000",
                "00000010" WHEN "001",
                "00000110" WHEN "010",
                "00001110" WHEN "011",
                "00011110" WHEN "100",
                "00111110" WHEN "101",
                "01111110" WHEN "110",
                "11111110" WHEN "111",
                "--------" WHEN OTHERS;
end Behavioral;
```

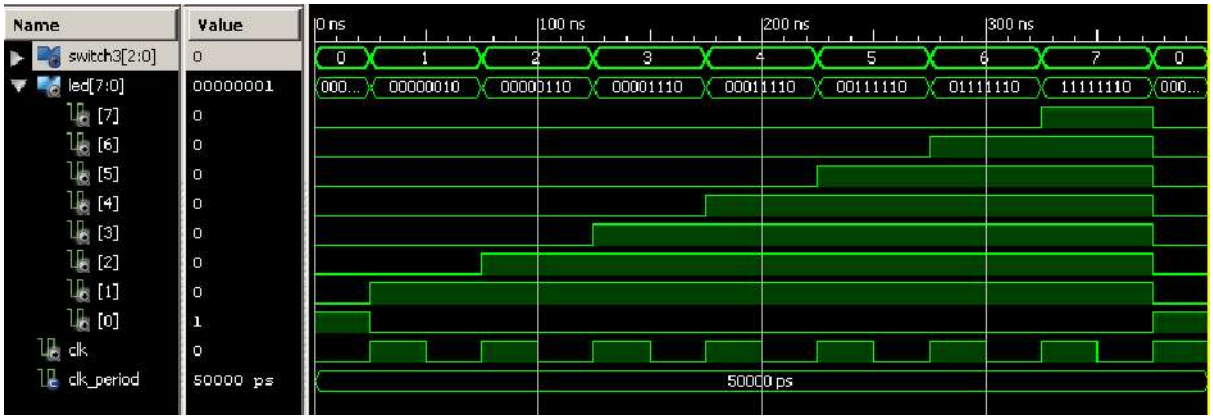


Bild 3.16: Simulation der Bargraph-Schaltung

***

Teil 2

16 Sequenzielle Schaltungen und getaktete Systeme

Bisher wurden digitale Schaltungen zur Verwirklichung logischer Funktionen behandelt. Viele Digitalsysteme erfordern jedoch eine "Ablaufsteuerung", d.h. eine Abfolge verschiedener *Zustände*. Hierzu ist ein "Gedächtnis" erforderlich, das den jeweiligen Zustand speichert.

16.1 Bistabile Elemente

Eine bistabile Schaltung behält eine gespeicherte Information beliebig lange. Die Stabilität beruht auf dem Prinzip der Mitkopplung, indem die Ausgänge wieder (positiv) auf die Eingänge wirken. Bild 8.1 zeigt das Prinzip, dass auch bei statischen Speichern angewandt wird.

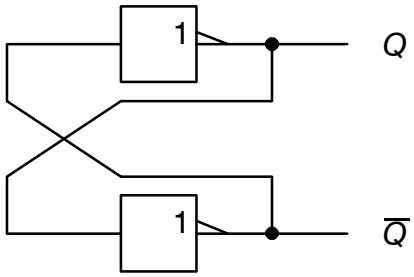


Bild 8.1: Bistabiler Speicher

Schaltet man die Spannung ein, so bleibt der einmal eingenommene Zustand gespeichert. Dies liegt an der positiven Rückkopplung der Schaltung (zwei Inverter in Reihe ergeben wieder das ursprüngliche Signal). Der Zustand nach dem Einschalten kann allerdings nicht vorhergesagt werden. Die Schaltung in Bild 8.1 verdeutlicht die Funktion eines Speichers, jedoch ist sie ziemlich sinnlos, da der einmal gespeicherte Zustand nicht mehr veränderbar ist.

> Eine Schaltung mit einem stabilen Zustand, aber undefiniertem Verhalten (wie in Bild 8.1) nennt man *metastabil*.

16.1.1 RS-Latch

In vielen Lehrbüchern werden die Begriffe “Latch” und “Flip-Flop” als Synonyme gebraucht. Wir wollen hier die Bezeichnung Latch für ein Speicherelement benutzen, bei dem seine Ausgänge Änderungen der Eingangssignale folgen *können*.

> Latch = Speicherelement, das (asynchron) gesetzt oder gelöscht werden kann (Ausgänge folgen Eingangsignalen).

Im Gegensatz dazu sind Flip-Flops Speicherelemente, die nur *synchron* mit Hilfe eine Takt-Signals (Clock) ihren Zustand abhängig von Eingangsgrößen ändern (s. Abschnitt 16.2).

Um also das Latch in Bild 8.1 steuerbar zu machen, werden zusätzliche Eingänge benötigt (Bild 8.2).

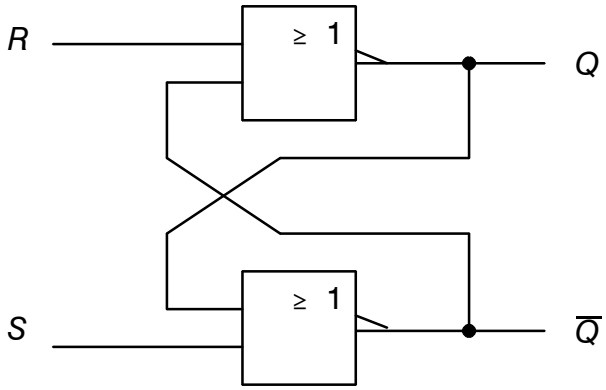


Bild 8.2: RS-Latch mit NOR-Gattern

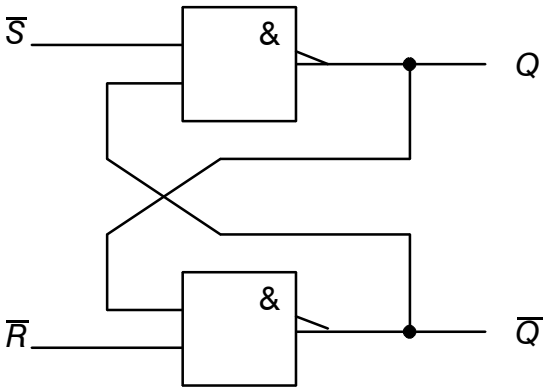
Die Rückkopplung – wie bei den Invertern – bleibt erhalten, wenn sowohl R als auch S den Wert ’0’ annehmen. Die vollständige Funktion wird durch die folgende Wahrheitstabelle beschrieben.

S	R	Q	$\overline{Q}$	
0	0	speichern		
0	1	0	1	(Löschen)
1	0	1	0	(Setzen)
1	1	0	0	(metastabil)

Man erkennt, dass mit dem S-Eingang das Latch gesetzt werden kann (Set) und mit dem R-Eingang das Latch gelöscht wird (Reset). Der metastabile Zustand wird verständlicherweise vermieden. Es spricht jedoch nichts dagegen, auch diesen Zustand für bestimmte Zwecke zu nutzen, da $R = 1$ und $S = 1$ natürlich nicht grundsätzlich unzulässig sind.

> RS-Latches können mit NOR- oder NAND-Gattern verwirklicht werden.

Eine Realisierung mit NAND-Gattern zeigt Bild 8.3. Dabei ist die Polarität der Ansteuersignale gegenüber der NOR-Schaltung vertauscht (folgt aus dem DeMorganschen Theorem).



$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$	
0	0	1	1	(metastabil)
0	1	1	0	(Setzen)
1	0	0	1	(Löschen)
1	1	speichern		

Bild 8.3: RS-Latch mit NAND-Gattern (auch $\overline{RS}$-Latch genannt)

Zu beachten ist, dass hier die Eingangswerte $S = 1$, $R = 1$ zum Speichern der Ausgangswerte verwendet werden.

Da Latches häufig in Schaltungen Verwendung finden, besitzen sie ein eigenes Symbol.

Bild 8.4: Schaltsymbole für RS-Latch und $\overline{RS}$-Latch

16.1.2 RS-Latch mit Enable-Eingang

Eine Erweiterung des RS-Latches ist ein Enable-Eingang, der die Änderung des Speicherinhalts nur zulässt, wenn der Enable-Eingang aktiv ist (abhängig von der Polarität Active-High oder Active-Low). Eine Verwirklichung mit NAND-Gattern zeigt Bild 8.5.

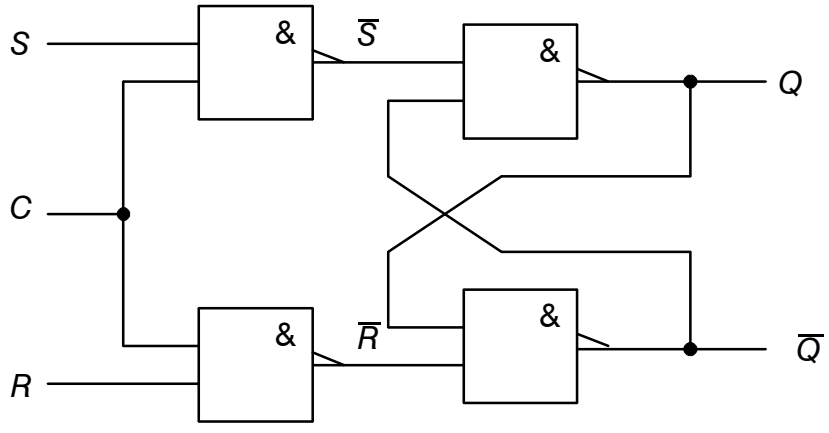


Bild 8.5: RS-Latch mit Enable-Eingang ($C =$ Clock)

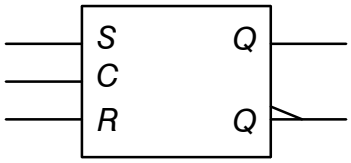


Bild 8.6: Schaltsymbol für RS-Latch mit Enable-Eingang

Das Latch verhält sich wie ein gewöhnliches RS-Latch, wenn der Eingang C = '1' ist. Nimmt C den '0'-Pegel an, so wird der gegenwärtige Zustand gespeichert; die Werte der R- und S-Eingänge sind dann irrelevant.

S	R	C	Q	$\overline{Q}$	
0	0	1	speichern		
0	1	1	0	1	(Löschen)
1	0	1	1	0	(Setzen)
1	1	1	1	1	(metastabil)
x	x	0	speichern		

16.1.3 D-Latch

Häufig möchte man nur Daten speichern und nicht Set- und Reset-Eingänge verwenden. Für diese Aufgabe ist ein D-Latch geeignet. Da Set- und Reset-Eingänge immer unterschiedliche Werte für setzen und Rücksetzen benötigen, kann man ein D-Latch mit einem Inverter aufbauen.

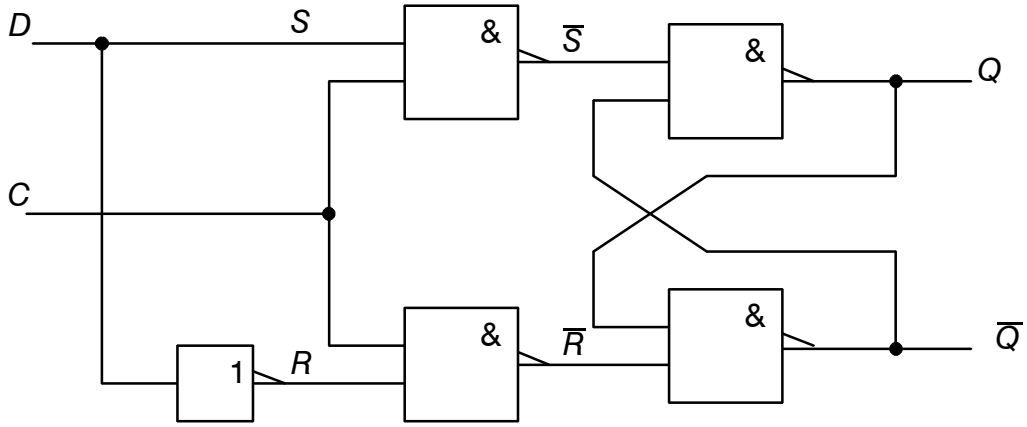


Bild 8.7: D-Latch mit Enable-Eingang (C = Clock)

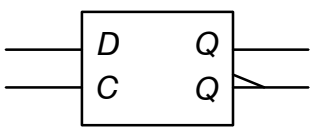


Bild 8.8: Schaltsymbol für D-Latch

D	C	Q	$\overline{Q}$	
0	1	0	1	(Löschen)
1	1	1	0	(Setzen)
x	0	speichern		

Wenn also $C = 1$ ist, folgt der Ausgang Q dem Eingang D. Man bezeichnet diesen Zustand als *transparent* oder *open*. Wird $C = 0$ gesetzt, so ist das Latch im Zustand *close*, d.h. der Ausgang hängt nicht mehr von gegenwärtigen Wert von D ab.

16.2 Flip-Flop

Die Übernahme von Daten des Eingangs/der Eingänge erfolgt beim Flip-Flop an der positiven oder der negativen Flanke eines Signals, dass mit *Clock* bezeichnet wird.

Flip-Flop = *flankengetriggertes* "Latch"

Die Übernahme von Daten mit einer Flanke wird auch *Flankentriggerung* genannt. Die Funktion lässt sich mit zwei D-Latches verwirklichen.

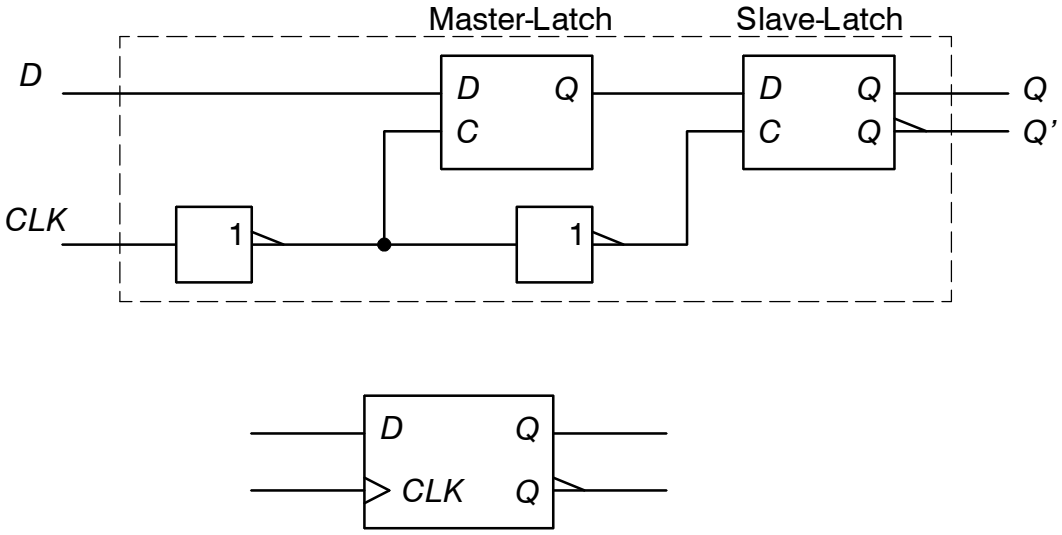


Bild 8.9: Flip-Flop mit zwei D-Latches (positiv flankengetriggert) und Schaltsymbol

Das Aufstellen der Wahrheitstabelle erfordert nun ein neues Symbol, das eine positive Flanke darstellt.

