IN Grundlagen der Informatik

Prof.Dr. Raimond Zavodnik

I.Einführung:

1.1. Was ist Informatik?

Informatik = Zusammensetzung aus **Infor**mation und Auto**matik**
Duden: Die Wissenschaft von der automatischen Verarbeitung von Information/Daten mit Hilfe von Computern.

1.2. Arbeitsgebiete der Informatik

1.2.1. Theoretische Informatik

Grundlegende Strukturen und Prozesse werden mit mathematischen Hilfsmitteln modelliert und untersucht.

- Theorie der Programmerstellung
- Theorie der formalen Sprache
- Theorie der Berechenbarkeit
- Automatentheorie [Hilfsmittel]
- Komplexitätstheorie

1.2.2. Praktische Informatik

Methoden zur Umsetzung von Problenlösungen auf dem Computer

- Programmiersprachen und Compiler
- Betriebssysteme
- KI
- Graphik
- Rechnerkommunikation (z.B. Netzwerke)

1.2.3. Technische Informatik

Aufbau von Rechnern (Hardware & Firmware)

- Hardwarekomponenten
- Mikroprogrammierung
- Computerarchitektur
- Rechnernetze

1.2.4. angewandte Informatik

Umsetzung von Problemlösungen aus anderen Fachgebieten auf dem Computer.

- Anwendungen aus Technik, Wirtschaft, Medizin, Verwaltung
- CAD/CAM
- Visualisierung/Animation
- Multimedia

1.3. historischer Abriß

s. Blatt!

1.4. Daten und Datenverarbeitung

Daten bestehen aus

- Zahlen (z.B. 3 ; 1.41459)
- Buchstaben (z.B. ‘c‘,‘X‘.‘2‘)
- Steuerzeichen (z.B. ^C ,‘\n‘)
- Maschinenkommandos (z.B. 1 0 0 X Y)

Prinzipien der Datenverarbeitung (EVA):

Eingabe → **V**erarbeitung → **A**usgabe

Daten werden auf EDV-Anlagen mittels eines Programmes verarbeitet.
Von Neumann: Programm wird in die EDV-Anlage eingelesen und gespeichert.
Daten (inklusive Programme) werden in den Hauptspeicher geladen.

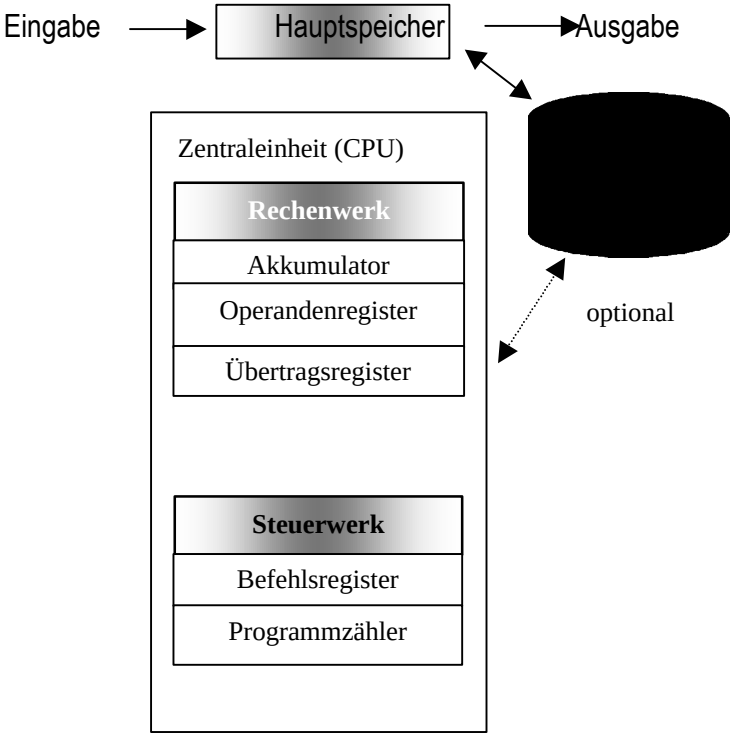
Gehirn: assoziativer Speicher, d.h. Inhaltsorientiert
Computer: Addressenorientierter Speicher: Daten werden in Speicherzellen abgelegt:

0	Speicherzelle 0
.	.
.	.
.	.
.	.
.	.
.	.
.	.
.	.
M	Speicherzelle M

i Addresse einer Speicherzelle
(i) Inhalt der Speicherzelle mit der Addresse i

$m = 2^n$
n=16 → m=65535

1.5. Aufbau und Funtionsweise von EDV-Anlagen

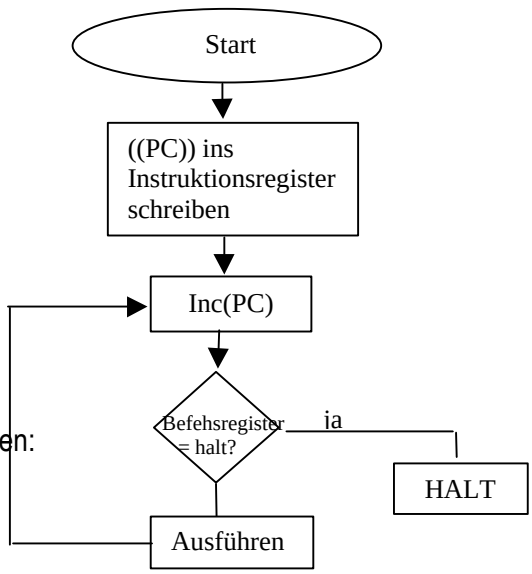


Eingabe: Aufnahme und Einschreiben von Daten und Programmen
CPU: Bearbeitung der Daten gemäß Programm
Steuerwerk: Holen und Entschlüsselung der Befehle
Rechenwerk: Abarbeitung der Befehle
externer Speicher: Magnetband/ -Karte, Floppy, HDD, Zip-Disk,...
Ausgabe: Darstellung der berechneten Ergebnisse

(Akkumulator = Speicher für Zwischenergebnisse)

Installationszyklus:

Programmzähler (PC) → erster Programmbefehl



Befehlsarten:

- Arithmetische Befehle (+,-,*,/,...)
- Logische befehle (AND,OR, in Pascal:Mengen)
- Sprungbefehle (bedingt/unbedingt)
- E/A (i/o)
- HALT

Bemerkungen:

- Sprungbefehle können den Inhalt des PC ändern
- Ausführung des sich im Hauptspeicher befindenden Programmes besteht aus einer Folge verschiedender Maschinenbefehle (nur Lesen!)
- Daten befinden sich in einem anderen „Programmsegment“ wie das eigentliche Programm!
- Daten sind r/w – Befehle dagegen nur lesbar!
- Programmbefehle befinden sich in aufeinanderfolgenden (=consecutiven) Speicherzellen
- Sequentielle Ausführung wird modifiziert durch Sprungebfehle

II. Datendarstellung in digitalen Rechnern

2.1.Zahlensysteme

2.1.1. Dezimalzahlen

ganze dezimale Zahlen Z_{10} lassen sich als Summe von Potenzen der Basiszahl (Radix) 10 schreiben:

$Z_{10} = a_{n-1}10^{n-1} + a_{n-2}10^{n-2} + ... + a_1 10^1 + a_0 10^0$ ($0 \leq a_i \leq 9$; n = Anzahl der Dezimalstellen von Z)

Beispiel:

$Z = 5*10^3 + 3*10^2 + 2*10 + 4 = 5324_{10}$

Man kann Z auch so darstellen:

$Z = (a_{n-1}a_{n-2}...a_1a_0)_{10}$

Reelle (*positive*) Dezimalzahlen entstehen wenn man negative Exponenten zuläßt.

$X_{10} = a_{n-1}10^{n-1} + a_{n-2}10^{n-2} + ... + a_1 10^1 + a_0 10^0 + a_{-1}10^{-1} + a_{-2}10^{-2} + ... + a_{-m}10^{-m}$ ($0 \leq a_i \leq 9$; n = Anzahl der Vorkommastellen von X ; m = Anzahl der Nachkommastellen von X)

Integerteil: $a_{n-1}10^{n-1} + a_{n-2}10^{n-2} + ... + a_1 10^1 + a_0 10^0$

Bruchteil: $a_{-1}10^{-1} + a_{-2}10^{-2} + ... + a_{-m}10^{-m}$

X_{10} = Integerteil + Bruchteil

Beispiel:

1.) $2.75 = 2*10^0 + 7*10^{-1} + 5*10^{-2}$

2.) $\Diamond = 3.1415927... = 3*10^0 + 1*10^{-1} + 4*10^{-2} + 1*10^{-3} + 5*10^{-4} + ...$
(m=∋)

2.1.2. polyarische Zahlendarstellungen:

Die Darstellungsweise funktioniert wie bei den Dezimalzahlen.
Jedoch gelte folgendes:

b sei eine beliebige ganze Zahl >1

$Z_b = a_{n-1}b^{n-1} + a_{n-2}b^{n-2} + ... + a_1b^1 + a_0b^0$

$= (a_{n-1}...a_0)_b$

Beispiel:

$(431)_6 = 4*6^2 + 3*6^1 + 1*6^0 = 144 + 18 + 1 = (163)_{10}$

auch Bruchteile sind möglich:

$X_b = a_{n-1}b^{n-1} + a_{n-2}b^{n-2} + ... + a_1b^1 + a_0b^0 + a_{-1}b^{-1} + a_{-2}b^{-2} + ... + a_{-m}b^{-m}$

Beispiel:

$(431.3)_6 = 4*6^2 + 3*6^1 + 1*6^0 + 3*6^{-1}$

Bemerkung:

$0 \leq a_i \leq b-1$ für *alle* i

a_i = „Ziffer“
b = Basis (Radix) der Darstellung
m = Anzahl der Vorkommastellen
n = Anzahl der Nachkommastellen

Überblick über gebräuchliche Zahlensysteme:

	System	Basis	Zahlenvorrat
1.	standard binär/dual	2	0,1
2.	tertiär, trial	3	0,1,2
3.	oktal	8	0,1,2,3,4,5,6,7
4.	dezimal	10	0,1,2,3,4,5,6,7,8,9
5.	hexadezimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Beispiel:

$(AC3)_{16} = A*16^2 + C*16^1 + 3*16^0 = 2755_{10}$

Bemerkung:

Die hexadezimale Schreibweise ist u.a. eine übersichtliche Vierergruppierung von Dualzahlen!

Hexadezimal	Binär
0	0000
1	0001
2	0010
3	0011
:	:
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Beispiel:

$(AC3)_{16}$ =

$$= (1010 \mid 1100 \mid 0011)_2$$
$$\quad A \quad\; C \quad\; 3$$

Durch succesive Herausfaktorisierung der Potenzen b aus den Teilsummen (statt jede Potenz einzeln zu berechnen) ergibt sich eine opitmal effiziente Auswertung und Umwandlung von „polyarischen" Zahlen (Hornerschema/~Methode):

$((...(a_{n-1}b+a_{n-2})b...+a_2)b+a_0)b+a_1$

2.1.3. Umwandlung von Dezimalzahlen – Restwertmethode:

Wiederholte Teilung duch b liefert die Koeffizienten a_i (im b-System) in *umgekehrter* Reihenfolge; d.h. von a_0 bis a_{n-i} .
Die nächste Teilung bezieht sich auf das ganzahlige Ergebnis (DIV) der vorhergehenden Teilung!

Beispiel:

$(671)_{10} = (???)_3$

671:3 = 223 Rest 2 (a_0)
223:3 = 74 Rest 1 (a_1)
74:3 = 24 Rest 2 (a_2)
24:3 = 8 Rest 0 (a_3)
8:3 = 2 Rest 2 (a_4)
2:3 = 0 Rest 2 (a_5)

$(671)_{10} = (220212)_3$

Übung:

$(17623)_{10} = (????)_{16}$

2.1.4. Umwandlung reeller (aber gebrochener) Dezimalzahlen

Wir können annehmen: Die Zahl x ist zwichen 0 und 1

Bemerkung: Es ist *nicht immer* möglich aus einer exakten Dezimaldarstellung eine exakte Darstellung im Zielsystem zu erlangen!

> Übung: Finden Sie eine solche Zahl (X_{10} exakt, aber es existiert keine exakte Darstellung in einem Zielsystem!

Für die Umwandlung wendet man wieder die Horner-Methode an:

Man multipliziere x mit b (b ist Basis des Zielsystems!):

$x= a_{-1}b^{-1} + ... + a_{-m}b^{-m}$
$x*b= (a_{-1}b^{-1} + ... + a_{-m}b^{-m})*b = a_{-1} + b^{-1}(a_{-2} + ... + a_{-m}b^{-m+2})...$

Ergebnis: a_{-1} wird isoliert und der Rest nimmt die Form des ursprünglichen

Bruchteils an, d.h. das Verfahren wird wiederholt (iterativ fortgesetzt) durch Multiplikation des Restes mit b usw.

Beispiel:

$(0.375)_{10} = (????)_2$

$0.375*2 = \mathbf{0}.75 \quad a_{-1} = 0$
$0.75*2 = \mathbf{1}.50 \quad a_{-2} = 1$
$0.50*2 = \mathbf{1}.\underline{00} \quad a_{-3} = 1$

Da der Bruchteil gleich Null ist, ist die Darstellung exakt!

$(0.375)_{10} = (0.011)_2$

> Übung:
>
> $(0.2)_{10} = (????)_2$
>
> Ist die Darstellung exakt?
> Erkennen Sie ein Muster?
>
> $(0.8)_{10} = (????)_8$

2.1.5. Gleitkommazahlen

Festkommazahlen = „Integerzahlen mit Komma“

Beispiel:

29.15 + 12.23 = ???

```
  2915        Komma: -2 (=2 Stellen nach rechts verchieben!)
+1233         Ergebnis: 41.38
------
  4138        „Normalisierung“
```

Nachteil: Diese Methode ist nicht geeignet für Berechnungen,bei denen die Kommastelle dynamisch bestimmt wird, insbesondere bei sehr großen und sehr kleinen Zahlen!

Beispiel:

Avogadro’s Konstante: $N = 6.02252 * 10^{23}$
Max Plank’s Konstante: $h = 1.0545 * 10^{-27}$ erg/s

Eine Gleitkommadarstellung besteht aus:

b = Basis der Gleitkommadarstellung
q = Exzesszahl (um negative Exponenten darstellen zu können)
m = „Mantisse“ (Bruchteil)/signifikante Ziffern (normalerweise $|m| \geq 1$)
p = Anzahl der Ziffern in m
e = Exponent $0 \leq e \leq E$ (INTEGER!)
v = Vorzeichen (0/1)

$$X = v * m * b^{e-q} = \Gamma \; .a_1 \ldots a_p * b^{e-q}$$

Beispiel:

b = 10 , q=50 , p=8

$N = +.60225200 * 10^{74-50}$ e=74
$n = +.10545000 * 10^{24-50}$

X heisst normalisiert falls $a_{-1} \cong 0$, so dass $1/b \leq |m| < 1$ oder m=0 *und* e=0
In der Praxis wird i.d.R. nur mit normalisierten Zahlen gearbeitet; d.h. die Grundrechenarten werden meist nur für normalisierte Operanden definiert!

Algorithmus:

Falls |m|>1 ist wird m solange duch b dividiert (*Shift*-Operation!) bis |m|<1 ist.
Bei jeder Teilung wird e um 1 erhöht!
Falls (dadurch) |m|<1/b wird, wird m solange mit b multipliziert (*Shift*-Operation!) bis 1 > |m| Σ 1/b ist und e dabei jedesmal um 1 verringert.

Schließlich muß m um p Stellen gerundet werden.
Falls m dadurch 1 wird, muß nachträglich m nochmals durch b dividiert und e um 1 erhöht werden.

Bei normalisierten Gleitkommazahlen:

- kleinste positive Zahl $X_{min} = X_{min}(b.e.p.q)$
  $X_{min}=0*b-b^{0-q} = b^{-(q+1)}$

- größte positive Zahl $X_{max}=X_{max}(b,e,p,q)$
  $X_{max}=0.(b-1)(b-1)...(b-1)*b^{E-q} =$

$$= \left(\frac{b-1}{b^1} + \frac{b-1}{b^2} + \cdots + \frac{b-1}{b^p} \right) * b^{E-q} =$$

$$= \frac{b-1}{b} * \left(\frac{1-\left(\frac{1}{b}\right)^p}{1-\frac{1}{b}} \right) * b^{E-q} =$$

$$= \left(1-b^{-p}\right) * b^{E-q}$$

Vorteil: bei normalisierter Gleitkommaartihmetik maximale Genauigkeit!
Nachteil: ungenaue Ergebnisse kommen vor!

Vorgehensweise für die Gleitkommaartihmetik

1.Addition/Subtraktion:

$X_1=v_1m_1b^{e1-q} = \Gamma a_1...a_pb^{e1-q}$
$X_2=v_2m_2b^{e2-q} = \Gamma a_1...a_pb^{e2-q}$

X_1,X_2 seien normalisiert!

Man kann annehmen: $e_1>e_2$ (sonst tauschen!).

Dann schreibe man X_2 um:

$$X_2 = \pm c_1 \ldots c_p b^{e_1 - q} =$$
$$= \pm \left(\frac{c_1}{b} + \ldots + \frac{c_p}{b^p} \right) \frac{1}{b^{(e_1 - e_2)}} b^{e_1 - q} =$$
$$= \pm \left(\frac{c_1}{b^{(e_1 - e_2)+1}} + \ldots + \frac{c_p}{b^{(e_1 - e_2)+p}} \right) b^{e_1 - q}$$

$$X_1 \oplus X_2 = \pm \left(\frac{a_1}{b} + \ldots + \frac{a_p}{b^p} + \frac{c_1}{b^{(e_1 - e_2)+1}} + \ldots + \frac{c_p}{b^{(e_1 - e_2)+p}} \right) b^{e_1 - q}$$

X_1 I X_2 = Normalisierung (X_1IX_2)

X_1 ϑ X_2 = X_1 I (-X_2)

Achtung:
Es ist nicht immer wahr, dass (X_1 I X_2) I X_3 = X_1 I (X_2 I X_3) gilt!

Ubung: Finden Sie drei Gleitkommazahlen, deren Addition *nicht* kommutativ ist!

Beispiel: b=2,p=8,q=0 (wird untrdrückt!)
$(2.25)_{10}$ I $(6.375)_{10}$ = ???

1.Umwandlug in binär:

$(2.25)_{10} = (10.01)_2$
$(6.375)_{10} = (110.011)_2$

2.Operanden normalisieren:

$(2.25)_{10} = (0.1001*2^2) = 0.10010000*2^2$ (wegen p=8!)
$(6.375)_{10} = 0.11001100*2^3$

3.kleinere Zahl anpassen:

$(2.25)_{10} = 0.01001000*2^3$

4.Mantissen addieren:

m_1 = 01001000
I m_2 = 11001100
= 100010100
= $1.00010100*2^3$

6. Ergebnis normalisieren:

$1.00010100*2^3 = .10001010*2^4$

Regeln für binäre Addition:

0 + 0 = 0
0 + 1 = 1 + 0 = 1
1 +1 = 0 Übertrag 1

Ergebnis prüfen:

$$1000.1010 = 1*2^3 + 0 + 0 + 0 + 1*2^{-1} + 0 + 1*2^{-4} + 0 =$$
$$= 8 + 0.5 + 0.125 = 8.625$$

Übung:

1. b=10, p=8, q=0

   + .98765432*10^50 I .3333333333333*10^49 = ??

2. b=10, p=8, q=0

   + .50000001*10^45 I .10000000*10^54 = ??

3. b=2, p=16, q=0

   $(2.234)_{10}$ I $(0.375)_{10}$ = ??
   (etwas langwierig!)

Bemerkung: Falls $e_1 - e_2 > p + 1$ ist, wird die Addition überflüssig,
d.h. Ergebnis = X_1

Multiplikation: $e_3 = (e_1 + e_2) - q$
$m_3 = m_1 * m_2$ (Binärmultiplikation!)

$$e_3 = (e_1 - e_2) + q + 1$$

Division:

$$m_3 = b^{-1} \frac{m_1}{m_2}, \; m_2 \neq 0$$

Bemerkung: $|m_3| = \frac{1}{b} \frac{|m_1|}{|m_2|} \leq \frac{1}{b} \frac{1}{\frac{1}{b}} = 1$

$$|m_3| \geq \frac{1}{b} \frac{\frac{1}{b}}{1} = \frac{1}{b^2}$$

⇨ Normalisierung ist notwendig!

2.2. Numerische Datenformate im Rechner

2.2.1. Das Maschinenwort

Da Digitalrechner Information durch das Vorhandensein (1) bzw Nichtvorhandensein (0) einer Schwellenspannung speichert

(FilpFlops!), ist das Binärsystem „natürlich“ das System in der rechnerinternen Darstellung von Daten.

1 Bit = kleinste Darstelungseinheit im Binärsystem, d.h. nur zwei mögliche Zustände (1/0 = ein/aus) werden angenommen (nicht addressierbar!)

1 Byte = kleinste Gruppierung von Bits in einem Speicher, die evtl. addressierbar ist. Heute besteht diese Gruppierung fast ausschließlich aus 8 Bits.
(Bits könen 256 verschiedene Binärwerte annehmen:

$00000000_2 = (00)_{16} = 0$
$00000001_2 = (01)_{16} = 1$
. . .
. . . 256
. . .
$11111111_2 = (FF)_{16} = 256$

1 Maschinenwort = Gruppierung von Bits zu einer im Rechner einsatzfähigen Einheit. Sehr häufig ist ein Maschinenwort ein ganzzahliges Vielfaches von Bytes (sog. „byteorientierte“ Rechner).