D	C	Q	$\overline{Q}$	
0	↓	0	1	(Löschen)
1	↓	1	0	(Setzen)
x	0	speichern		
x	1	speichern		

Für *CLK* = 0 übernimmt das Master-Latch die Daten vom *D*-Eingang. Mit der positiven Flanke des *CLK*-Signals "schließt" das Master-Latch und das Slave-Latch "öffnet". Damit wird nur für den Übergang von '0' auf '1' (positive Flanke) des *CLK*-Signals der Wert auf dem *D*-Eingang gespeichert. Geht das *CLK*-Signal wieder auf '0', so behält das Flip-Flop seinen Wert bei.

Negativ flankengetriggerte Flip-Flops arbeiten einfach mit einem invertierten *CLK*-Signal.

Häufig weisen D-Flip-Flops auf Eingänge für ein *asynchrones* (= unabhängig vom *CLK*-Signal) Setzen oder Löschen auf. Die Eingänge nennt man *PRESET* und *CLEAR*.

PRESET = asynchrones Setzen eines Flip-Flops.

CLEAR = asynchrones Löschen eines Flip-Flops.

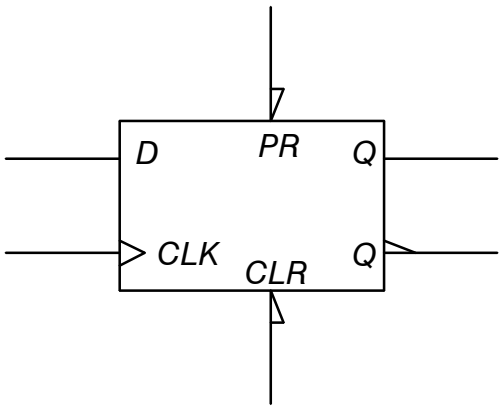


Bild 8.10: D-Flip-Flop mit *PRESET* und *CLEAR*

Alle weiteren Flip-Flop-Typen lassen sich leicht aus einem D-Flip-Flop herleiten.

16.3 Getaktete Systeme (Zustandsautomaten, State-Machines)

Es bestehen grundsätzlich zwei verschiedene Architekturen für Zustandsautomaten, die sich in der Verwendung der Eingangssignale unterscheiden.

Zustandsautomat = Erzeugung von Ausgangssignalen als Folge von Eingangssignalen und Zustandsgrößen. Zustandsgrößen sind die Inhalte von Speichern (Flip-Flops).

Die Zustandsgrößen werden jeweils synchron durch ein *Clock*-Signal neu gesetzt. Dieses *Clock*-Signal nennt man auch Taktsignal. Man nennt diese Systeme deshalb auch getaktete Systeme.

Komplexe digitale Systeme wie Mikroprozessoren oder Anlagensteuerungen sind immer getaktete Systeme

Man unterscheidet sogenannte *Moore*- und *Mealy*-Maschinen.

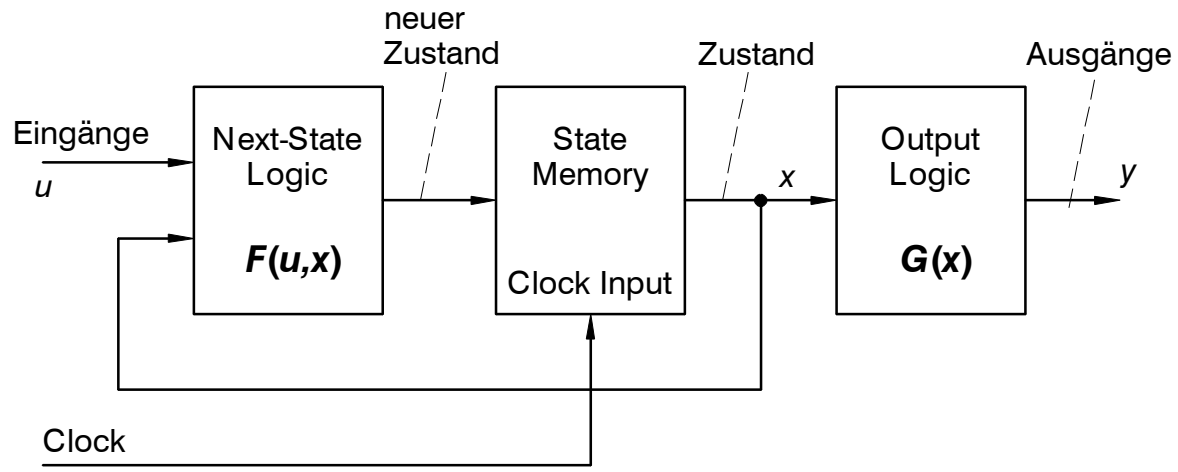


Bild 8.11: Moore-Maschine

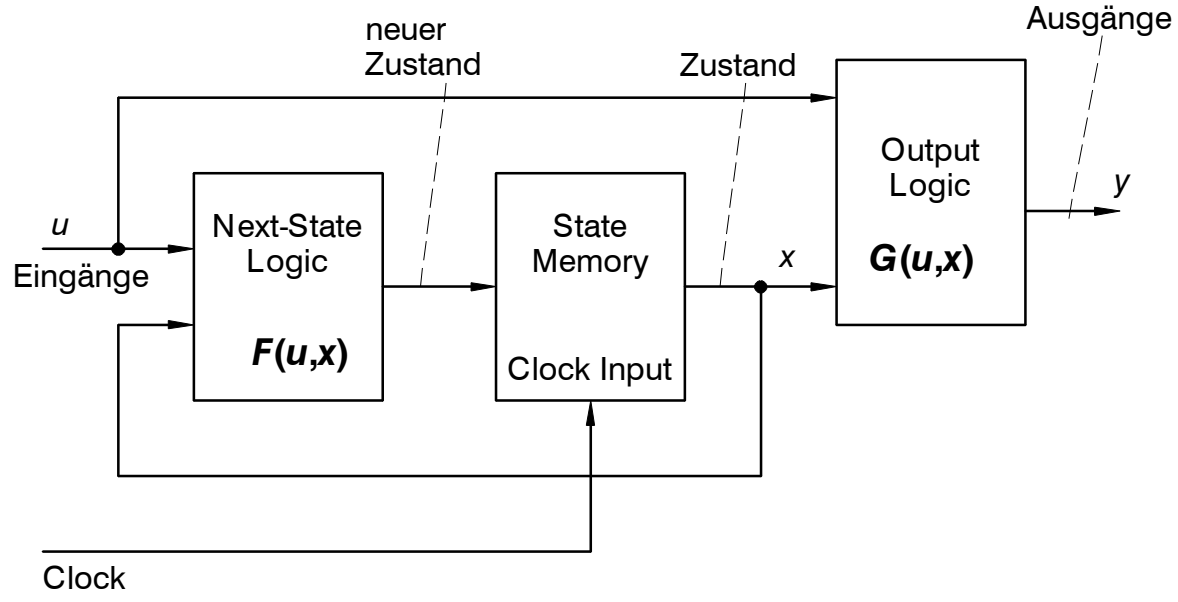


Bild 8.12: Mealy-Maschine

Der offensichtliche Unterschied zwischen beiden Maschinen besteht darin, dass bei der Mealy-Maschine die Eingangssignalen sich unmittelbar auf die Ausgänge auswirken. Bei der Moore-Maschine hängen die Ausgänge ausschließlich von den Zustandsgrößen ab.

Die logischen Funktionen

$$F(u,x)\,, \quad G(x) \tag{8.1}$$

der Moore-Maschine bzw.

$$F(u,x)\,, \quad G(u,x) \tag{8.2}$$

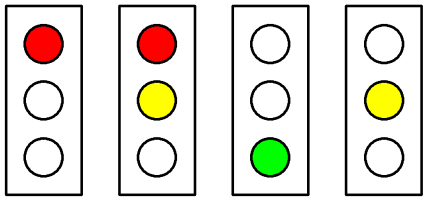
der Mealy-Maschine sind dabei die bekannten logischen Funktionen der vorangegangenen Kapitel. Das State-Memory besteht aus einer Reihe von Flip-Flops ohne jegliche weitere Logik.

Diese formale Darstellung eignet sich gut zum Entwurf von komplexen Schaltungen. Einfache Automaten kann man natürlich auch intuitiv entwerfen.

Bei praktischen Anwendungen können einzelnen Teile der Zustandsautomaten auch entfallen (bis auf die Speicher und die Clock-Leitung natürlich). Beispielsweise können die Ausgangsgrößen y identisch mit den Zustandsgrößen x sein. Dann entfällt die Funktion $F(x)$.

16.4 Beispiel: Ampelsteuerung

Eine einfache Ampelsteuerung für eine Fahrtrichtung muss mindestens die Zustände in Bild 8.13 beherrschen.



Zustand 0 1 2 3

Bild 8.13: Minimaler Satz von Zuständen für eine Ampel

Natürlich existieren bei einer realen Ampelanlage wesentlich mehr Zustände (Ampel der Kreuzungsfahrbahn, Fußgängerampel, Blinken des gelben Lichtes bei Störungen oder im Nachtbetrieb usw.). Wir wollen aber zunächst nur die Ampelphase gemäß Bild 8.13 verwirklichen, die sich zyklisch wiederholen soll.

Zu Beginn müssen wir die Eingangs-, Ausgangs- und Zustandsgrößen definieren, mit denen wir die Aufgabe lösen können.

EINGANGSGRÖSSE *u*: Reset-Signal, mit dem die Ampel auf “rot” geschaltet wird (soll Active-Low sein, d.h. ’0’ bedeutet “Reset”).

ZUSTANDSGRÖSSEN *x*: Es existieren 4 Zustände. Dafür benötige wir 2 Bits, d.h. wir benötigen 2 Flip-Flops (z.B. D-Flip-Flops).

AUSGANGSGRÖSSEN *y*: rot, gelb, grün (Ansteuersignale für die einzelnen Lampen).

Für dies Aufgabe ist eine Moore-Maschine geeignet, da sich die Ausgangsgrößen unmittelbar aus den Zustandsgrößen ableiten lassen.

16.4.1 Next-State-Logic (Eingangslogik) $F(u, x)$

Wenn u = '0', dann soll immer der Zustand 0 generiert werden, anderenfalls der jeweilige Folgezustand. Die Wahrheitstabelle sieht dann wie folgt aus.

			Folgezustände	
u	x1	x0	x1n	x0n
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	1
1	1	1	0	0

$F(u, x)$

Eine Minimale Realisierung (mit Karnaugh-Diagramm) ist in Bild 8.14 dargestellt.

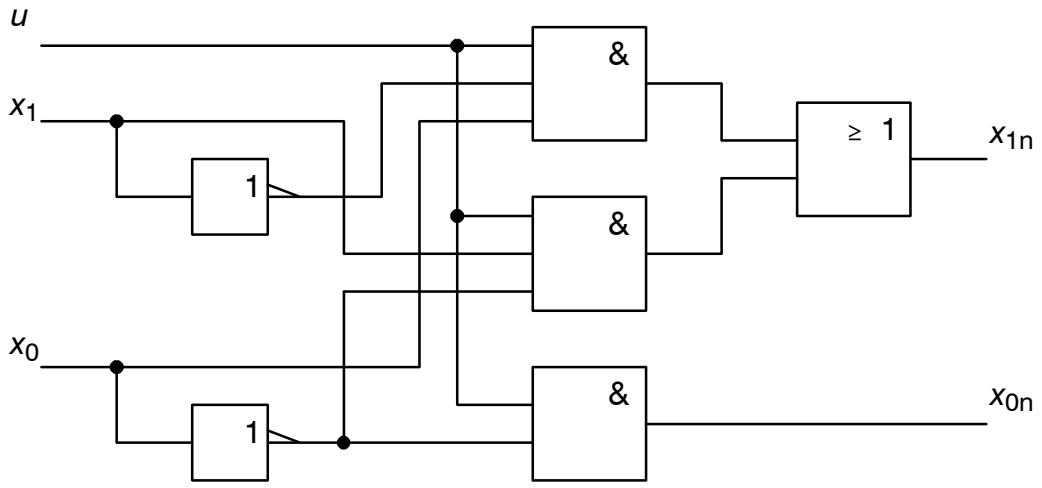


Bild 8.14: Minimale Realisierung der Funktion $F(u, x)$

16.4.2 Output-Logic (Ausgangslogik) $G(x)$ (Moore-Maschine)

Hier muss die Zuordnung der Zustände zu den Ansteuersignalen für die einzelnen Lampen generiert werden.

		Lampen		
x1	x0	rot	gelb	grün
0	0	1	0	0
0	1	1	1	0
1	0	0	0	1
1	1	0	1	0

G(x)

Die Realisierung zeigt Bild 8.15.

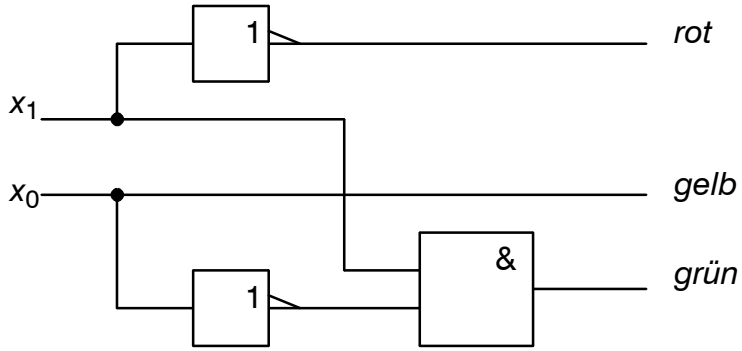


Bild 8.15: Minimale Realisierung der Funktion $G(x)$

16.4.3 State-Memory (Zustandsspeicher)

Das State-Memory ist der einfachste Teil der Gesamtschaltung. Er besteht aus den 2 Flip-Flops zur Speicherung der Zustandsgrößen x_0 sowie x_1.

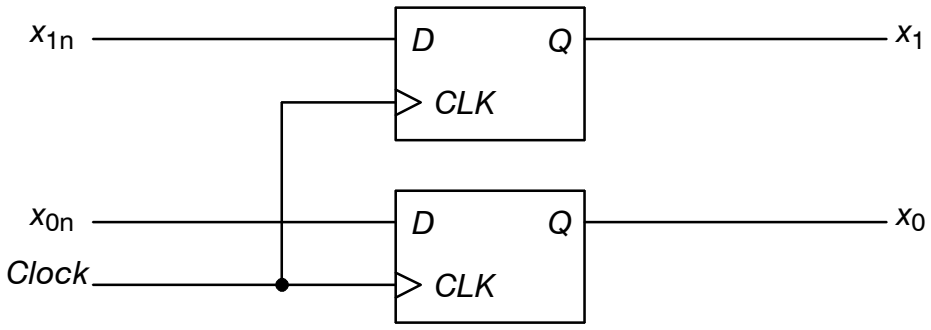


Bild 8.16: State-Memory

16.4.4 Vollständige Ampelsteuerung

Die Gesamtschaltung ist die Zusammenfassung der Bildern 8.14–8.16.