Beispiele:

Telefunken Tel440	52 Bits
CDC	60 Bits
Sperry Umac	36 Bits
Microcomputer	4/8/16/32/64 Bits

2.2.2. Darstellung von ganzen Zahlen im Rechner

Thema: Interpretation von Bitfolgen

2.2.2.1. Vorzeichen und Betrag:

Bei jedem Rechner (heute) wird eine bestimmta Anzahl n von Bytes für die Zahlendarstellung festgelegt. Diese Anzahl ist maschinenabhängig!

A) n = 2 (Bytes) Zahlenbereich *unsigned* Integer und Festkommazahlen:

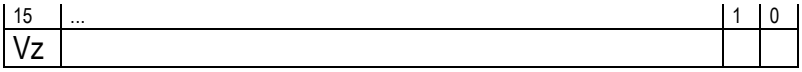
00000000	00000000	⇔ 0
:	:	
11111111	11111111	⇔ $2^{16} – 1 = 65535$

Falls keine negativen Zahlen benötigt werden reicht diese Darstellung aus,

z.B. bei der Darstellung von Speicheraddressen.

n = 4 ergibt einen *unsigned* Zahlenbereich von 0 bis 2^{32} –1 (= 4294967295)

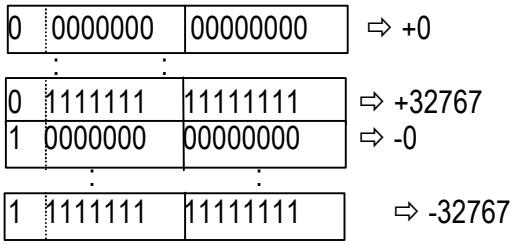
B) n= 2 (Bytes) Zahlenbereich Integer *mit* Vorzeichen:
Falls negative Zahlen dargestellt werden sollen, muss ein Bit („höchstes“) als Vorzeichen dienen!



Vz=0 ⇨ Die Zahl ist positiv; Vz = 1 ⇨ Die Zahl ist negativ

Bemerkung: Die eigentliche physikalische Reihenfolge der einzelnen Bits, die die successiven Stellenwerte beispielsweise im Dualsystem darstellt, ist maschinenabhängig und wird daher hier nicht diskutiert!

⇨ Zahlenbereich:



⇨ kein Optimalsystem!

Bemerkung: 1.) ***G 0 !!***
2.) Integer-Operationen sind schwer in Hardware zu realisieren!

2.2.2.2. Einkomplement-Darstellung (n=2, n=4)

- schnelle Kombination von Integerzahlen.
- Positive Zahlen wie 2.2.2.1. B
- Negative Zaheln werden durch bitweise Komplementierung von positiven Zahlen gebildet.

Beispiel (n=2):

+211 ⇨ 0000000011010011 $00D3_{16}$
⇨ -211 ⇨ 1111111100101100 $FF2C_{16}$

Zahl	Bitfolge (16/32-Stellig) n=2/n=4
0	000...0
1	000...01
2	000...10
:	:
32767	011...11 größte positive Zahl (= 2147483647 für n = 4)
-32768	100...00 größte negative Zahl (=-2147483648 für n = 4)
:	:
-2	111...01
-1	111...10
-0	111...11

Problem: Γ0 !

2.2.2.3. Zweierkomplementdarstellung:

- Erlaubt schnelle Integerarithmetik ohne 2x multiplikation!
- positive Zahlen wie 2.2.2.2.
- um eine negative ganze Zahl –N zu bilden geht man wie folgt vor:

  1. +N bilden
  2. Das Einerkomplement ~N wie in 2.2.2.2. bilden
  3. ~N+1 stellenweise addieren (*ohne* Übertrag!)

  Beispiel:

  $N= -1607_{10}$, n=2

  1607_{10} = 0000 0110 0100 0111 = 0647_{16}
  ~1607 = 1111 1001 1011 1000
  +1

  Ergebnis: 1111 1001 1011 1001 = $F9B9_{16}$

Übung:

a) N= -10
b) N= 0
c) Zahlenbereich für ganze Zahlen bei n=1 (Byte)

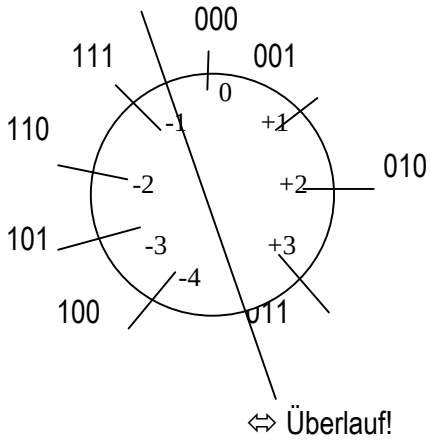
Zahlenbereich für n=2/n=4

Zahl	Bitfolge
0	0.....0
1	0.....1
:	:
32767	011.....1 (2147483647 für n=4)
-32768	10.....0 (-2147483648 für n=4)
:	:
-2	11.....10
-1	11.....11

Für n-Bit Integerdarstellung gilt:

-2^{n-1} P N P 2^{n-1} -1

Zahlenrad für 3 Bits:



⇔ Überlauf!

Bemerkung: Da die Stellenzahl fest ist, kann es bei Operationen mit allen Angegebenen Zahlen zu einem Überlauf kommen, d.h. der Zulässige Zahlenbereich wird überschritten!

Beispiel:
-1 + (-4) bei 3 Bit-Darstellung:

111 + 100 = 011 *Übertrag 1*

Übertrag wird ignoriert ⇨ falsches Ergebnis!

2.2.2.4. Exzessdarstellung

- wird für die Darstellung negativer Exponenten bei Gleitkommazahlen verwendet
- Der interpretierte Wert einer Binärfolge X errechnet sich aus dem Integerwert der binären Differenz von X und einer festen Zahl q.

Beispiel: $q=2^{m-1}$ (m=Anzahl der Bits)

m=8; $q=128=2^7$

Zahl	Bitfolge X	Differenz
-128	00000000	0-128
-127	00000001	1-128
:	:	:
0	10000000	128-128
+1	10000001	129-128
:	:	:
+127	11111111	255-128

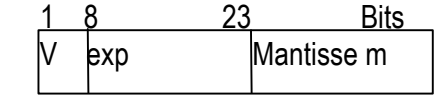
⇨ -2^{m-1} P Wert(X) P $2^{m-1} - 1$

wenn $q = 2^{m-1}$

2.2.3. Gleitkommadarstellungen im rechnerinternen Format

Aufgabe: $z = \Gamma\ m*2^{e-q}$ (b=2) ist als Bitmuster zu speichern.

1.)PPP-11 bzw VAX-11
PPP-11: 2 16Bit-Maschinenwörter
VAX-11: 1 32Bit-Maschinenwort



15 7 6 0 Bit # (PPP)
31 16 15 0 Bit # (VAX)

V = 0 ⇨ z ist positiv
V = 1 ⇨ z ist negativ

Der Binärexponent wird in Exzess –128-Form gespeichert.
z ist *immer* normalisiert, aber führende 1 wird *nicht* gespeichert!

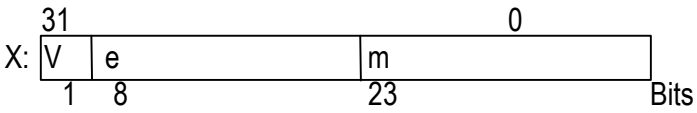
2.IEEE-Binär-Gleitkommadarstellung

IEEE = ***I***nstitute for ***E***lectrical and ***E***lectronic ***E***ngineering („I – triple – E“)

- Enstanden als Reaktion auf den Mangel eines einheitlichen Gleitkommaformates für verschiedene Rechner
  ⇨ gleiches Programm könnte verschiedene Ergebinisse auf verschiedenen Rechnern bringen!

- Verfügbar in Hardware (z.B: 8087) und Software. Software-Funktionen laufen viel langsamer.

A: der Gleitkommatyp float (n=4,4Bytes)



⇨ X=(-19*[1]*2^{e-127} normalisierte reelle Zahl ([1]=führende 1 ⇨ 1. ...)

e	M	X	Wert
0	0		Null
0	≅0	$(-1)^{⇑}0m2^{1-127}$	Nicht normalisierte reelle Zahl
255	0	$(-1)^{⇑}$ə	Unendlich
255	≅0	NaN	„Not a Number“ ⇨ ungültige Zahl!

Bemerkung:

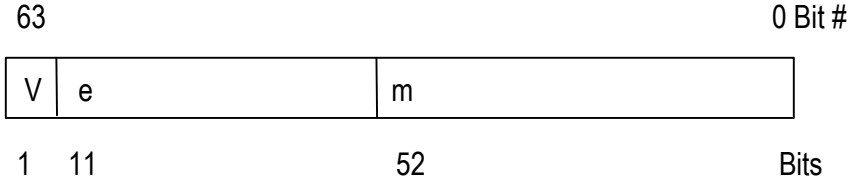
1) Exzess-127-Codierung für Exponenten
2) siehe float.h im C-Compiler
3) gültige Zahlen werden *immer* normalisiert!

größte Zahl: FLT_MAX (float.h)
kleinste Zahl: FLT_MIN (float.h)
Gemauigkeit: FLT_EPSILON (float.h) i.d.R. zwischen 6 und 7 Dezimalnachkommastellen

Bemerkung:
Zahlen ™ mit |™| < ⌈ werden als Null betrachtet, weil 1 + ™

B: Der Gleitkommatyp double im IEEE-Format

n=8



0 < e < 2047 ⇨ x = $(-1)^{⇑}$1m2^{e-1023}
e = 0,m = 0 ⇨ x = 0

$e = 0, m \cong 0 \Rightarrow x = (-1)^{\Uparrow} 0 m 2^{1-1023}$
$e = 2047, m = 0 \Rightarrow x = (-1)^{\Uparrow} э$
$e = 2047, m \cong 0 \Rightarrow x = NaN$

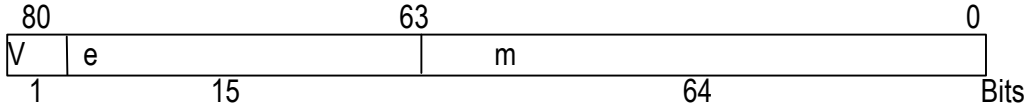
Übung:

Man berechne:

a)DBL_MAX
b)DBL_MIN
c)DBL_EPSILON

C: Der Gleitkommatyp long double im IEEE-Format

n=10 (d.h. 80 Bits)



$0 < e < 32767 \Rightarrow X = (-1)^{\Uparrow} 1 m 2^{e-16383}$ normale reelle Zahl
$e = 32767, m = 0 \Rightarrow X = (-1)^{\Uparrow} э$ „unendlich“
$e = 32767, m \cong 0 \Rightarrow X = NaN$ „Not a Number“

min_pos long double = 1.681051571556e-4932
max_pos long double = 1.189731495357e+4952
Epsilon long double = 1.0842021784855e-19

Normale Rundung in IEEE

Standardmäßig wird die Funktion round(x) angeboten (X=IEEE-Typ).
Voreinstellung: „round to nearest“ : wenn das Ergebnis (mit Extrabits
Im temporären Speicher) genau zwischen zwei güligen IEEE-Gleitkomma-
zahlen liegt, dann wird die Zahl mit letzten Bit=0 gewählt.

⇨ ca. 50% der Zahlen werden aufgerundet, ca. 50% werden abgerundet.

Auch möglich: round up, round down, chop (=truncate)
In Borland: unsigned_control187(unsigned NewCw, unsigned Mask)

2.2.4. Grundoperationen mit computer-internen Zahlen

2.2.4.1. Integer und Festkommazahlen

1. binäre Addition

Regeln: 0+0=0 ; 0+1=1+0=1 ; 1+1 = 0 *Übertrag* 1

Beispiel 1: Summe zweier Integer im Zweierkomplementsystem.

Die Summe zweier Integer N_1 und N_2 im n-Bit Zweierkomplementsystem wird durch die binäre Addtition ihrer entsprechenden Computerinternen Bitmuster gebildet.
Der Übertrag wird ignoriert!

A) 11	0000000000001011	000B
+21	+ 0000000000010101	+ 0015
32	0000000000100000	0020
B) 21	0000000000010101	0015
-11	+1111111111110101	- FFF5
10	0000000000001010	0020
C) 11	0000000000001011	0015
- 21	+ 1111111111101011	- FFEB
- 10	1111111111110110	FFF6
D) -11	1111111111110101	FFF5
-21	+1111111111101011	FFEB
-32	1111111111100000	FFE0

Übung: man bilde die gleichen Summen in Einerkomplementsystem!

Die BCD-Addition

Problem: Dezimalzahlen zwischen 0 und 9 als Dualzahlen zu codieren (Binary coded Decimal)

Lösung: Erste 10 Hexstellen belkegen (Tetraden) – die nicht benötigten Tetraden (10-15) nennt man „Pseudotetraden“, sie werden Ignoriert!

Dec	BCD-Tetrade
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
Pseudotetraden	1010 } 6 Pseudot.
	:
	1111

Die Addition wird bitweise für jede Tetrade durchgeführt.

Fall1: Die Summe zweier Tetraden verursacht einen 16er Überlauf (=Übetrag 1):

```
   9          1001
+  9        + 1001
  18          0010 Übertrag 1
            + 0110
              1000
```

Lösung: $0110_2 = 6_{10}$ addieren (16-10=6)

Fall2: Das Ergebnis liegt in den Pseudotetraden:

⇨ Korrektur-Addition von 6_{10} durchführen und Übertrag in die nächste Tetrade erzeugen!

z.B.:

```
   8        1000
 + 5      + 0101
 ----     ------
 13         1101  Pseudotetrade!
          + 0110
          ------
          1 1011
```

Übung: 4086+8075 in BCD durchführen

2. Binäre Subtraktion

Regeln: 0-0 = 0 ; 1-0=1; 0-1=1 „Borg“ 1; 1-1=0

Subtraktion im Vz + Betrag-System:

21	0000000000010101	0015
- 11	- 0000000000001011	- 000B
+10	0000000000001010	000A

3.Binäre Multiplikation

Regeln: 0*0 = 0*1 = 1*0 = 0; 1*1=1

Bemerkung: Die Multiplikation wiurd normalerweise durch eine Summe Von verschobenen Operanden realisiert.

z.B.

21*11	10101x101	15xB = E7
21	101001	
21	00000	
	10101	
231	10101	
	11100111	

Folgerung:

Die Multiplikation mit einer Potenz 2^k von 2 entspricht einer Verschiebung um K Stellen nach links.

z.B. 21x4=84

$4=2^2$

10101x100

10101
00000
00000

10010100 (2-stellige Linksverschiebung!)

(vgl. Algorithmus von Booth!)

4.Binäre Division:

Wie sich Multiplikation realisieren lässt, lässt sich Division durch wiederholte Subtraktion realisieren.

z.B.

1111011/111110100 =

1111011.00000
- 0111110.100

111100.10000
- 011111.01000

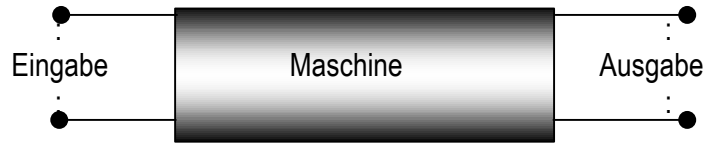
Übung: weiterrechnen für zwei weitere Nachkommastellen

2.2.4.2. Gleitkommaoperationen

siehe 2.1.5.

II. Endliche Automaten und ihre Anwendungen

Ziel: Eine Maschine mit endlich vielen Eingaben und Ausgaben zu modellieren

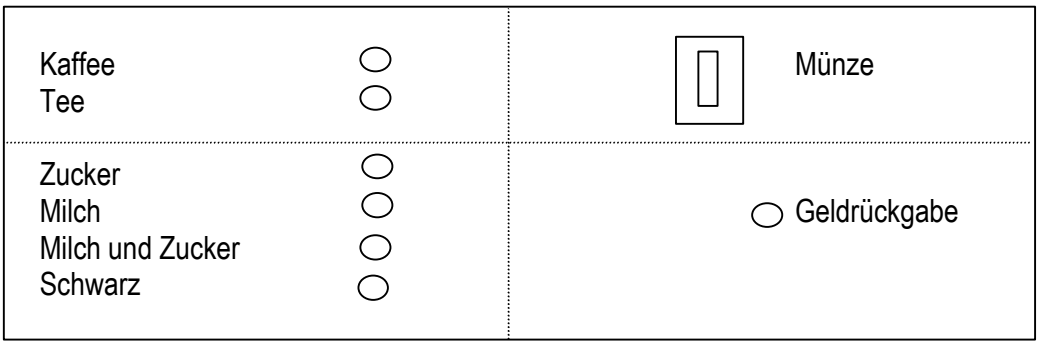


Die Maschine befindet sich in einem von endlich vielen Zuständen.
Der Systemzustand ist das Ergebnis von vorangegangenen Eingaben, die ihrerseits
Das Verhatlen der Maschine im Bezug auf zukünftige Eingaben bestimmen.

3.1. Beispiele und Definition

Jede Maschine (inklusive EDV-Anlagen), die ihr Verhalten (d.h. die Menge von Aussen bestimmter innerer Zustände des Systems) selbsttätig steuert ist ein *Automat*.

Beispiel 1: Kaffeeautomat



Zustand:neutral

Schritt 1: Student wirft 1 DM ein ⇨ Die Machine ändert ihren inneren Zustand von neutral auf ready! Ausgabe: keine

Schritt 2: Student drückt entweder den Kaffee- oder Teeknopf
⇨ die Maschine ändert ihren inneren Zustand von ready auf select! Ausgabe: keine

Schritt 3: Student betätigt einen von den Zusatzknöpfen: Zucker, Milch, Milch und Zucker, schwarz.

Ausgabe: Die Maschine gibt das gewüpnschte Getränk mit Dem gewählten Zusatz aus.

Maschine kehrt in den neutral-Zustand zurück.

Vor der endgültigen Wahl des Getränkes kann der Geldrückgabe-Knopf gedrückt werden:
⇨ Maschine kehrt dann wieder in den neutralen Zustand zurück
⇨ Ausgabe:Münze

Beispiel:

Schritt	Eingabe	Ausgabe
1	Milchknopf drücken	--
2	Geldrückgabeknopf drücken	--
3	1 DM einwerfen	--
4	Kaffeeknopf drücken	--
5	Milchknopf drücken	Kaffee mit Milch
6	1 DM einwerfen	--
7	Geldrückgabeknopf drücken	1 DM

Folgerung: Die gleiche Eingabe verursacht nicht unbedingt die gleiche Ausgabe! ⇨ Es hängt von der „Vorgeschichte“ ab!

Kaffeeautomat:

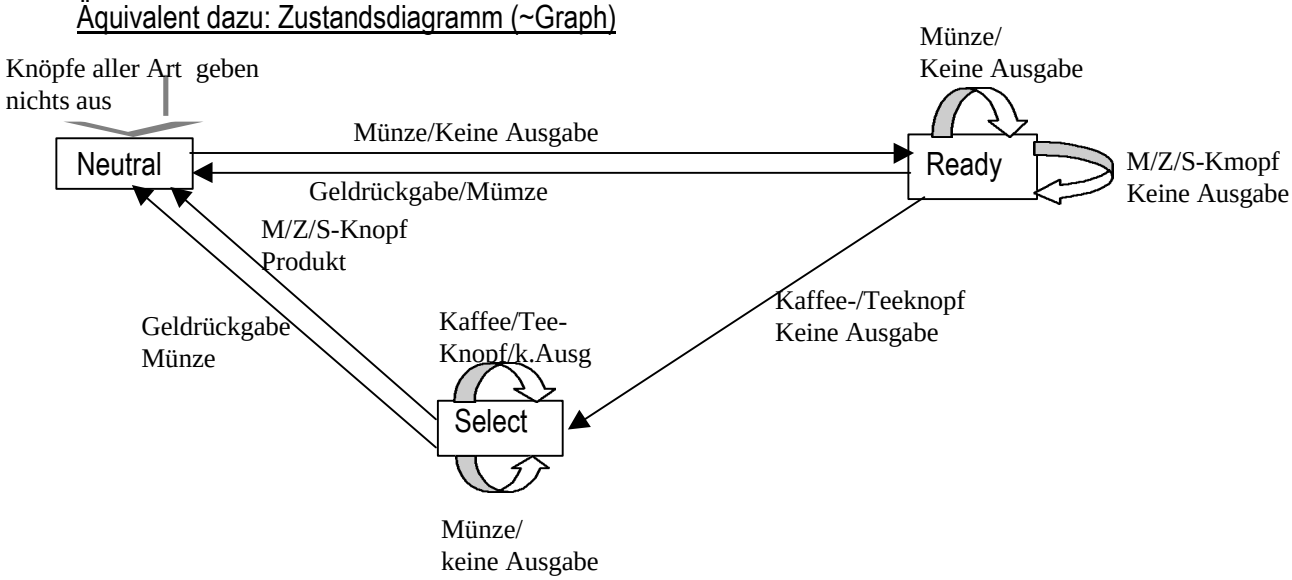
3 Zustände:

- Neutral
- Ready (Es kann gewählt werden)
- Select (Es kann ausgegeben werden)

Zustandstafel:

	Münze		Kaffee/Tee-Kn		M/Z/S-Knopf		Geldrückg.-Kn	
Akt. Zustand	Ausg.	Folgez	A	F	A	F	A	F
Neutral	--	Rdy	--	Neutral	--	Neutral	--	Neutral
Ready	--	Rdy	--	Select	--	Rdy	Münze	Neutral
Select	--	Select	--	Select	Produkt	Neutral	Münze	Neutral

Äquivalent dazu: Zustandsdiagramm (~Graph)