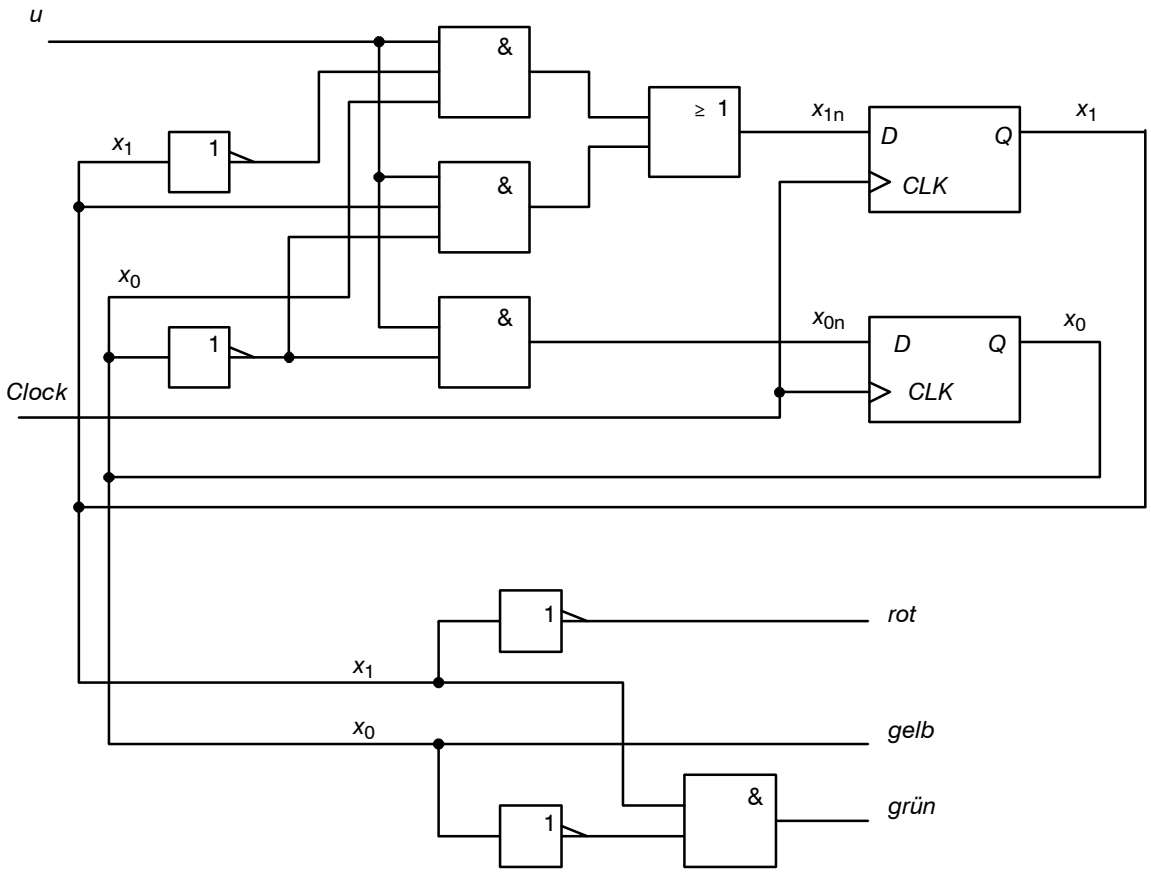


Bild 8.17: Vollständige Schaltung

16.5 Zustandsdiagramm

Das Verhalten eines Zustandsautomaten lässt sich im Zustandsdiagramm verdeutlichen.

Jeder Zustand wird durch einen Kreis symbolisiert; Pfeile kennzeichnen die möglichen Übergänge. An den Pfeilen sind wird die Bedingung notiert, unter der der Übergang erfolgt. Steht an einem Pfeil kein Bezeichnung, so erfolgt der übergang im nächsten Takt.

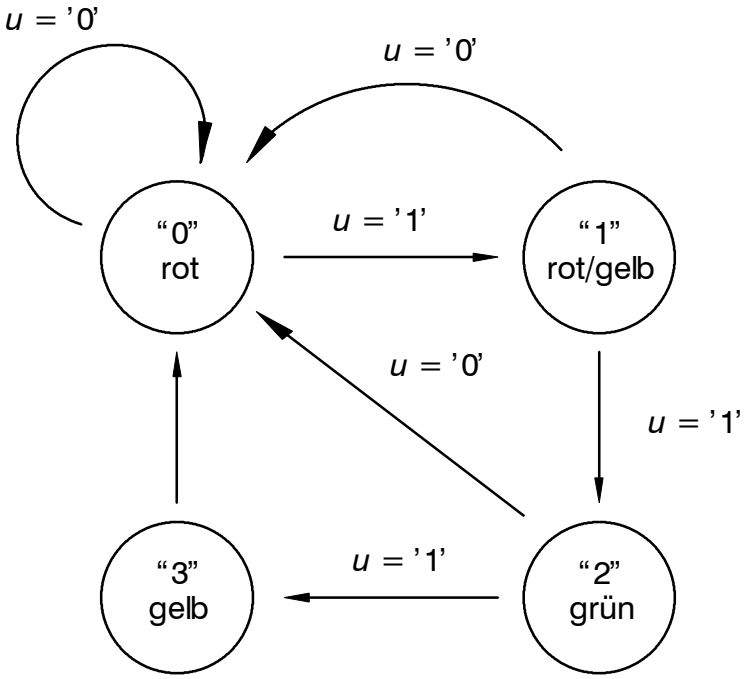


Bild 8.18: Zustandsdiagramm für die Ampelsteuerung

16.6 Übung: Erweiterung der Ampelsteuerung um die Funktion "Blinken des gelben Lichts"

Ist eine Ampelanlage außer Betrieb, so muss dies durch Blinken des gelben Lichts angezeigt werden. Verwenden Sie dazu den Eingang u. Ist $u = '0'$, so soll das gelbe Licht blinken. Für $u = '1'$ soll der normale Ampelzyklus ablaufen.

- a) Definieren Sie alle benötigten Zustände. Wieviele Flip-Flops werden benötigt?
- b) Zeichnen Sie ein Zustandsdiagramm für diese Zustände mit den Übergangsbedingungen.
- c) Entwerfen Sie die Eingangslogik $F(u, x)$.
- d) Entwerfen Sie die Ausgangslogik $G(x)$.
- e) Zeichnen Sie die vollständige Ampelsteuerung.

17 Labore Sequentielle Schaltungen

Sequentielle Schaltungen enthalten Speicher. In digitalen Schaltungen werden Informationen in einem Latch oder einem Flip-Flop gespeichert. Ein Latch steuert den

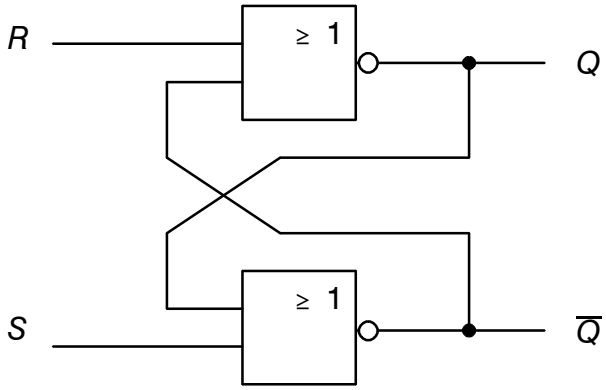
Speichervorgang durch den Pegel eines Signals; ein Flip-Flop reagiert immer auf eine Flanke (Übergang $0 \rightarrow 1$ oder $1 \rightarrow 0$).

Lab #03:

17.1 Labor #03: Latch und Flip-Flop

Die folgenden Schaltung sind in Hardware aufzubauen und durch ihre Wahrheitstabelle auf Funktion zu überprüfen.

17.1.1 RS-Latch mit NOR-Gattern

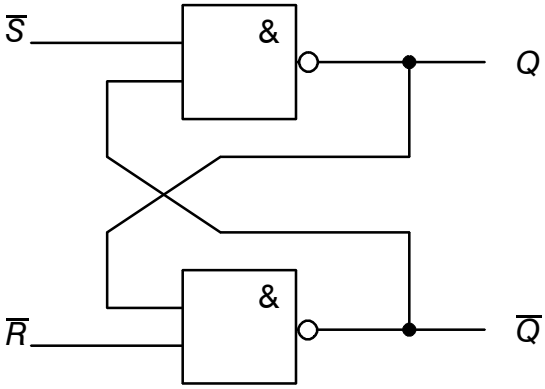


Die Signale R und S werden durch Kippschalter vorgegeben.

S	R	Q	$\overline{Q}$	
0	0	Q	$\overline{Q}$	(Speichern)
0	1	0	1	(Löschen)
1	0	1	0	(Setzen)
1	1	0	0	(metastabil)

- Überprüfen Sie die Gültigkeit der Wahrheitstabelle
- Erläutern Sie den Metastabilität und geben Sie eine Schaltung an, mit der Sie von Metastabilität in den Speicherzustand wechseln können.

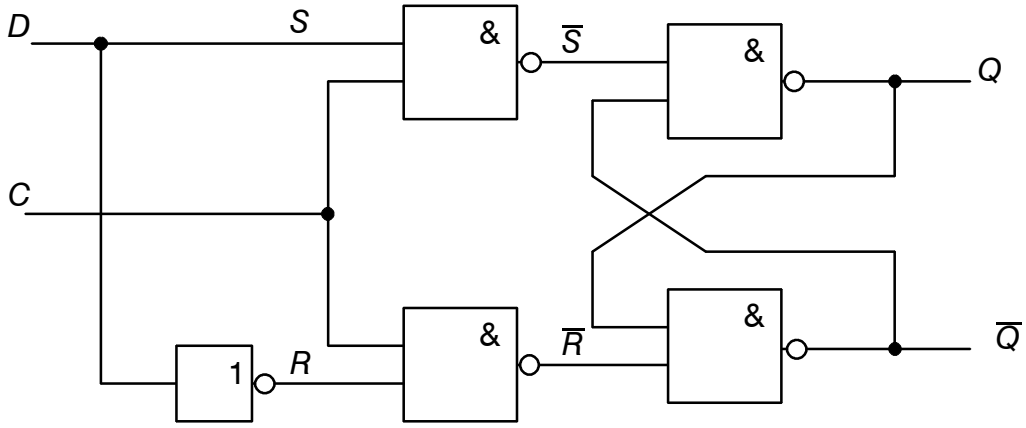
17.1.2 RS-Latch mit NAND-Gattern



$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$	
0	0	1	1	(metastabil)
0	1	1	0	(Setzen)
1	0	0	1	(Löschen)
1	1	Q	$\overline{Q}$	(speichern)

▶ Überprüfen Sie die Gültigkeit der Wahrheitstabelle durch unterschiedliche Wahl der Eingangsgrößen.

17.1.3 D-Latch (NAND)



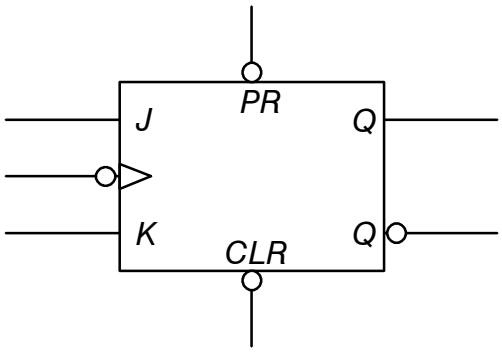
C	D	Q	$\overline{Q}$	
0	0	Q	$\overline{Q}$	(speichern)
0	1	Q	$\overline{Q}$	(speichern)
1	0	0	1	(löschen)
1	1	1	0	(setzen)

▶ Überprüfen Sie die Gültigkeit der Wahrheitstabelle durch unterschiedliche Wahl der Eingangsgrößen.

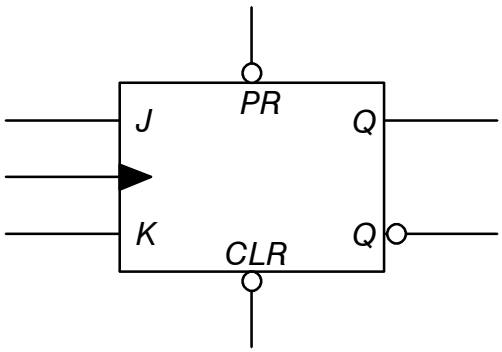
17.1.4 JK-Flip-Flops

Obwohl Flip-Flops aus (rückgekoppelten) Gattern erzeugt werden können, ist der Schaltungsaufwand mit den zur Verfügung stehenden Gattern recht hoch. Es sollen deshalb die vorhandenen JK-Flip-Flops verwendet werden.

Die zur Verfügung stehenden JK-Flip-Flops haben folgendes Symbol:



Das Taktsignal bei diesem Flip-Flop wirkt sich immer auf der negativen Flanke aus. Das auf der Schaltung verwendete Schaltbild symbolisiert dies durch ein ausgefüllte Dreieck (veraltet).



Das Flip-Flop besteht aus einem asynchronen Teil (taktunabhängig), die durch PR (Preset) und CLR (Clear) angesteuert wird und einem synchronen Teil. Die Wahrheitstabelle für den asynchronen Teil lautet:

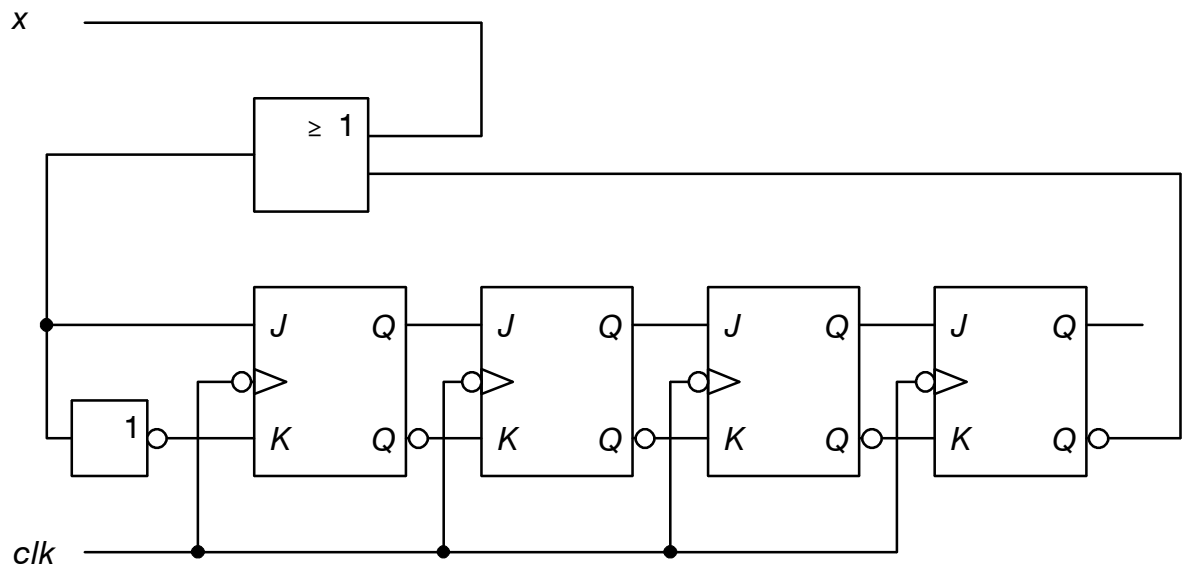
PR	$\overline{CLR}$	Q	$\overline{Q}$	
0	0	unzulässig/undefiniert		
0	1	0	1	(Löschen asynchron)
1	0	1	0	(Setzen asynchron)
1	1	Q	$\overline{Q}$	(speichern)

Die Eingänge PR und CLR werden häufig nur zur Initialisierung benutzt. Im normalen Betrieb sind diese Signale stets '1' (speichern).

Der synchrone Teil wird durch die folgende "Wahrheitstabelle" beschrieben (Wahrheitstabellen bestehen streng genommen nur für kombinatorische Logik, d.h. Gatterschaltungen):

CLK	J	K	Q	$\overline{Q}$	
0	X	X	Q	$\overline{Q}$	(speichern)
1	X	X	Q	$\overline{Q}$	(speichern)
↑	X	X	Q	$\overline{Q}$	(speichern)
↓	0	0	Q	$\overline{Q}$	(speichern)
↓	0	1	0	1	(löschen)
↓	1	0	1	0	(setzen)
↓	1	1	$\overline{Q}$	Q	(wechseln/toggeln)

▶ Bauen Sie die folgende Schaltung auf und erläutern Sie deren Funktion anhand der durchgeführten Experimente. Verwenden Sie als Taktsignal den Ausgang des Tasters $\overline{Q}$. Das Eingangssignal *x* soll durch einen Schalter bereit gestellt werden. PR (Preset) und CLR (Clear) werden nicht verwendet.