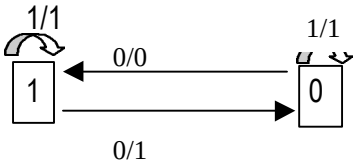
Beispiel 2: Eine Mausefalle

Zustände: Z={0;1} Eingabe={0;1}
Z=0 : Falle gespannt E=0: Maus kommt in Reichweite der Falle
Z=1 : Falle nicht gespannt E=1: Maus kommt nicht in Reichweite der Falle

Ausgabe: Y={0;1}
Y=0: Maus wird erschlagen
Y=1: Maus wird nicht erschlagen

Zustandstafel:

	0		1	
	A	F	A	F
0	0	1	1	0
1	1	0	1	1



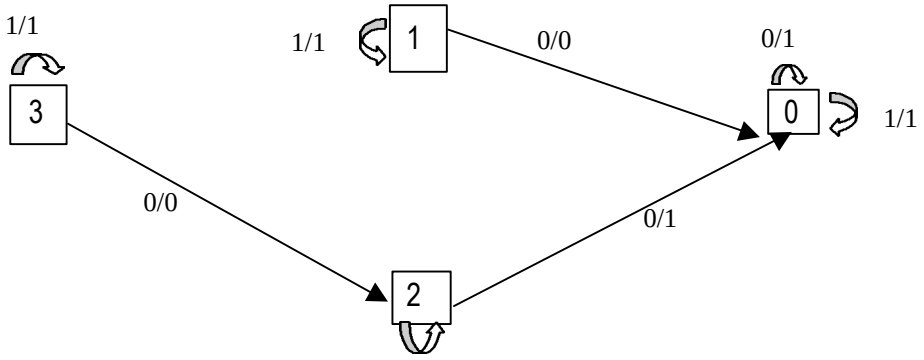
Beispiel 3: Mausefalle mit Speck

X und Y wie in Beispiel 2
Z = {0,1,2,3,}
Z=0: Speck nicht vorhanden, Falle nicht gespannt
Z=1: Speck nicht vorhanden, Falle gespannt
Z=2: Speck vorhanden, Falle nicht gespannt

Z=3: Speck vorhanden, Falle gespannt

Zustandstafel:

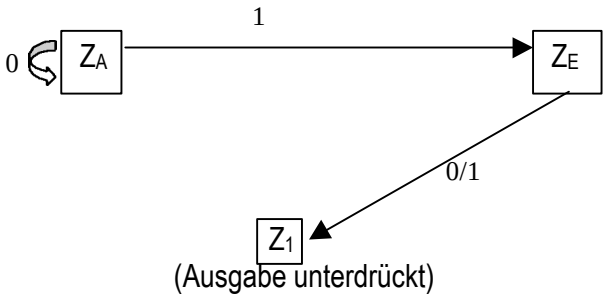
	0		1	
	A	F	A	F
0	1	0	1	0
1	0	0	1	1
2	1	0	1	2
3	0	2	1	3



Es ist zweckmäßig, einem Automaten einen Anfangszustand Z_A und einen Endzustand Z_E zuzuordnen.

⇨ Eine Eingabefolge wird vom Automaten akzeptiert, wenn sich der Automat über eine Folge von Zwischenzuständen von Z_A nach Z_E während der sequentiellen Abarbeitung der Eingabe bewegt.

Beispiel:



(Ausgabe unterdrückt)

akzeptiert Eingaben der Form
{1;01;001;...}=0*1
=Sprachsatz

Definition: Ein endlicher Automat ist durch 5 Bestandteile gegeben:

1) Eingabemenge (endliche Menge X)
2) Ausgabemenge (endliche Menge Y)
3)Zustandsmenge (endliche Menge Z)
4)Eine Ausgabefunktion g:XxZ⇨Y
5)Eine Übergangsfunktion f:XxZ⇨Z

Anfangszustand Z_A wird sets vorrausgesetzt!
Ausgabe- und Übergabefunktionen werden in einer Zustandstandstafel Beschrieben.

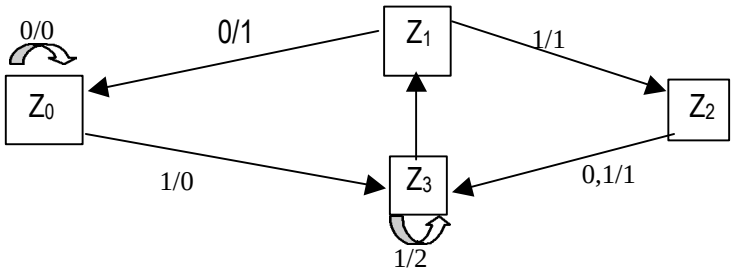
Aufgaben:

1) Gegeben ist folgende Zustandstafel, die das Verhalten eines endlichen Automaten beschreibt:

	0		1	
	A	F	A	F
Z_0	0	Z_0	0	Z_0
Z_1	1	Z_2	1	Z_1
Z_2	1	Z_2	1	Z_0

a) Stellen Sie das Zustandsdiagramm auf
b) welche Ausgabesequenz resultiert aus der Eingabesequenz 1101?

2) Gegeben ist ein Zustandsgraph für einen endlichen Automaten:



Geben Sie eine zugehörige Zustandstafel an.

3) Die Addition zweier binärer Zahlen soll über einen endlichen Automaten realisiert werden.

Eingabemenge X={(0;0),(0;1),(1;0),(1;1)}

Definition der Addition:

0+0=0
0+1=1+0=1
1+1=0 Übertrag 1

Zustände: $Z=\{Z_0,Z_1,Z_2,Z_3\}$

Z_0 : Ausgabe 0 kein Übertrag
Z_1 : Ausgabe 0 mit Übertrag
Z_2 : Ausgabe 1 kein Übertrag
Z_3 : Ausgabe 1 mit Übertrag

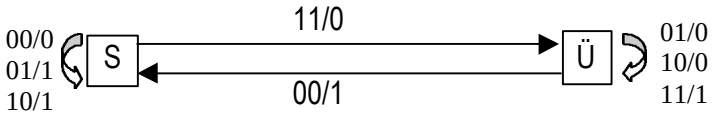
Ausgabemenge Y={0;1}

Zeit	t_0	t_1	t_2	t_3	t_4
Eingabe	11	10	01	00	--
(Folge)Zust.	Z_0	Z_1	Z_1	Z_2	--
Ausgabe	0	0	0	1	--

a) Stellen Sie den Zustandsgraph dar! Gibt es illegale Übergänge?
b) Berechnen Sie mit Hilfe von a die Summe 01110110 + 01010101

Bemerkung: Für das Innenleben eines Addierers ist lediglich interressant, ob die Summenbildung zu einem Übertrag führt oder nicht!

Reduzierter Automat: ZU={S,Ü} $S=Z_A$



4) Man gebe das Zustandsdiagramm eines endlichen Automaten an, der Paritätsprüfung („Parity Check“) modelliert. Falls die Eingabe, über die Zeit t_i betrachtet, eine gerade Anzahl von Einsen enthält, soll die Ausgabe zur Zeit t_{i+1}
gleich 1 sein. Andernfalls ist die Ausgabe 0!

5) man gebe das Zustandsdiagramm eines endlichen Automaten an, der Zeichenketten aus 0 und 1 bestehend, in denen 0 und 1 gerade oft vorkommen,überprüft (d.h. wenn 0 und 1 gerade oft vorkommen ist die Ausgabe 1
sonst 0).
Hinweis: Zustände Z_A und Z_E angeben!
Auch möglich: Z_E kann durchaus aus mehreren Zuständen bestehen!

3.2. Schaltnetze

Elektrisches Bauteil des Rechners hat 2 Zustände:

Spannung hoch (HI): 1
Spannung niedrig (LO): 0

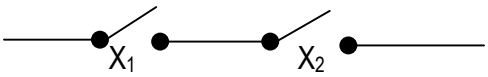
⇨ Verhalten kann durch Funktion(en) von binären Variablen (d.h. über {0;1})

modelliert werden, z.B.:

Schalter geschlossen: 1 (wahr)
Schalter offen: 0 (falsch)

Wahrheitswerte = {0;1}
Beliebige logische Verknüpfungen können durch Schaltungen ermittelt werden:

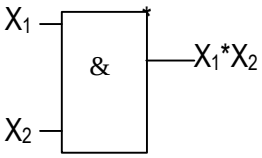
a)Reihenschaltung (UND):

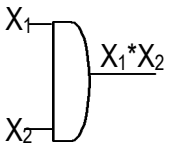


Wahrheitstabelle:

X_1	X_2	X_1*X_2
0	0	0
0	1	0
1	0	0
1	1	1

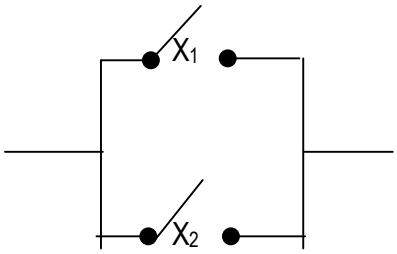
Gatterdarstellung:





* = DIN 40700 Norm

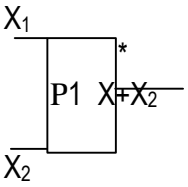
b) Parallelschaltung (OR)

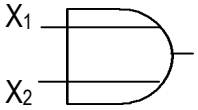


Wahrheitstabelle

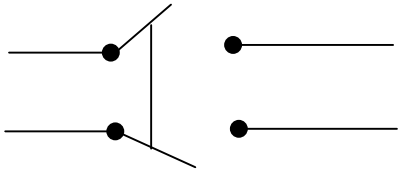
X_1	X_2	X_1+X_2
0	0	0
0	1	1
1	0	1
1	1	1

Gatterdarstellung:





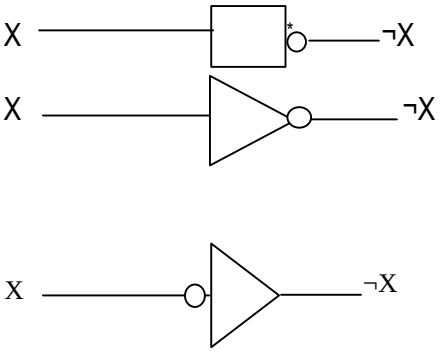
c) Negation wird durch 2 gekoppelte Schalter realisiert:



Wahrheitstabelle:

X	¬X
0	1
1	0

Gatterdarstellung:



3.2.1. Boolesche Algebra

A.Binäre Funktionen einer Variable:

X	f_0 NIE	f_1 Identität	f_2 Negation	f_3 IMMER
0	0	0	1	1
1	0	1	0	1

B.Binäre Funktionen zweier Vaiablen:

X_0	X_1	NIE	UND	--	X_0	--	X_1	XOR	OR	NOR	EQU	$\neg X_1$	i	$\neg X_0$	-	NAND	IMMER
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

I = Implication

Axiome der booleschen Algebra ⇨ Kopie

Vorrangsregel:

- Zuerst Negation (NICHT)
- Dann AND
- Dann + (OR)
- Dann Implication ⇨ ⇔ ⇦ usw.

Mehrere gleichrangige Opertionen werden von links nach rechts ausgewerte

3.2.2. Darstellungen logischer Funktionen:

Ziel: Vereinfachung von Schaltfunktionen

Aufgabe: Man zeige, dass der boolesche Ausdruck

$$X_1X_2X_3 + X_1\neg X_2X_3 + X_1\neg X_2\neg X_3 + \neg X_1\neg X_2X_3 + \neg X_1\neg X_2\neg X_3$$

auf den Ausdruck $X_1X_3+\neg X_2$ reduziert werden kann.

Lösung:

$X_1X_2X_3 + X_1\neg X_2X_3 + X_1\neg X_2\neg X_3 + \neg X_1\neg X_2X_3 + \neg X_1\neg X_2\neg X_3 =$

$X_1X_2X_3 + X_1\neg X_2X_3 + X_1\neg X_2X_3 + \neg X_1\neg X_2\neg X_3 + \neg X_1\neg X_2X_3 + \neg X_1\neg X_2\neg X_3 =$ (ID)

$X_1X_2X_3 + X_1X_3\neg X_2 + X_1\neg X_2X_3 + X_1\neg X_2\neg X_3 + \neg X_1\neg X_2X_3 + \neg X_1\neg X_2\neg X_3 =$(KOM)

$X_1X_3(X_2 + \neg X_2) + X_1\neg X_2(X_3 + \neg X_3) + \neg X_1\neg X_2(X_3 + \neg X_3) =$(DIST)

$X_1X_31 + X_1\neg X_21 + \neg X_1\neg X_21 =$ (a+¬a=1)

$X_1X_3 + X_1\neg X_2 + \neg X_1\neg X_2 =$ (a*1=a)

$X_1X_3 + \neg X_2(X_1 + \neg X_1) = (DIST)$

$X_1X_3 + \neg X_2 = (a+\neg a=1 \text{ und } a*1=a)$

q.e.d

Übung: $f(X_0,X_1,X_2) = \neg(\neg X_0(\neg X_1 + X_2)) = X_0+X_1\neg X_2$

Jede Schaltfunktion kann auf die Grundverknüpfungen Konjunktion (UND), Disjunktion (OR) du Negation zurückgeführt werden.

A. Disjunktive Normalform (DNF)

Die DNF setzt sich aus einer oder mehreren disjumktiv verknüpften Konjunktionen („Minterme“) zusammen.

Beispiel:

Gegeben ist:

X_1	X_2	X_3	$F(X_1,X_2,X_3)$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

DNF:
$F(X_1,X_2,X_3)=\neg X_1\neg X_2\neg X_3 + \neg X_1\neg X_2X_3 + X_1\neg X_2X_3$

Übung: Stellen Sie die DNF der in der vorhergehenden Übung gegebenen Funktion auf! (Hinweis: $X_0 = X_0*1*1=X_0(X_1+ \neg X_1)(X_2+\neg X_2)$)

Jede binäre Funktion von n Variablen lässt sich eindeutig in DNF darstellen, indem man diejenigen Minterme miteinander disjunktiv verknüpft, für die die Funktion den Wert 1 hat.

Definition von Mintermen und Maxtermen (n=3)

$X_0X_1X_2$	Minterm	Maxterm
000	$m_0 = \neg X_0\neg X_1\neg X_2$	$M_0=X_0+X_1+X_2$
001	$m_1 = \neg X_0\neg X_1X_2$	$M_1=X_0+X_1+\neg X_2$
010	$m_2 = \neg X_0X_1\neg X_2$	$M_2=X_0+\neg X_1+X_2$
011	$m_3 = \neg X_0X_1X_2$	$M_3=X_0+\neg X_1+\neg X_2$
100	$m_4 = X_0\neg X_1\neg X_2$	$M_4=\neg X_0+X_1+X_2$

101	$m_5 = X_0\neg X_1X_2$	$M_5=\neg X_0+X_1+\neg X_2$
110	$m_6 = X_0X_1\neg X_2$	$M_6=\neg X_0+\neg X_1+X_2$
111	$m_7 = X_0X_1X_2$	$M_7=\neg X_0+\neg X_1+\neg X_2$

Beispiel: $f(X_1,X_2,X_3) = \neg X_1\neg X_2\neg X_3 + \neg X_1\neg X_2X_3 + X_1\neg X_2X_3 = m_0 + m_1 + m_5$

B. Konjunktive Normalform (KNF)

Die KNF setzt aus einer oder mehreren konjunktiv verkniüpften Disjunktion („Maxterm“) zusammen.

Beispiel: f wie oben

$f= (X_1+\neg X_2+X_3)*(X_1+\neg X_2+\neg X_3)*(\neg X_1 + X_2+X_3)*(\neg X_1+\neg X_2+X_3)*(\neg X_1+\neg X_2+\neg X_3) =$
$= M_2*M_3*M_4*M_6*M_7$

Übung: KNF für $f(X_0,X_1,X_2)=\neg(\neg X_0(\neg X_1+X_2))$
a) Direkt
b) aus DNF

Zusätzliche Aufgaben:

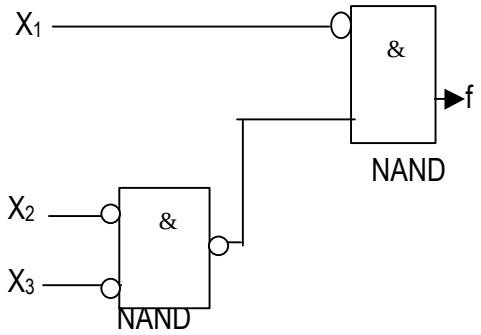
1) a)Man bestimme die DNF der gegebenen Schaltfunktion f
b)Man gebe den Schaltplan für den booleschen Ausdruck f in DNF an (logisches Netzwerk)
c)Man vereinfache den in DNF vorliegenden booleschen Ausdruck $X_0\neg X_1\neg X_2+ X_0X_1\neg X_2$
d) Man zeichne das Schaltnetz für c)

$X_0X_1X_2$	f
000	0
001	0
010	0
011	0
100	1
101	0
110	1
111	0

2) Man zeige über die Wahrheitstabelle, dass die beiden folgenden Ausdrücke äquivalent sind:

$(X_1+X_2)(\neg X_1+X_3)(X_2+X_3)$
$X_1X_3+\neg X_1X_2$

3) Man ersetze fogendes Schaltnetz durch ein äquivalentes Schaltnetz, das
1 UND-Gatter, 1 OR-Gatter und 1 Inverter enthält.



Minimierung von binären Funktionen

Ziel: Schaltfunktion mit geringstem Aufwand

A. Direkte Anwendung der Gesetze der booleschen Algebra

1. Die binäre Funktion ist in eine Form zu bringen, die der DNF ähnlich ist.
2. Die Anwendung einfacher Rechenregeln z.B. $X_1X_2+X_1\neg X_2 = X_1$ ist zu überprüfen.
3. Die Anwendung von Kürzungsregeln z,B. X+0=0; X*1=X ist zu prüfen.
4. Ausklammern⇨(oft) geringerer Aufbau.

B. Das Karnaugh-Verfahren (KV)

Das KV-Diagramm ist eine Ansammlung von Quadratischen Feldern, die je einem Minterm entsprechen.

KV-Diagramme:

1)1 Variable

	X
¬X	X

2)2 Variablen

		X_0
	$\neg X_0\neg X_1$ m_0	$X_0\neg X_1$ m_2
X_1	$\neg X_0X_1$ m_1	X_0X_1 m_3

3)3 Variablen

			X_0	
	$\neg X_0\neg X_1\neg X_2$ m_0	$\neg X_0 X_1\neg X_2$ m_2	$X_0X_1\neg X_2$ m_6	$X_0\neg X_1\neg X_2$ m_4
X_2	$\neg X_0\neg X_1 X_2$ m_1	$\neg X_0 X_1 X_2$ m_3	$X_0X_1X_2$ m_7	$X_0\neg X_1 X_2$ m_5
		X_1		

4)4 Variablen

			X_0	
	$\neg X_0\neg X_1\neg X_2\neg X_3$	$\neg X_0 X_1\neg X_2\neg X_3$	$X_0X_1\neg X_2\neg X_3$	$X_0\neg X_1\neg X_2\neg X_3$
	$\neg X_0\neg X_1\neg X_2 X_3$	$\neg X_0 X_1\neg X_2 X_3$	$X_0X_1\neg X_2 X_3$	$X_0\neg X_1\neg X_2 X_3$
X_0	$\neg X_0\neg X_1 X_2 X_3$	$\neg X_0 X_1 X_2 X_3$	$X_0X_1X_2X_3$	$X_0\neg X_1 X_2 X_3$
	$\neg X_0\neg X_1 X_2\neg X_3$	$\neg X_0 X_1 X_2\neg X_3$	$X_0X_1X_2\neg X_3$	$X_0\neg X_1 X_2\neg X_3$
		X_1		

X_3

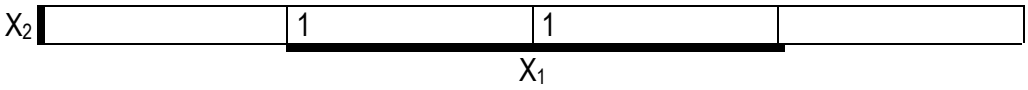
Jeder Minterm wird durch eine „1“ im KV-Diagramm gekennzeichnet. Benachbarte Felder können beispielsweise nach der Regel $X_1X_2 + \neg X_1X_2 = X_2$ vereinfacht werden.
Dies geschieht durch Zusammenfassen benachbarter Felder zu sogenannten Elementarblöcken.

<u>Beispiel:</u>

$X_0X_1X_2$	f	
000	1	$m_0=\neg X_0\neg X_1\neg X_2$
001	0	
010	0	
011	1	$m_3=\neg X_0X_1X_2$
100	1	$m_4=X_0\neg X_1\neg X_2$
101	0	
110	0	
111	1	$m_7=X_0X_1X_2$

KV-Diagramm:

		X_0	
1			1



DNF = $m_0 + m_3 + m_4 + m_7$

$m_0 + m_4 = \neg X_0\neg X_1\neg X_3 + X_0\neg X_1\neg X_2 = \neg X_1\neg X_2$

$m_3 + m_7 = \neg X_0X_1X_2 + X_0X_1X_2 = X_1X_2$

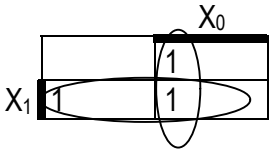
⇨ $f = X_1X_2 + \neg X_1\neg X_2$

Bemerkung zu KV-Diagrammen

- Die Variable in der sich die Felder unterscheiden entfällt (komplimentär!)
- Die benachbarten Minterme reduzieren sich auf einen Term mit den verbleibenden gemeinsamen Variablen
- Alle gekürzten und ungekürzten Minterme des KV-Diagramms werden disjunktiv miteinander verknüpft
- Bei der Vereinfachung kommt es darauf an möglichst viele Minterme zu einem Elemtarblock zusammenzufassen!