***

Lab #04:

17.2 Labor #04: Rückwärtszähler (3 Bit)

Sequentielle Schaltungen ermöglichen den Aufbau digitaler Steuerungen. Sie erfordern einen Takt, der die Weiterschaltung der Zustände bewirkt.

Im Rahmen dieses Labors soll eine Aufgabe mit Gatterschaltungen in minimaler Form (DNF und KNF) gelöst werden. Nach der Minimierung der logischen Funktionen kann die Schaltung durch Verbinden von Gattern verwirklicht werden. Hierzu stehen genug Gatterfunktionen (AND, OR, NAND, NOR) und Verbindungsleitungen zur Verfügung.

Ein Rückwärtszähler soll folgende Werte jeweils bei Anliegen des Taktsignals erzeugen:

$$000 - 111 - 110 - 101 - 100 - 011 - 010 - 001 - 000 - 111 - ...$$

Es wird also zyklisch von 7_{10} bis 0_{10} gezählt.

Der Zustandsgraph hat damit die in Bild 2.11 gezeigte Form.

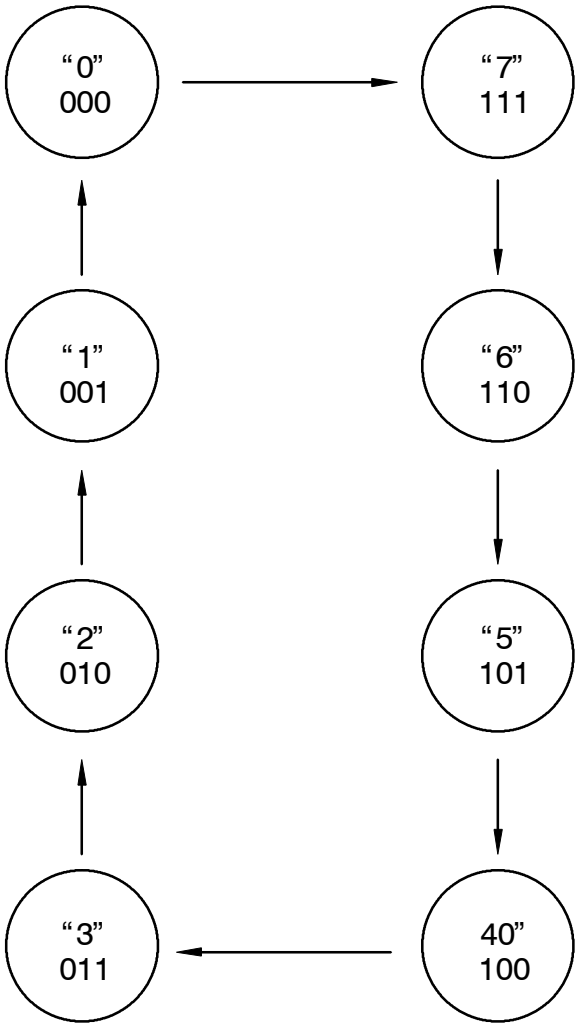
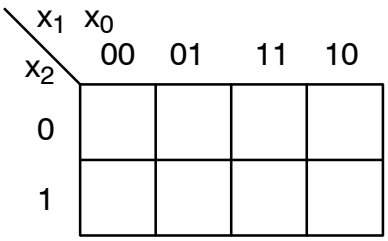


Bild 2.20: Zustandsdiagramm für den 3 Bit-Rückwärtszähler

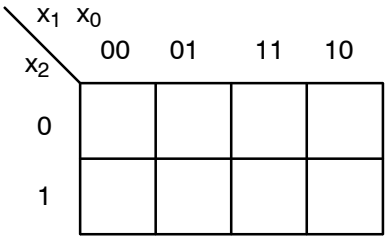
Es ist die Funktion $F(x)$ (Next-State-Logic) zu entwerfen und aufzubauen.

- Geben Sie die Wahrheitstabelle für x2n, x1n und x0n an.
- Minimieren Sie die Funktionen im Karnaugh-Diagramm (s. folgende Vordrucke). Geben Sie die minimale boolsche Gleichung für den jeweiligen Ausgang an.



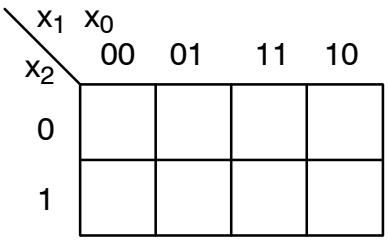
$x_{2n} =$

Bild 2.21: Karnaugh-Diagramm für x_{2n}



$x_{1n} =$

Bild 2.22: Karnaugh-Diagramm für x_{1n}

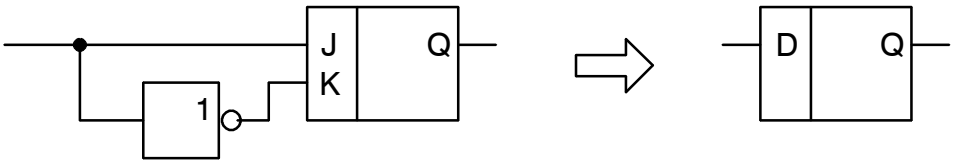


$x_{0n} =$

Bild 2.23: Karnaugh-Diagramm für x_{0n}

- Zeichnen Sie die gesamte Schaltung.
- Bauen Sie die Schaltung auf und überprüfen Sie deren Funktion.
  Der Takt ist mit dem Taster zu verbinden. Der Zählerstand (Ausgänge der Flip-Flops) kann mit der digitalen Anzeige verbunden werden.

Hinweis 1: Im Versuchsaufbau stehen keine D-Flip-Flops zur Verfügung. Sie können die JK-Flip-Flops aber durch folgende Beschaltung in D-Flip-Flops wandeln:



Hinweis: Das JK-Flip-Flop wechselt seinen Ausgang auch ohne Beschaltung der Eingänge (Toggle-Modus).

***

Lab #05:

17.3 Labor #05: Ampelschaltung (One-Hot-Encoding)

Kleine Automaten mit einer geringen Anzahl an Zustandsgrößen werden oft anstelle einer binären Kodierung als “One-Hot-Encoded”-Schaltung verwirklicht. Das bedeutet, dass jeder Zustand sin eigenes FF erhält.

Man benötigt mehr FFs, aber die Aufwand für kombinatorische Logik (Gatterschaltungen) verringert sich häufig. Eine Minimierung ist in der Regel überflüssig.

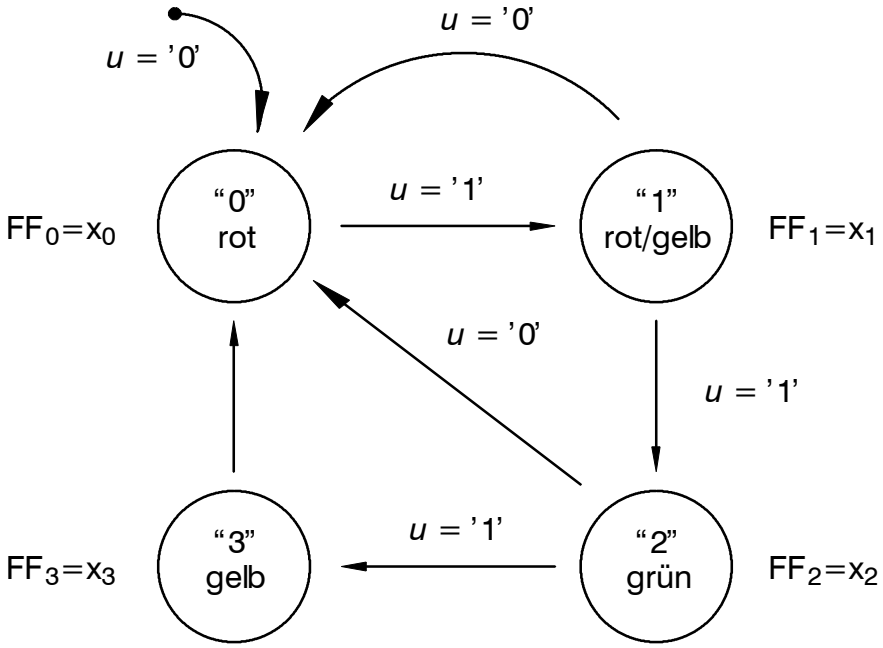


Bild 3.17: Zustandsdiagramm für eine Ampelschaltung mit One-Hot-Encoding

Es ergibt sich folgender Ablauf für die Zustands-Flip-Flops x_3 bis x_0:

u	x3	x2	x1	x0	x3n	x2n	x1n	x0n
0	X	X	X	X	0	0	0	1
1	0	0	0	1	0	0	1	0
1	0	0	1	0	0	1	0	0
1	0	1	0	0	1	0	0	0
1	1	0	0	0	0	0	0	1

Gegenüber einer binären Kodierung ist die Zahl der möglichen Zustände stark reduziert. Aus der obigen Tabelle ergeben sich die folgenden Grundregeln für One-Hot-Encoding:

- Wenn u='0', dann wird x_1 gesetzt und alle anderen Zustände gelöscht.

- Wenn u='1', dann wird der Folgezustand gesetzt und der aktuelle Zustand setzt sich selbst zurück..

Für x_0n erhält man somit

x_{0n} wird immer gesetzt, wenn u='0'

x_0 setzt sich selbst wieder zurück

$$x_{0n} = u' + (x_3 \cdot x_0')$$

Folgezustand Vorgängerzustand

▶ Geben Sie die boolsche Gleichung für x_{1n} an:

$x_{1n} =$

▶ Geben Sie die boolsche Gleichung für x_{2n} an:

$x_{2n} =$

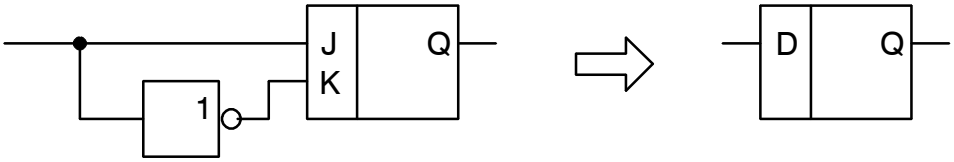
▶ Geben Sie die boolsche Gleichung für x_3' an:

$x_3' =$

▶ Geben Sie die Logik für die Kodierung der Ampelfarben rot, gelb und grün aus den Zustandsgrößen an:

- Bauen Sie die Schaltung auf und kontrollieren Sie deren Funktion. Der Takt wird durch den Taster vorgegeben. Für die Ampelleuchten verwenden Sie LEDs.

Hinweis: Im Versuchsaufbau stehen keine D-Flip-Flops zur Verfügung. Sie können die JK-Flip-Flops aber durch folgende Beschaltung in D-Flip-Flops wandeln:



***

Teil 3

18 Mikroprozessoren

Mikroprozessoren sind sequenzielle (getaktete) Systeme, deren Funktion durch ein Programm (Software) bestimmt wird. Der große Vorteil besteht darin, dass der Mikroprozessor als Standardbauelement günstig in großen Stückzahlen herstellbar ist. Der Entwickler kann durch Programmierung dann nahezu beliebige Funktionen realisieren. Dies ist oft günstiger, als für eine bestimme Anwendung spezielle ICs zu entwickeln.

Die Komplexitiät der logischen Funktion eines Mikroprozessorsystems ist nur noch durch den verfügbaren Speicher bzw. den *Adressraum* begrenzt. Die Programmierung von Mikroprozessoren ist durch die Verwendung von Hochsprachen (C, C++, Java, Ada) sehr komfortabel und effizient geworden.

Eine schnellere und teilweise kostengünstigere Alternative zu Mikroprozessoren sind programmierbare Digitalbausteine (CPLDs, FPGAs), die keine Programmsequenz abarbeiten, sondern deren Funktion fest programmierbar ist. Auch hierfür existieren inzwischen Hochsprachen (VHDL, Verilog), jedoch ist die Programmierung deutlich aufwendiger als bei Mikroprozessoren.

Durch die Möglichkeiten der heutigen Digitaltechnik kann zwischen beiden Ansätzen jedoch nicht mehr einfach getrennt werden, da moderne FPGAs eine Größe besitzen (ca. 8 Millionen Gatterfunktionen) die eine Integration eines kompletten Mikroprozessors in ein FPGA gestatten.

18.1 Historische Entwicklung

Die grundlegenden Ideen zum Bau eines universellen Computers stammen von Charles Babbage, dessen Konzept aus dem Jahre 1833 auch heute noch Gültigkeit hat. man kann zeigen, mit sich mit einem sogenannten Universalrechner beliebige Probleme lösen lassen.

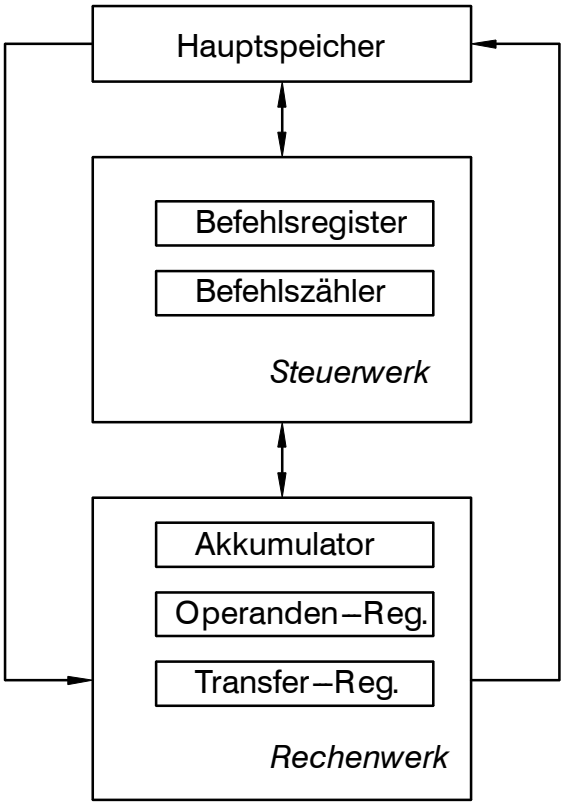


Bild 3.18: Universalrechner (ohne Ein-/Ausgabeeinheiten)

Der Hauptspeicher enthält das Programm und die Operanden. In dem Steuerwerk wird ein Befehl aus dem Hauptspeicher in das Befehlsregister geladen und dort interpertiert. Der Befehlszähler wird inkrementiert, so dass der folgende Befehl aus dem Hauptspeicher geladen werden kann. Bei Sprüngen im Programm wird der Befehlszeiger auf den nächsten zu bearbeitenden Befehl gesetzt.

Das Rechenwerk führt die arithmetischen und logischen Funktionen mit Daten aus, die ebenfalls aus dem Hauptspeicher geladen und aich dort wieder abgelegt werden. Gewöhnlich enthält das Rechenwerk mehrere Operanden-Register, die einen schnellen Zugriff auf die zu verarbeitenden Daten gestatten. Das Ergebnis einer Verknüpfung steht geöhnlich im sogenannten Akkummulator.

Der erste elektronische Rechner nach diesem Prinzip, Univac1 (1950), wurde vollständig in Röhrentechnik aufgebaut. Moderne Prozessoren bestehen aus Silizium und Metall (Aluminium oder Kupfer) und sind in der sogenannten CMOS-Technologie aufgebaut (Complementary Metal Oxyde Semiconductor). Oft weicht die Sruktur auch von der klasssischen Form in Bild 3.18 ab.