Beispiel:

$f = X_0\neg X_1 + X_0X_1 + \neg X_0X_1$

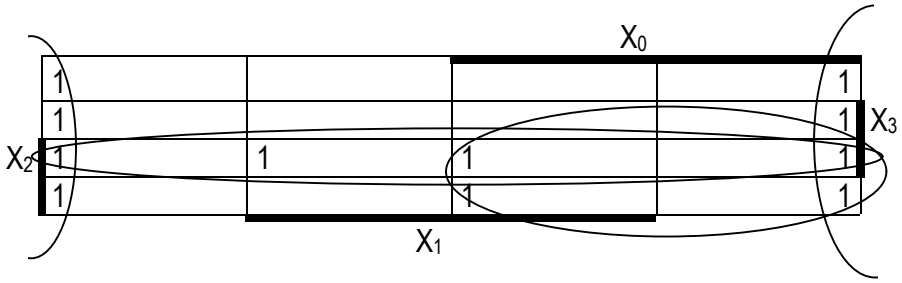


⇨ $f = X_0 + X_1$

So wie sich benachbarte Minterme zu einem „Zweierblock“ zusammenfassen lassen, so kann man auch benachbarte „Zweierblöcke“ zu einem „Viererblock“ zusammenfassen usw.

Beispiel:

$f = X_0\neg X_1 + \neg X_0X_2X_3 + X_0X_1X_2\neg X_3 + X_0X_1X_2X_3$



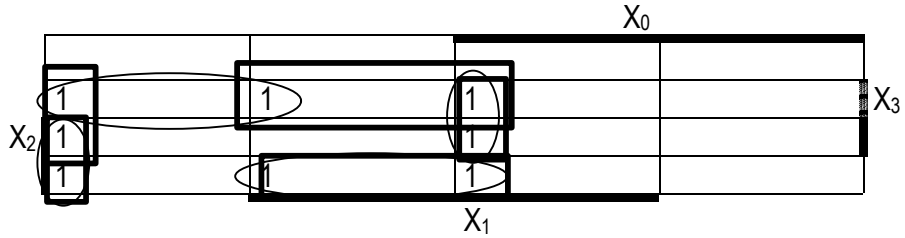
⇨ $f = \neg X_1 + X_0X_2 + X_2X_3$

⇨ Die Elementarblöcke dürfen nur 2^n – Felder enthalten!

<u>Problem des Minimierungsverfahrens</u>

Mehrere Möglichkeiten der Blockbildung können existieren!

<u>Beispiel:</u>



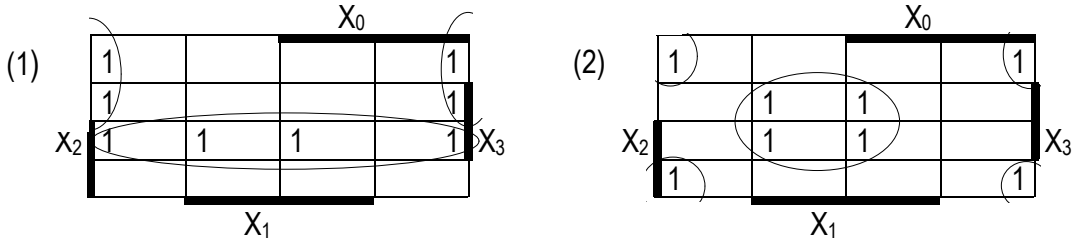
(1) $f_{Rechtecke} = \neg X_0\neg X_1X_3 + \neg X_0\neg X_1X_2 + X_1X_2\neg X_3 + X_0X_1X_3 + X_1X_3\neg X_2$

(2) $f_{Kreise} = \neg X_0\neg X_2X_3 + \neg X\neg X_1X_2 + X_1X_2\neg X_3 + X_0X_1X_3$

(1) erfüllt <u>nicht</u> die Forderung nach der minimalen Anzahl von Blöcken!

(2) zeigt optimale Blockbildung

<u>Beispiel:</u> Zusammenfassung in Viererblöcken



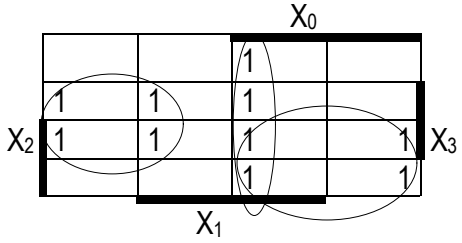
$f_{(1)} = \neg X_1\neg X_2 + X_2X_3$
$f_{(2)} = \neg X_1\neg X_3 + X_1X_3$

Das KV-Diagramm läßt sich auch für die Ermittlung er minimalen DNF nutzen (auch KNF)

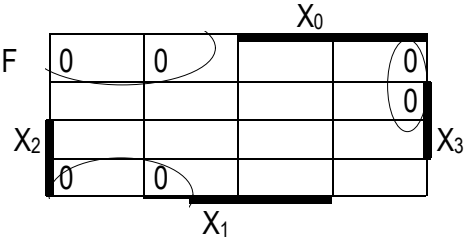
<u>Beispiel:</u>

$f = X_0X_1\neg X_3 + X_1X_2X_3 + \neg X_1X_2X_3 + X_0\neg X_1X_2\neg X_3 + \neg X_0\neg X_1\neg X_2\neg X_3 + X_1\neg X_2X_3$

DNF



KNF



$f = \neg X_0 X_3 + X_0 X_1 + X_0 X_2$

$\neg f = \neg X_0 \neg X_3 + X_0 \neg X_1 \neg X_2$

$f = \neg\neg f = (X_0 + X_3)(\neg X_0 + X_1 + X_2)$

Zur Festlegung der KNF wird zuerst die minimale DNF für jede Funktion ¬f durch Zusammenfassen aller Nullen im KV-Diagramm bestimmt. Danach wird durch Anwendung des De-Morganschen Gesetzes die KNF für f = ¬¬f betimmt.

Redundanzen

Redundanzen beeinflussen nicht das Funktionsergebnis! Die Funktionswerte für diese Kombination(en) sind daher frei wählbar. Redundanzen werden im KV-Diagramm mit einem „X“ gekennzeichnet.
Beispiel:

4 Verbraucher soolen gegenseitig so verschaltet werden, dass nur maximal 2 Anschlüsse gleichzeitig betrieben werden können, d.h. beim Anschluss eines zweiten Verbrauchers müssen die restlichen beiden Anschlüsse daktiviert werden.

X_1 = 1 : Verbraucher 1 ist angeschlossen
X_2 = 1 : Verbraucher 2 -“-
X_3 = 1 : Verbraucher 3 -“-
X_4 = 1 : Verbraucher 4 -“-

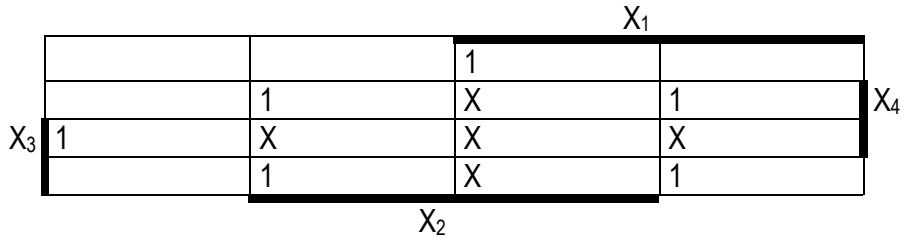
Ziel: $f(X_1,X_2,X_3,X_4)$=?

Die Funktion f wird erfüllt, wenn kein weiterer Verbraucher angeschlossen werden darf.

X_1	X_2	X_3	X_4	Minterm
0	0	0	0	
0	0	0	1	
0	0	1	0	$\neg X_1 \neg X_2 X_3 \neg X_4$
0	0	1	1	
0	1	0	0	
0	1	0	1	$\neg X_1 X_2 \neg X_3 X_4$
0	1	1	0	$\neg X_1 X_2 X_3 \neg X_4$
0	1	1	1	$\neg X_1 X_2 X_3 X_4$
1	0	0	0	
1	0	0	1	$X_1 \neg X_2 \neg X_3 X_4$
1	0	1	0	$X_1 \neg X_2 X_3 \neg X_4$
1	0	1	1	$X_1 \neg X_2 X_3 X_4$
1	1	0	0	$X_1 X_2 \neg X_3 \neg X_4$

1	1	0	1	$X_1X_2\neg X_3X_4$
1	1	1	0	$X_1X_2X_3\neg X_4$
1	1	1	1	$X_1X_2X_3X_4$

In dieser Funktion f sind einige Minterme, die praktisch nicht vorkommen können: d.h. ein gleichzeitiger Anschluss von 3 oder 4 Verbrauchern ist hier nicht möglich.



$f = X_1X_2 + X_1X_3 + X_2X_4 + X_3X_4$

Rechnerisches Verfahren nach Quine/McClusky

Ausgangspunkt:DNF

Idee: Die in der DNF auftretenden Konjunktionen werden miteinander kombiniert.
Besteht dann eine Konjunktion (Produkt) aus n Variablen, von denen m „bejaht“ (ohne „Quer“) sind, dann ist nur eine Kombination mit einer Konjunktion sinvoll, in der m+1 bzw m-1 Variablen „bejaht“ sind.

Vollkonjunktionen werden daher nach Anzahl der „bejahten“ Variablen in Gruppen zusammengefaßt. Lediglich aufeinanderfolgende Gruppen müssen miteinander verglichen werden.

Beispiel:

$f(x_0,X_1,X_2,X_3) = \neg X_0X_1\neg X_2\neg X_3 + X_0X_1\neg X_2\neg X_3 + \neg X_0\neg X_1X_2\neg X_3 + X_0\neg X_1X_2\neg X_3 + \neg X_0X_1\neg X_2X_3 + X_0X_1\neg X_2X_3 + \neg X_0\neg X_1X_2X_3 + X_0\neg X_1X_2X_3 + X_0X_1X_2X_3$

⇨ Vereinfachungstabelle aufstellen

Gruppe	DNF	1.Vereinfachung	2.Vereinfachung
1	$\neg X_0X_1\neg X_2\neg X_3$	$\neg X_0X_1\neg X_2$	$X_1\neg X_2$ P_3
	$\neg X_0\neg X_1X_2\neg X_3$	$X_1\neg X_2X_3$	$\neg X_1X_2$ P_4
2	$X_0\neg X_1X_2\neg X_3$	$\neg X_1X_2\neg X_3$	
	$\neg X_0X_1\neg X_2X_3$	$\neg X_0X_1X_2$	
	$X_0X_1\neg X_2\neg X_3$	$X_0\neg X_1X_2$	
	$\neg X_0\neg X_1X_2X_3$	$X_1\neg X_2X_3$	
3	$X_0X_1\neg X_2X_3$	$X_0X_1\neg X_3$	
	$X_0\neg X_1X_2X_3$	$\neg X_1X_2X_3$	
4	$X_0X_1X_2X_3$	$X_0X_1X_r$ P_1	
		$X_0X_2X_3$ P_2	

- Nach Anzahl der „bejahten“ Variablen geordnet ergibt sich der 1.Bereich in der Vereinfachungstabelle (Spalte DNF).