19 CISC und RISC

Die Leistungsfähigkeit eines Prozessors lässt sich einfach über den Kehrwert der Laufzeit für ein Programm definieren

$$\text{Laufzeit} = \frac{\text{Zeit}}{\text{Takt}} \times \frac{\text{Takte}}{\text{Instruktion}} \times \frac{\text{Instruktionen}}{\text{Programm}} . \tag{3.37}$$

Die gesamte Laufzeit ist also das Produkt aus der Zeit für einen Takt (die Periodendauer des Taktsignals) der Zahl der Takte pro Instruktion mal der Anzahl der Instruktionen für eine bestimme Aufgabe, d.h. für ein Programm. Um also einen leitungsfähigen Prozessor zu entwerfen müssen im Prinzip alle drei Faktoren minimiert werden. Es hat sich jedoch gezeigt, dass dies nicht unabhängig voneinander erfolgen kann.

In frühen Architekuren wurde die Laufzeit durch eine möglichst geringe Anzahl notwendiger Operationen zu verringern, d.h. man minimiert den letzten Teil der Gleichung

$$\text{Laufzeit} = \frac{\text{Zeit}}{\text{Takt}} \times \frac{\text{Takte}}{\text{Instruktion}} \times \frac{\text{Instruktionen}}{\text{Programm}} . \tag{3.38}$$

Eine auf Minimierung der Zahl der Instruktionen optimierte Architektur bezeichnet man als *CISC* (Complex Instruction Set Computer). Möchte man mit einer geringen Anzahl von Instruktionen auskommen, so müssen diese Befehle zwangslaufig sehr komplex ausfallen und benötigen folglich auch viele Takte pro Instruktion.

$$\text{Laufzeit} = \frac{\text{Zeit}}{\text{Takt}} \times \frac{\text{Takte}}{\text{Instruktion}} \times \frac{\text{Instruktionen}}{\text{Programm}} . \tag{3.39}$$

Bei *RISC*-Architekturen (Reduced Instruction Set Computer) verwendet man ganz einfache Instruktionen, die sich in einem Takt ausführen lassen. Zwangsläufig steigt damit aber auch die Anzahl der für Programm benötigte Anzahl an Instruktionen. Die RISC-Architektur ist CISC-Rechnern deutlich überlegen, da eine genaue Analyse von Programmen zeigt, dass wenige einfache Befehle sehr häufig auftreten. RISC-Rechner setzen sich in den Bereichen durch, in denen hohe Rechenleistung oder geringer Leistungsbedarf (z.B. Batteriebetrieb) maßgeblich sind.

Ein wesentlich Grund in der langen Lebensdauer der veralteten CISC-Rechner ist die starke Marktposition von Intel in Zusammenhang mit dem Betriebssystem Windows, das bis Windows/NT wenig für RISC-Rechner geeignet ist.

Auch im Bereich der Mikrocontroller hat sich die RISC-Architektur inzwischen durchgesetzt. Wir wollen exemplarisch den Mikrocontroller am Beispiel des etwas in die Jahre gekommenen 8051-Controllers behandeln. Dieser Controller ist durch die Intel-typische, leicht verkorkste Struktur gekennzeichnet. Diese CISC-Controller sind in der Industrie weit verbreitet und werden in unzähligen Steuerungen eingesetzt.

Die 8051-Struktur ist etwa 25 Jahre alt und wird immer noch für Entwicklungen genutzt. Für Neuentwicklungen ist dies natürlich wenig empfehlenswert. Wir verwenden immerhin ein modernes Derivat der inzwischen schon antiken 8051-Familie. Der 8051 ist ein typischer

Vertreter der CISC-CPUs. Er benötigt 12−24 Takzyklen / Instruktion (RISC = 1 Takt / Instruktion). Das erklärt, warum CISC-Rechner auch bei sehr hohen Taktfrequenzen relativ langsam sind bzw. warum die Performance von RISC-Rechnern auf bei geringer Taktfrequenz höher ist als die von CISC-CPUs.

20 Mikrocontroller

Es existieren viele Definitionen für Mikrocontroller, die sich alle an der Zahl und Art der auf einen Chip integrierten Funktionen orientieren. Wir wollen uns im Rahmen dieser Veranstaltung auf folgende Definition einigen:

> Ein Mikrocontroller MCU (MicroController Unit) ist ein Computer auf einem Chip, d.h. der Chip enthält neben der CPU auch Programm- (Flash) und Datenspeicher (RAM, EEPROM).

Daneben kann der Mikrocontroller auch weitere Peripherie enthalten wie beispielsweise

- Digitale Ein- und Ausgänge (Ports),
- ADC/DAC (analoge Ein- und Ausgänge),
- Serielle Schnittstellen (z.B. USB, USART, CAN),
- Timer,
- Zähler,
- PWM (Pulsweitenmodulation).

***

21 Architektur des eingesetzten Mikrocontrollers

Die Programmentwicklung soll am Beispiel des modernen RISC-Mikrocontrollers “PicoBlaze™ ” von Ken Chapman (Xilinx Inc.) erfolgen. Dieser Mikrocontroller besitzt einen Registersatz von 16 universellen Registern in Verbindung mit einen leistungsfähigen symmetrischen Befehlssatz. Weiterhin eignet sich die Architektur zur Implementierung in einem FPGA oder einem größeren CPLD.

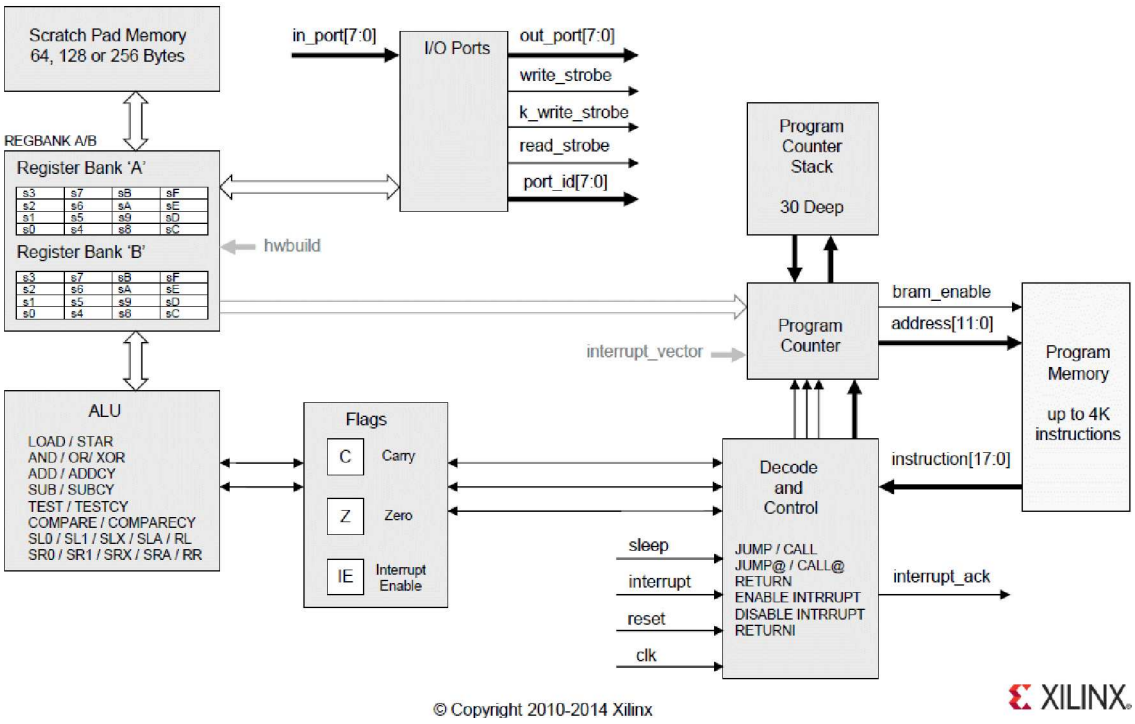


Bild 3.19: Aufbau des PicoBlaze™ Controller-Kerns (KCPSM6)

Das Programm wird in einem 1024 x 18 Bit ROM gespeichert. Die Wortbreite von 18 Bit ermöglicht es, dass jeder Befehl lediglich eine Adresse des Speichers belegt. Andere Architekturen (z.B. die populäre 8051-Derivate) belegen mit einer Anweisung 1, 2 oder 3 Bytes des Programmspeichers.

Für Interrupt und Unterprogramme (CALL xxx) steht ein 31 x 10 Bit Hardware-Stack zur Verfügung. Somit lässt sich das Programm durch ausreichend viele Unterprogramme strukturieren.

Alle Berechnungen sowie alle logischen Operationen können - wie bei RISC-CPUs üblich – nur mit den Registern durchgeführt werden. Hierzu stehen die 16 Register s0–sF in zwei Registerbänken zur Verfügung.

Neben den 16 Registern steht ein RAM (”Scratchpad RAM”) von 64 Byte zur Verfügung (mit den Befehlen FETCH/STORE).

Die PicoBlaze-Architektur beschränkt sich sinnvollerweise auf die Struktur von Bild 3.19, da zusätzliche Funktionen leicht über VHDL oder Verilog im FPGA bzw. CPLD programmiert werden können. Der Mikrocontroller besitzt dann genau die Funktionen, die für eine bestimmte Anwendung benötigt werden.

Der Zugriff auf diese Peripherie (digital I/O, Timer, Zähler, serielle Schnittstellen, externe Busse wie CAN, I^2C usw.) erfolgt durch Transfer zwischen den Registern und dem IN_PORT bzw. dem OUT_PORT (via INPUT/ OUTPUT).

21.1 Zielsystem Zynq-7020

Als Zielsystem steht ein AVNET/Digilent Board mit 53.200 LUTs (Lookup Tables) und 106.400 Flip-Flops zur Verfügung. Der PicoBlaze-Prozessor ist bereits im Flash-Memory gespeichert und wird einschließlich einem *Monitorprogramm* in das Spartan-FPGA geladen.

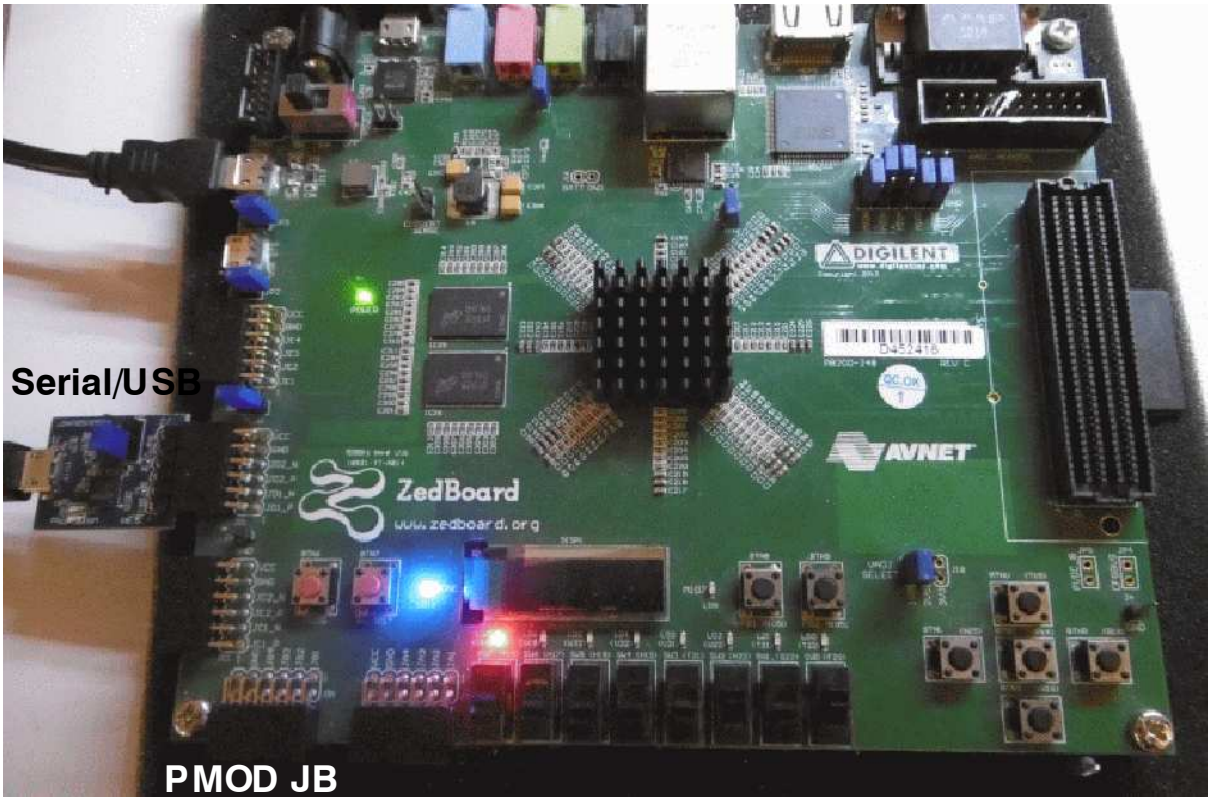


Bild 3.20: Zielsystem ZedBoard mit Znyq-7020 FPGA

Durch zusätzliche VHDL-Module wurden der PicoBlaze-Controller um folgende Schnittstellen erweitert:

- LED-Port = Ansteuerung einer Reihe von 8 LED
- SWITCH-Port = Einlesen der Schalterstellung von 8 Schiebeschaltern

- BUTTON-Port = Einlesen der Stellung von 4 Tastern (PBU ist nicht verfügbar – wird als Reset für PicoBlaze verwendet)
- Serieller Port = RS232-Schnittstelle (fest programmiert sind 115,2 KBit/s, 8 Bit Daten, keine Parität, 1 Stop-Bit). Die Parameter lassen sich nur über VHDL-Programme ändern.

21.2 Portdefinitionen Zenq-7020 (ZedBoard) NEW! [PicoBlaze 6]

Folgende Portadressen wurden durch entsprechende Kodierung von VHDL fest vergeben:

Adresse [hex]	INPUT	OUTPUT	OUTPUT (const.) OUTPUTK
00	UART_STAT_port	– –	– –
01	UART_RX6_port	UART_TX6_port	RESET_UART_port
02	SWITCH_port	LED_port	– –
03	PUSHBUT_port	– –	– –
04	DIN4_port	DOUT4_port	– –

21.2.1 Beschreibung der Ports:

UART_STAT_port (RD, ADDRESS= **00**):

7	6	5	4	3	2	1	0
0	0	RX full	RX half full	RX data present	TX full	TX half full	TX data present

UART_RX6_port (RD, ADDRESS= **01**):

7	6	5	4	3	2	1	0
RX7	RX6	RX5	RX4	RX3	RX2	RX1	RX0

SWITCH_port (RD, ADDRESS= **02**):

7	6	5	4	3	2	1	0
SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0

PUSHBUT_port (RD, ADDRESS= **03**):

7	6	5	4	3	2	1	0
0	0	0	0	PBL	PNC	PBR	PBD

DIN4_port (RD, ADDRESS= **04**, siehe Bild 3.21):

7	6	5	4	3	2	1	0
0	0	0	0	JB8	JB7	JB2	JB1

UART_TX6_port (WR, ADDRESS= **01**):

7	6	5	4	3	2	1	0
TX7	TX6	TX5	TX4	TX3	TX2	TX1	TX0

LED_port (WR, ADDRESS= **02**):

7	6	5	4	3	2	1	0
LED7	LED6	LED5	LED4	LED3	LED2	LED1	LED0

DOUT4_port routed through JB3, JB4, JB9, JB10 (WR, ADDRESS= **04**, siehe Bild 3.21):

7	6	5	4	3	2	1	0
X	X	X	X	X	b2h	b1h	ena

RESET_UART_port (WR CONSTANT, ADDRESS= **01**):

7	6	5	4	3	2	1	0
X	X	X	X	X	X	RX reset	TX reset

The pushbutton PBU (pushbutton "UP") is wired as a hardware "RESET" for PicoBlaze. This is the reason why it is not included in PUSHBUT_port.

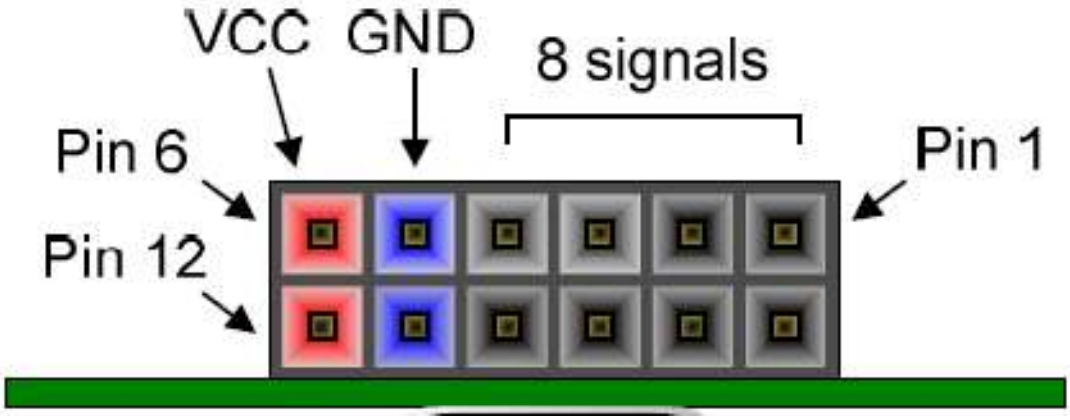


Bild 3.21: PMOD Connector (ZedBoard JB)

21.3 Programmentwicklung

PicoBlaze-Programme lasen sich mit der GUI "PicoTool" der HS Bremerhaven entwickeln.

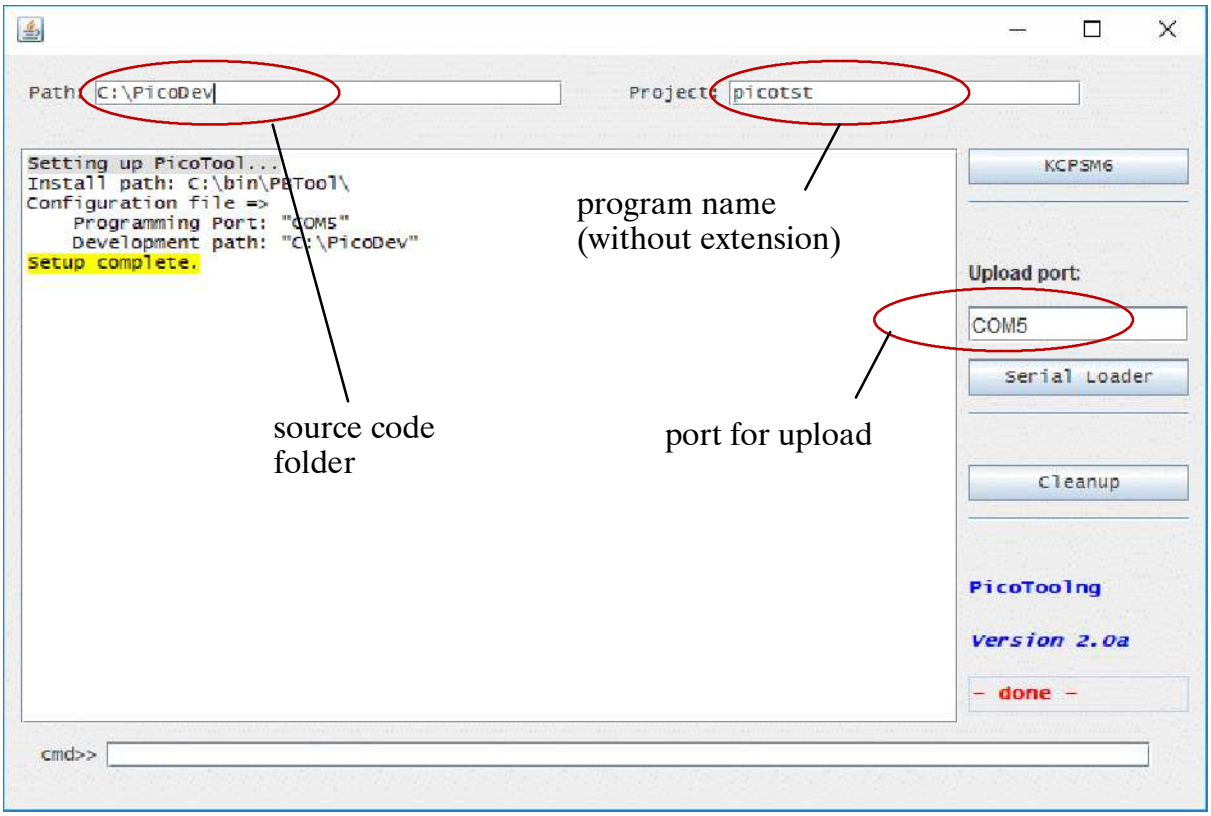


Bild 3.22: PicoBlaze Assembler / Programmer Tool for KCPSM6-Assembler

Mit dem Button “Assembler” wird der Sourcekode übersetzt. Eventuellen Fehlermeldungen werden im Text-Fenster angezeigt. Sollte das Programm fehlerfrei übersetzt worden sein (wie im obigen Beispiel), so kann mit dem Button “Programmer” das Programm in das FPGA geladen werden (s. Bild 3.23).

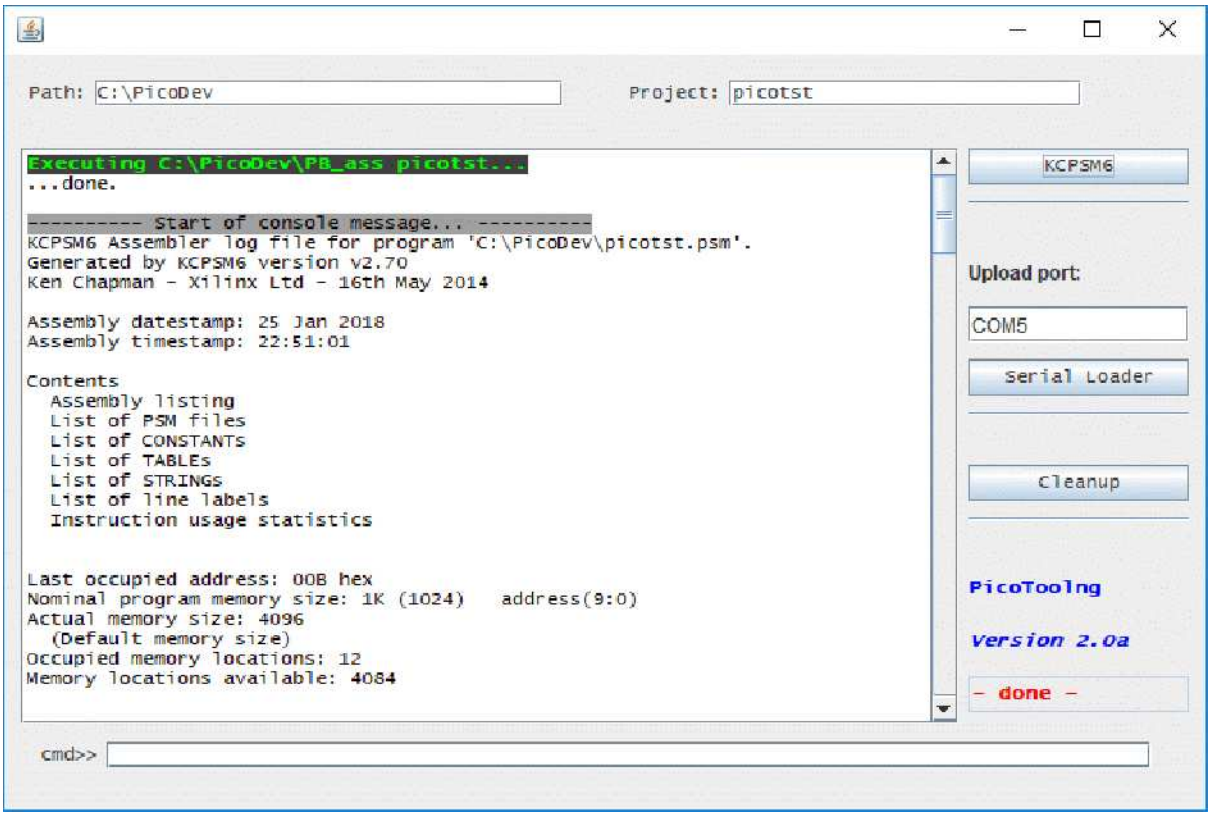


Bild 3.23: PicoBlaze Assembler / Programmer Tool: Assembler

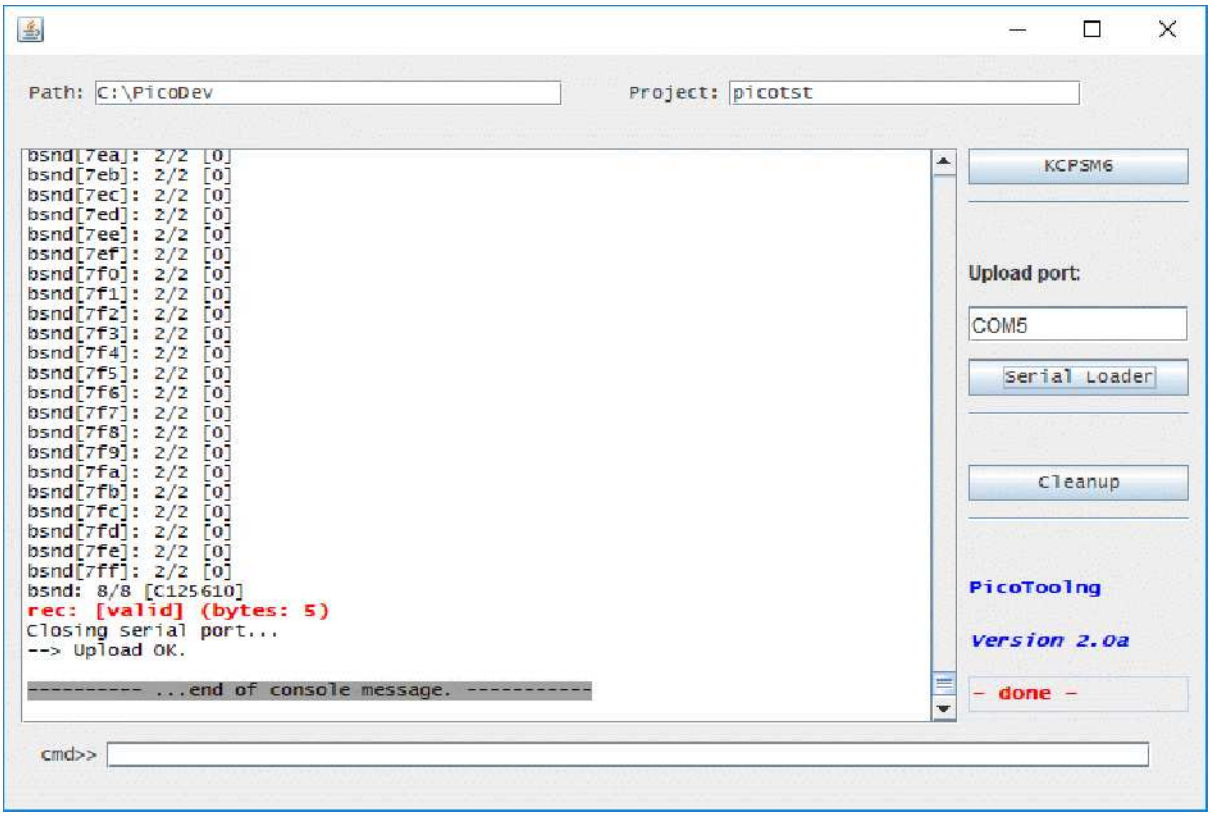


Bild 3.24: Upload eines neuen Programms

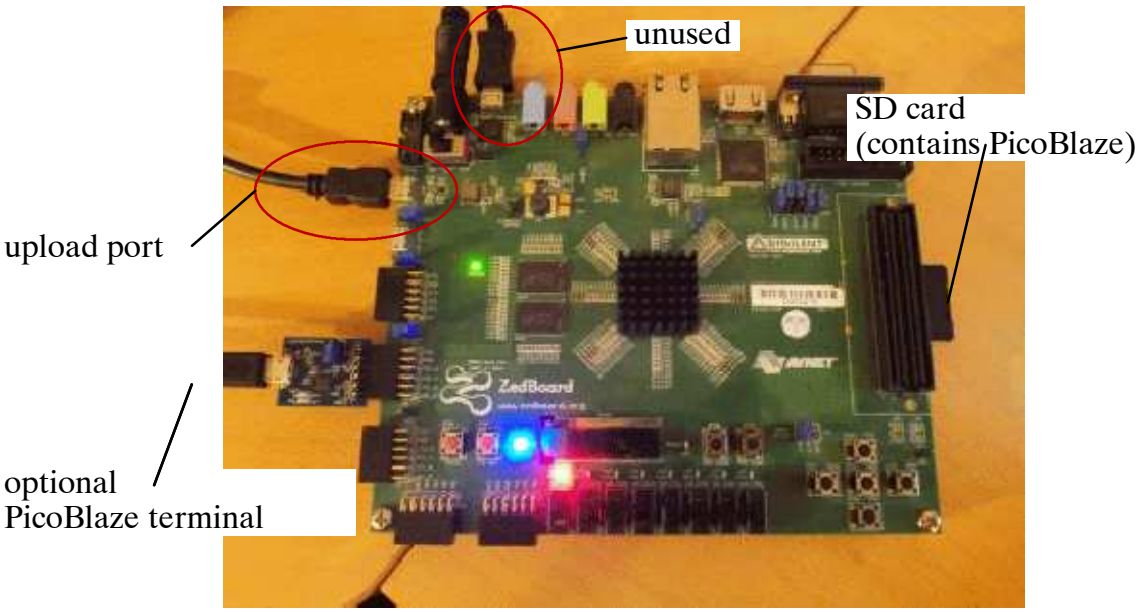


Bild 3.25: ZedBoard executing PicoBlaze Software

***

22 Mikroprozessorlabor

Lab #06:

22.1 Labor #06: Einführung in die Programmierung des PicoBlaze6 Mikrocontrollers

Für die Programmierung sind mehre Software-Werkzeuge erforderlich:

- Assembler:
  Der Assembler übersetzt mnemonischen Code in Binärcode, der vom Microcontroller direkt verarbeitet werden kann. So wird beispielsweise die mnemonische Form
  ADD sA, 01
  in den Binärcode
  11A01 (Hex)
  übersetzt. Gleichzeitig verarbeitet der Assembler Konstanten und Label (Adressen).
- Das Programm für den Assembler wird in einem dafür geeigneten Editor geschrieben. Alle Instruktionen, Zahlen, Assembler-Direktiven und Kommentare werden dabei farblich getrennt hervorgehoben.
- Debugger:
  Ein Debugger erlaubt die Überwachung aller internen Größen im Prozessor, der Ein- und Ausgänge sowie des Speichers. Das Programm wird dabei nicht auf dem eigentlichen Mikrocontroller ausgeführt, sondern durch Simulation der Befehle. Das Programm lässt sich in Einzelschritt ausführen oder es können “Breakpoints” gesetzt werden, an denen das Programm bei Erreichen der entsprechenden Stelle anhält.
- Programmer:
  Aufgabe des Programmers ist das Laden des übersetzten Programmes in den Arbeitsspeicher des Mikrocontrollers. Der Mikrocontroller wird dabei angehalten und sein Arbeitsspeicher neu beschrieben. Anschließend führt der Prozessor einen “Reset” (Neustart) durch.

22.1.1 Erstellen von Programmen mit Jedit

Programme sind in dem Verzeichnis **C:\PicoDev** zu erstellen. Ein Assemblerfile muss die Endung **.psm** tragen. Vor dem Punkt dürfen nur Namen verwendet werden, die keine Leerzeichen, keine Sonderzeichen (’§’, ’$’, ’%’, ’*’ usw.) enthalten. Die Anzahl der Zeichen sollte auf 8 beschränkt werden (bei neuen Assemblern nicht notwendig).

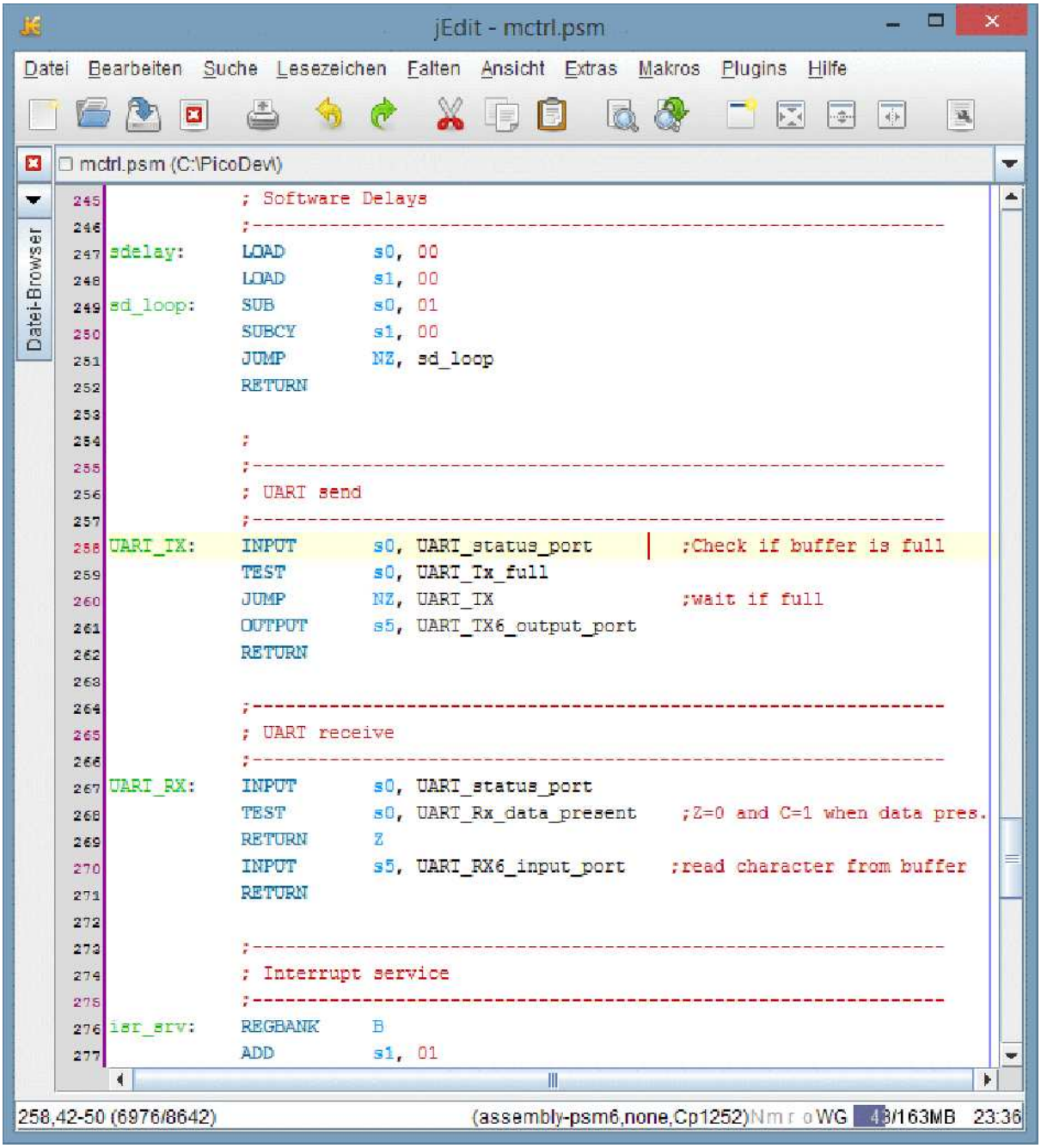


Bild 3.26: .psm-Dateien mit Jedit erstellt

Die Datei wird mit “CTRL–S” oder über Buttons gespeichert.

22.1.2 Assembler, Debugger und Programmer

Alle weiteren Schritte sind in der Oberfläche **PicoTool** zusammen gefasst.

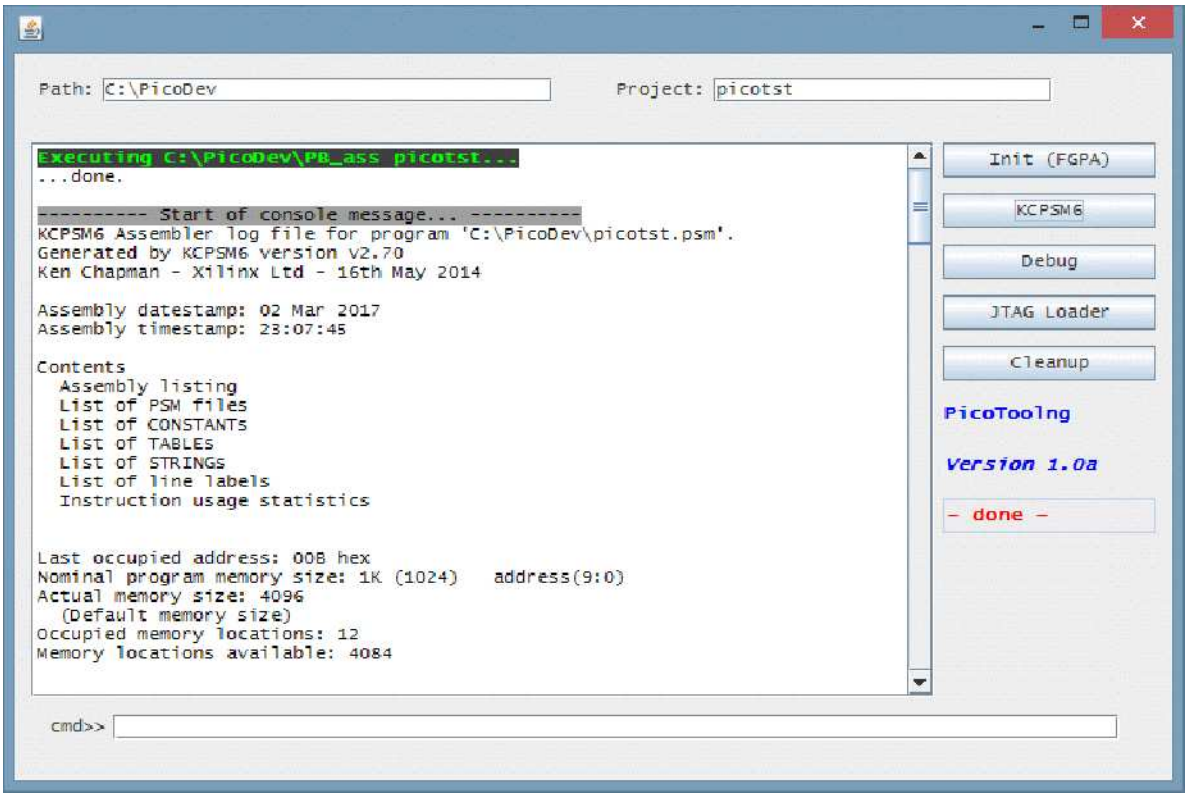


Bild 3.27: PicoTool Development Software

Die Menüs werden nur selten und auch nur zur Konfiguration benutzt. Die Steuerung erfolgt für den Entwicklungsvorgang durch die Buttonleiste (“assemble”, “debug”, “program” und “cleanup”).

WICHTIG: Tragen Sie in dem rechten Textfeld den Namen Ihrer “.psm”-Datei ein (**ohne die Endung .psm!**).

assemble:

Aufruf des Assemblers. Falls Fehler bei der Übersetzung auftreten, wird dies vom Assembler angezeigt. Bei erfolgreicher Übersetzung erscheint das Ergebnis (logfile) im Textwindow.

debug: (Mediatronix debugger – outdated)

Aufruf des Debuggers. Programme lassen sich vor dem Laden in den Mikrocontroller überprüfen.

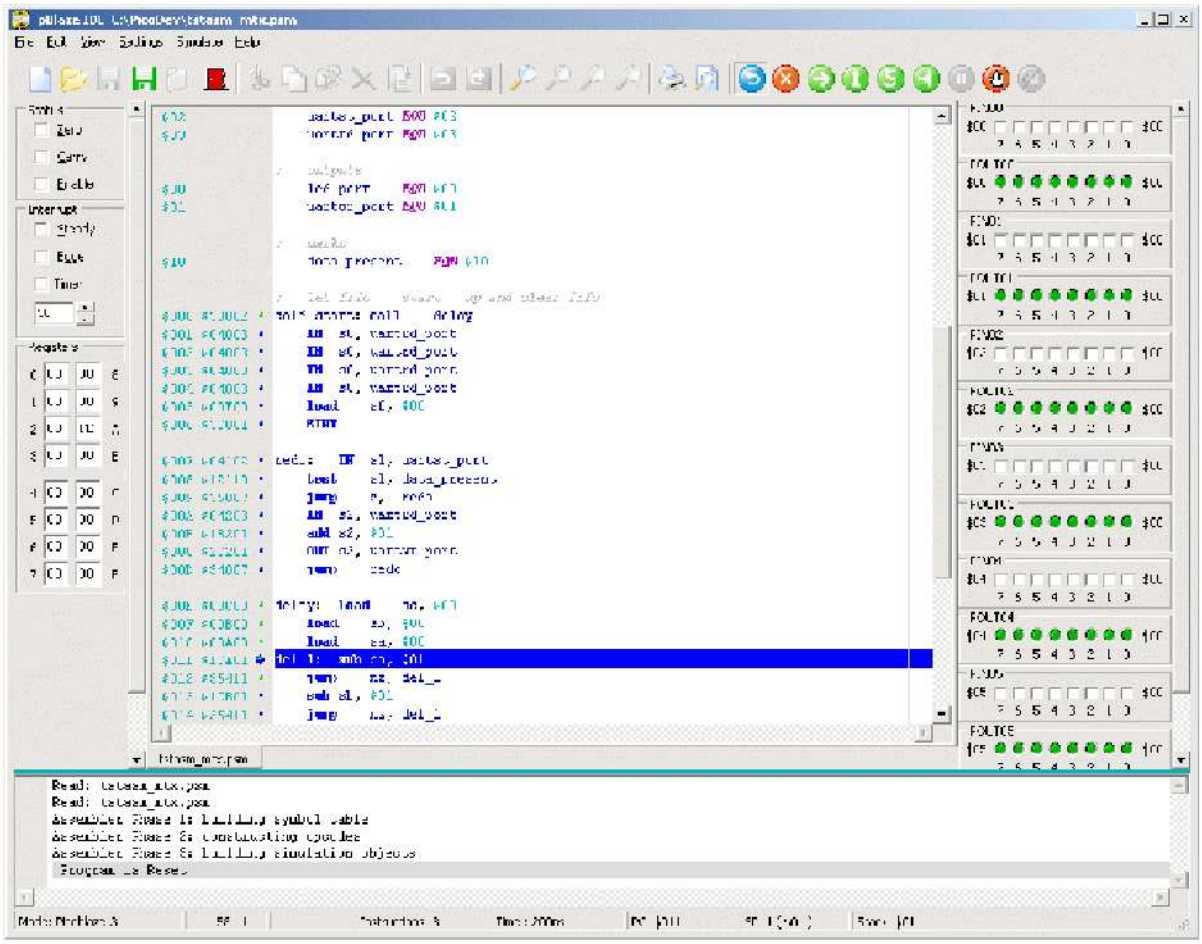


Bild 3.28: Mediatronix Debugger

program:

Laden des Programmes in den Mikrocontroller. Es wird eigentlich das FPGA neu konfiguriert. Dieser Vorgang dauert einige Sekunden – bitte Geduld.

cleanup:

Alle Schritte erfordern das Erzeugen einen Reihe von Dateien, die nach Abschluss der Schritte nicht mehr benötigt werden. Mit “cleanup tmp files” werden diese (überflüssigen) Dateien gelöscht. Alle Quellen bleiben natürlich erhalten.

Vor dem Beenden Ihrer Arbeit oder vor einem Wechsel des Projektnamens sollten Sie den Button “cleanup” betätigen.

- Erzeugen Sie eine neue Assemblerdatei “psmtest.psm” mit Jedit (nach Aufgabenstellungen an der Tafel)
- Übersetzen Sie das Programm und debuggen Sie Ihr Programm.

***

Lab #07:

22.2 Labor #07: Programmierung von logischen Funktionen (NOT, Flip, ON, OFF)

Entwickeln Sie ein Programm für den PicoBlaze™ Mikrocontroller, das logische Funktionen auf die eingelesenen Werte (Byte) der 8 Schiebeschalter (Port 03, SWITCH_port) anwendet. Das Ergebnis soll auf den 8 LEDs (Port 00, LED_port) angezeigt werden.

Die logische Funktion wird durch Betätigen einer der 4 Buttons (Port 02, BUTTON_port) ausgewählt:

- Kein Button betätigt ⇒ LEDs zeigen die Position der Schalter an.
- Button “East” betätigt ⇒ Byte wird invertiert.
- Button “North” betätigt ⇒ “Flip”-Operation auf das Byte wird angewandt, d.h. die Reihenfolge der Bits kehrt sich um:

  0 → 7
  1 → 6
  2 → 5
  3 → 4
  4 → 3
  5 → 2
  6 → 1
  7 → 0

- Button “South” betätigt ⇒ alle LEDs aus.
- Button “West” betätigt ⇒ alle LEDs an.

Zeichnen Sie *vor* Beginn der Programmierung ein **Struktogramm** (**Nassi-Shneiderman-Diagramm**), um den Algorithmus zu dokumentieren und eine zielgerichtete Programmierung zu ermöglichen (Top-Down-Design und *nicht Bottom-Down-Design*).

Die erfolgreiche Teilnahme an der Laborveranstaltung kann nur bescheinigt werden, wenn Struktogramm sowie *zugehöriges* Programm vorliegen.

Lab #08:

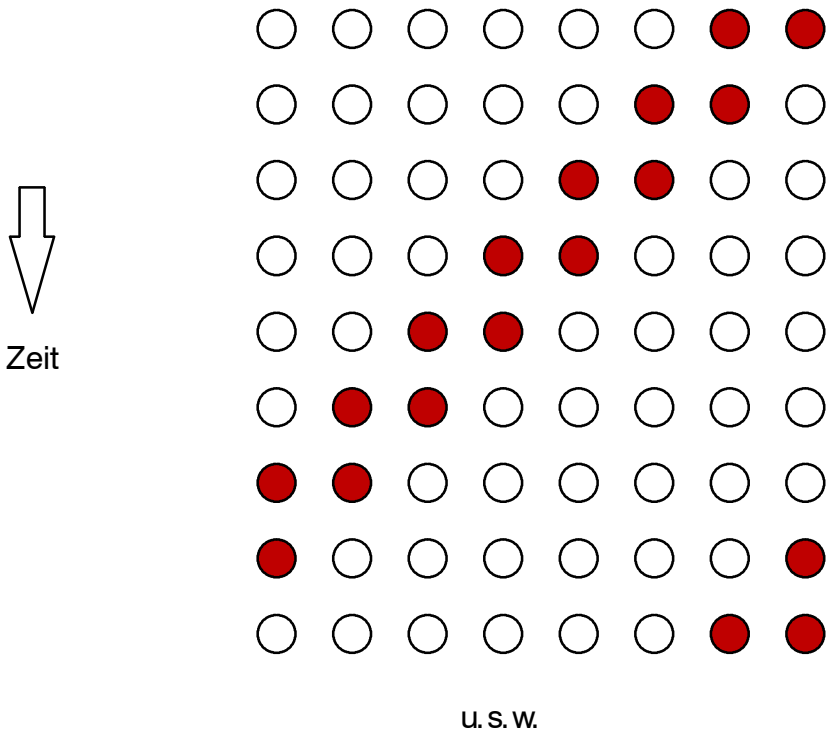
22.3 Labor #08: Lauflicht mit veränderlichem Muster

Entwickeln Sie ein Programm für den PicoBlaze™ Mikrocontroller, das ein Lauflicht auf den LEDs (Port 00, LED_port) verwirklicht.. Das Muster wird durch die Schiebeschalter (Port 03, SWITCH_port) vorgegeben.

Wird einer der 4 Buttons (Port 02, BUTTON_port) betätigt, so wird ein neues Muster von den Schiebeschaltern für das Lauflicht geladen.

- Frequenz des Lauflichts: 1-2Hz
- Mögliche Erweiterung: Umkehrung der Richtung des Lauflichts

Beispiel:



Zeichnen Sie *vor* Beginn der Programmierung ein **Struktogramm (Nassi-Shneiderman-Diagramm)**, um den Algorithmus zu dokumentieren und eine zielgerichtete Programmierung zu ermöglichen (Top-Down-Design und *nicht Bottom-Down-Design*).

Die erfolgreiche Teilnahme an der Laborveranstaltung kann nur bescheinigt werden, wenn Struktogramm sowie *zugehöriges* Programm vorliegen.

Lab #09:

22.4 Labor #09: Serielle Schnittstelle

Entwickeln Sie ein Programm für den PicoBlaze™ Mikrocontroller, das die Tastatur abfragt und bei Betätigung einer Taste folgenden Text ausgibt:

Zeichen “< character> ” wurde eingegeben.< CR>

< character> ist dabei das Zeichen, das von der Tastatur eingelesen wurde; < CR> ist “carriage return” (CHRASC_CR = $0D_H$). < CR> bewirkt eine neue Zeile auf dem Terminal.

Am Statusbit 4 des UART_status_ports (RX data present) erkennen Sie, wann ein Zeichen eingegeben wurde.

Beachten Sie, dass Sie nur Zeichen an die serielle Schnittstelle ausgeben dürfen (UART_write_port), wenn Bit 1 (TX full) = ’0’ ist. In diesem Fall kann in den Sende-FIFO-Puffer noch ein Zeichen geschrieben werden.

Lab #10:

22.5 Labor #10: Lottozahlengenerator mit Terminal-Abfrage

Entwickeln Sie ein Programm für den PicoBlaze™ Mikrocontroller, das Lottozahlen (Zahlen von 1−49) in (pseudo-)zufälliger Reihenfolge liefert. Die Zahlen liefert ein arithmetischer Zufallszahlengenerator auf der Basis von Primzahlen (linearer Kongruenzgenerator).

Der Pseudo-Zufallsgenerator soll 7 Zahlen von 1-49 liefern. Die nächstgrößere Primzahl ist 53. Der Algorithmus für einen einfachen Zufallszahlengenerator für die Zahlen von 1 bis 52 lautet

$$x := 19\,x \ \text{ modulo } \ 53\,. \tag{3.40}$$

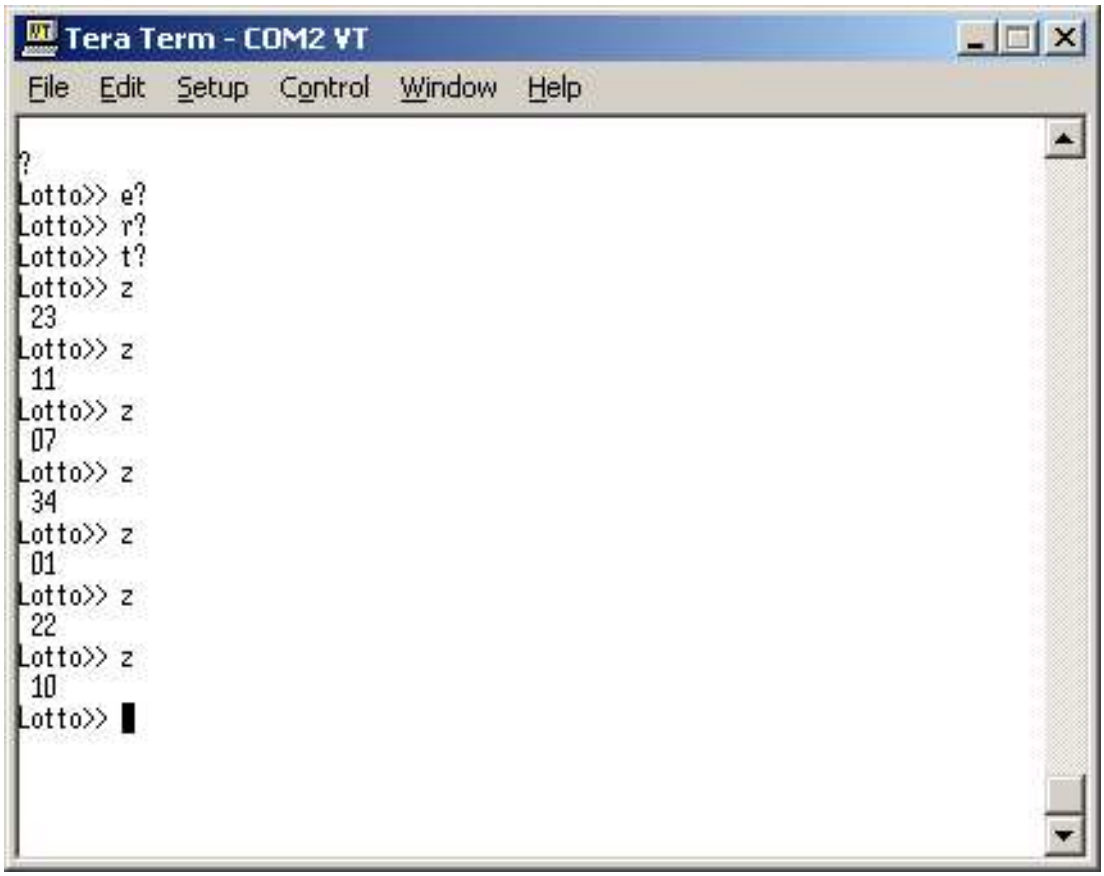
Der Anfangswert für die Variable x ist eine beliebige Zahl im Bereich 1..52. Zahlen größer als 49 werden einfach ignoriert und es kann ein neuer Wert aus dem Zufallszahlengenerator angefordert werden. Dieser Wert kann einem Zähler entnommen werden, der bei Tastenbetätigung ausgelesen wird und somit ebenfalls eine Zufallszahl darstellt.

Die Operationen Multiplikation und Modulo stehen im PicoBlaze™ nicht zur Verfügung und müssen aus Unterprogramm entwickelt werden.

Aus Tastendruck sollen sieben Lottozahlen ausgegeben werden. Beachten Sie, dass die Terminalausgabe eine Kodierung der Binärzahl in das ASCII-Format erfordert.

$0 \triangleq 30_H$
$1 \triangleq 31_H$
.
.
.
$9 \triangleq 39_H$

Die Kommunikation mit dem PC erfolgt über ein Terminalprogramm (z.B. “TeraTerm”).



Das Programm auf dem Mikrocontroller meldet sich mit dem Prompt “Lotto>> ” und erwartet die Eingabe eines Zeichens. Alle Zeichen werden bis auf “z” ignoriert. Wird das Zeichen “z” (ASCII $7A_{16}$) eingegeben, so soll eine gültige Lottozahl (1–49) ausgegeben werden und gleichzeitig in der 7-Segment-Anzeige (untere Stellen) angezeigt werden.

Zum Zugriff auf die serielle Schnittstelle und die übriges Peripherie des Mikrocontrollers können folgenden Vereinbarungen dienen (ACHTUNG: gilt nur für Spartan-3 Board, für das Spartan-3E Board gilt die Port-Belegung gemäß Umdruck):

```
;-- Inputs ------------------------------------------
        CONSTANT UART_status_port, 00   ; UART status input
        CONSTANT UART_read_port, 01     ; UART Rx data input
        ;
        CONSTANT BUTTON_port, 02        ; button input
        ;
        CONSTANT SWITCH_port, 03        ; switches input
        ;
        ;-- Outputs -----------------------------------
        CONSTANT LED_port, 00           ; LEDs output
        ;
        CONSTANT UART_write_port, 01    ; UART Tx data output
        ;
        CONSTANT SSEGLO_port, 02        ; 7-segment lo byte
        ;
        CONSTANT SSEGHI_port, 03        ; 7-segment hi byte
        ;
        CONSTANT SSDOTS_port, 04        ; 7-segment dots
```

Vor der Ausgabe auf die 7-Segment-Anzeige ist die Zahl in das dafür benötigte Format umzusetzen. Das gleiche gilt für die Darstellung auf dem Terminal (ASCII-Zeichensatz).

Hinweise zum Betrieb der seriellen Schnittstelle:

- Daten werden gesendet, indem auf den Port `UART_write_port` geschrieben wird. Dies darf aber nur erfolgen, wenn im UART_status_port das Bit 1 (TX full) '0' ist (Transmitter FIFO nicht voll).

UART_status_port (RD, 0x00):

7	6	5	4	3	2	1	0
–	–	–	RX data present	RX full	RX half full	TX full	TX half full

- Daten werden gelesen, indem `UART_read_port` in ein Register kopiert wird. Dies darf jedoch nur erfolgen, wenn auch Daten vorhanden sind Bit 4 (RX data present) des UART_status_port muss '1' sein.

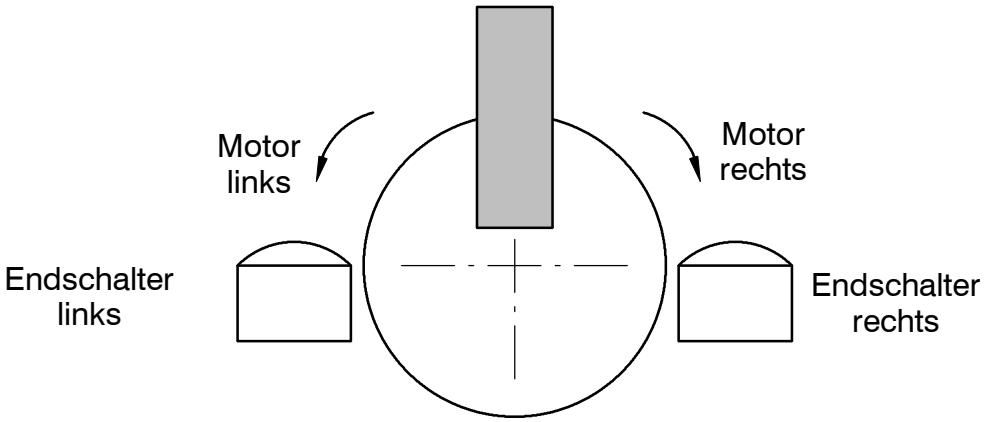
Zeichnen Sie *vor* Beginn der Programmierung ein **Struktogramm (Nassi-Shneiderman-Diagramm)**, um den Algorithmus zu dokumentieren und eine zielgerichtete Programmierung zu ermöglichen.

***

Lab #11:

22.6 Labor #11: Motorsteuerung (Labor- und Klausuraufgabe!)

Entwickeln Sie ein Programm für den PicoBlaze™ Mikrocontroller, mit dem sich ein Motor mit Endschaltern steuern lässt.



Über den DIN_port (04_H, RD) lassen sich die Endschalter erfassen:

7	6	5	4	3	2	1	0
X	X	X	X	X	Encoder	ES left	ES right

Der Motor kann mit zwei Signalen in Rechts- oder Linkslauf versetzt werden.
DOUT_port (04_H, WR):

7	6	5	4	3	2	1	0
X	X	X	X	X	M left	M right	M enable

Damit sind folgende Betriebsarten möglich:

M left	M right	M enable	Betriebsart
0	0	0	alles aus
0	0	1	Leistungsverst. an (Motor Stillstand)
0	1	1	Motor rechts
1	0	1	Motos links

Andere Betriebsarten sind nicht zulässig bzw. erforderlich.

Das Programm soll über das Terminal bedient werden. Folgende Befehle müssen verarbeitet werden:

- 'h': Motor aus (**h**alt)
- 'l': Motor dreht **l**inks bis zum linken Anschlag. Dort ist der Motor anzuhalten.
- 'r': Motor dreht **r**echts bis zum rechten Anschlag. Dort ist der Motor anzuhalten.
- 'c': Motor hält in der Mittelposition an (**c**enter). Hierzu ist die Zeit für einen vollständigen Verstellvorgang in einem 32-Bit-Zähler zu messen. Anschließend soll der Motor die halbe Zeit in der Gegenrichtung betrieben werden.

Hinweis: Ein 32-Bit-Zähler lässt sich beispielsweise durch folgende Programmsequenz erzeugen:

```
ADD     sA, 01
ADDCY   sB, 00
ADDCY   sC, 00
ADDCY   sD, 00
```

Das Rückwärtszählen könnte durch

```
SUB     sA, 01
SUBCY   sB, 00
SUBCY   sC, 00
SUBCY   sD, 00
```

erfolgen.

Um die Zeit genau zu messen, ist ein **Timer-Interrupt** erforderlich. Der Interrupt wird mit einer Periode $T_S = 1\text{ms}$ ausgelöst. Der Timer ist bereits Bestandteil des Microcontrollers. Der Interrupt-Service (ISR) hat folgende grundsätzliche Struktur:

```
isr_srv:        •
                •
                •
                RETURNI ENABLE

ADDRESS 3FF
JUMP    isr_srv
```

Zeichnen Sie *vor* Beginn der Programmierung ein **Zustandsdiagramm**, um den Ablauf zu dokumentieren und eine zielgerichtete Programmierung zu ermöglichen. Das Diagramm ist Bestandteil der Lösung.

***

23 Literatur

[1] Heesel, N. und W. Reichstein: Mikrocontroller Praxis.
Vieweg, 1996

[2] Limbach, S.: Kompaktkurs Mikrocontroller.
Vieweg, 2002

[3] Raisonance 80C51 and XA Development Tools Manual.
Raisonance S.A., 2000

[4] K. Urbanski u. R. Woitowitz: Digitaltechnik.
Springer, 2000

[5] J. Wakerly: Digital Design: Principles and Practices.
Prentice-Hall, 1999

[6] Xilinx Vivado Users's Guide.
Xilinx Corp., 2018