- In diesem Bereich wird jede Gruppe mit jeder Zeile der darauffolgenden Gruppe verglichen.
- Läßt sich aus zwei Konjunktionen eine Variable eliminieren, dann werden diese Konjunktionen abgehakt. Das Vereinfachungsergebnis wird in die Spalte „1.Vereinfachung“ eingetragen. Die Ergebnisse sind automatisch nach Anzahl der „bejahten“ Variablen geordnet.
- Nicht abgehakte Terme werden als „Primterm“ („Primimplicanten“) bezeichnet.

usw bis keine weiteren Vreinfachungen mehr möglich sind.

⇨ Dann Primterm-Minterm-Tabelle aufstellen

	Primterme			
	P_1	P_2	P_3	P_4
Minterme	$X_0X_1X_3$	$X_0X_2X_3$	$X_1\neg X_2$	$\neg X_1X_2$
$\neg X_0X_1\neg X_2\neg X_3$			✓	
$\neg X_0\neg X_1X_2\neg X_3$				✓
$X_0X_1\neg X_2\neg X_3$			✓	
$X_0\neg X_1X_2\neg X_3$				✓
$\neg X_0X_1\neg X_2X_3$			✓	
$X_0X_1\neg X_2X_3$	✓		(✓)	
$X_0\neg X_1X_2X_3$		✓		(✓)
$X_0X_1X_2X_3$	✓	(✓)		

- Die Minimalfunktion erhält man duch disjunktive Verknüpfung derjenigen Primterme der die alle Minterme erfasst werden.

  Hierfür gibt es häufig mehrere Möglichkeiten!

  $P_1 + P_3 + P_4 = X_0X_1X_3 + X_1\neg X_2 + \neg X_1X_2$
  $P_2 + P_3 + P_4 = X_0X_2X_3 + X_1\neg X_2 + \neg X_1X_2$

Aufgaben:

1. Gegeben: Wahrheitstabelle

$X_0X_1X_2X_3$	f
0000	1
0001	1
0010	1
0011	1
0100	0
0101	1
0110	1
0111	0
1000	1
1001	0
1010	0

a) DNF aufstellen
b) Vereinfachungstabelle aufstellen
c) b) Vverarbeiten zur Ermittlung der Primterme
d) Vereinfachung mit Primterm-Minterm-Tabelle

1011	1
1100	0
1101	0
1110	0
1111	0

3.2.3. Schaltungen

1. Codier- und Decodierschaltungen

a)Grundlagen

- binäre Codierung von Zahlen

  2 Möglichkeiten:

  1) Darstellung von ganzen Dezimalzahlen durch eine Kombination von binären Codezeichen

     Beispiel:

     $42_{10} = (1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0)_{10}$

  2) Codierung jeder einzelnen Ziffer der Dezimalzahl für sich allein (s. Kopie!)

- Codierung der Buchstaben und Sonderzeichen

  Systematik wie bei Zahlen gibt es nicht!
  Forderung: binäre Codierung!
  ASCII (**A**merican **S**tandard **C**ode for **I**nterchange of **I**nformation)
  - Benötigt 7 Bit/Zeichen
  - 8. Bit für Paritätscheck

  ⇨ 1 Byte/Zeichen

  2 Möglichkeiten bei Paritäts-Check:

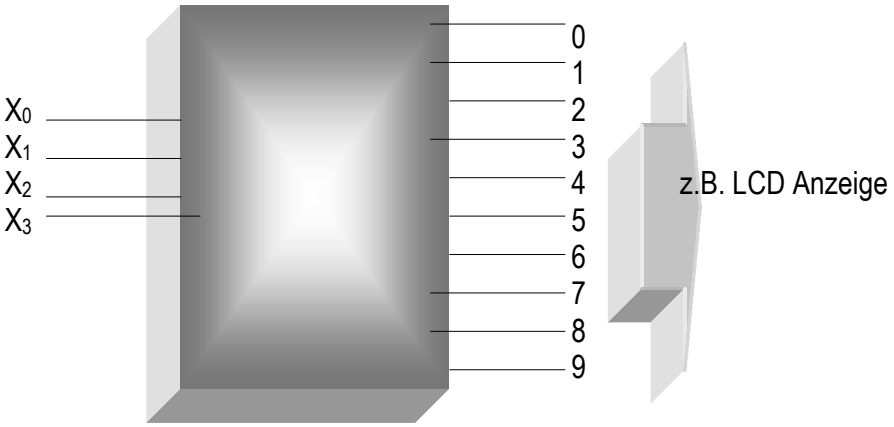
  1. *even Parity : Ergänzt die Anzahl der Einsen auf gerade Zahl*
     z.B. B = 01000010 (even)
     C = 1100011 (even

2. *odd Parity : Ergänzt die Anzahl der Einsen auf ungerade Zahl*
   z.B. B=11000010 (odd)
   C=01000011 (odd)

Beispiel: RAY in ASCII mit even Parity

11010010 01000001 01011001
R A Y

b)Decodierschaltungen: 8-4-2-1 BCD Decoder



Zuordnungstabelle:

X_0	X_1	X_2	X_3	Ausgabe	Minterm	KV-Ergebnis
0	0	0	0	0	$\neg X_0\neg X_1\neg X_2\neg X_3$	$\neg X_0\neg X_1\neg X_2\neg X_3$
0	0	0	1	1	$\neg X_0\neg X_1\neg X_2 X_3$	$\neg X_0\neg X_1\neg X_2 X_3$
0	0	1	0	2	$\neg X_0\neg X_1 X_2\neg X_3$	$\neg X_1 X_2\neg X_3$
0	0	1	1	3	$\neg X_0\neg X_1 X_2 X_3$	$\neg X_1 X_2 X_3$
0	1	0	0	4	$\neg X_0 X_1\neg X_2\neg X_3$	$X_1\neg X_2\neg X_3$
0	1	0	1	5	$\neg X_0 X_1\neg X_2 X_3$	$X_1\neg X_2 X_3$
0	1	1	0	6	$\neg X_0 X_1 X_2\neg X_3$	$X_1 X_2\neg X_3$
0	1	1	1	7	$\neg X_0 X_1 X_2 X_3$	$X_1 X_2 X_3$
1	0	0	0	8	$X_0\neg X_1\neg X_2\neg X_3$	$X_0\neg X_3$
1	0	0	1	9	$X_0\neg X_1\neg X_2 X_3$	$X_0 X_3$

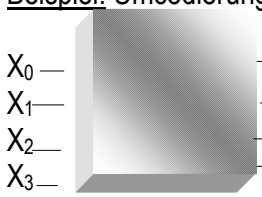
Die restlichen Minterme sind Redundanzen!

KV-Diagramm:

			X_0	
	0	4	X	8
	1	5	X	9
X_2	3	7	X	X
	2	6	X	X
		X_1		X_3

c) Umcodierschaltungen (Umwandlung eines Zahlencodes in einen anderen)

Beispiel: Umcodierung Exzess-3-Code in 8-4-2-1 BCD-Code



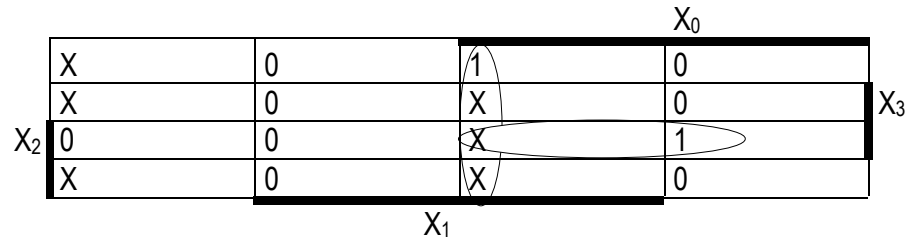
Zuordnungstabelle

X_0	X_1	X_2	X_3	Dec	Y_0	Y_1	Y_2	Y_3
0	0	1	1	0	0	0	0	0
0	1	0	0	1	0	0	0	1
0	1	0	1	2	0	0	1	0
0	1	1	0	3	0	0	1	1
0	1	1	1	4	0	1	0	0
1	0	0	0	5	0	1	0	1
1	0	0	1	6	0	1	1	0
1	0	1	0	7	0	1	1	1
1	0	1	1	8	1	0	0	0
1	1	0	0	9	1	0	0	1

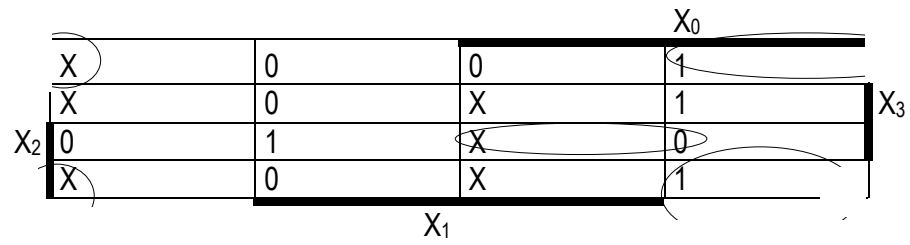
Die restlichen Minterme sind Redundanzen!

Für jedes Bit des Ausgangscodes wird eine Schaltfunktion definiert!

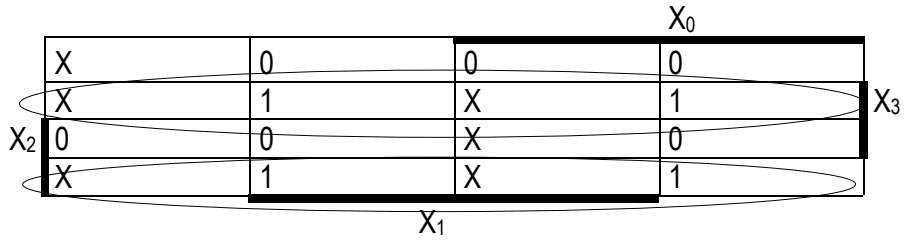
Y_0



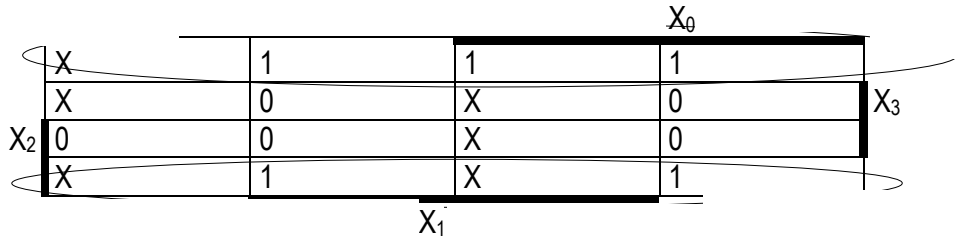
Y_1



Y_2



Y_3



$Y_0 = X_0X_1 + X_2X_3X_0$

$Y_1 = \neg X_1\neg X_2 + \neg X_1\neg X_3 + X_1X_2X_3$

$Y_2 = X_2\neg X_3 + X_3\neg X_2$

$Y_3 = X_3$

Rechnerisches Verfahren nach Quine/McClusky

Ausgangspunkt:DNF

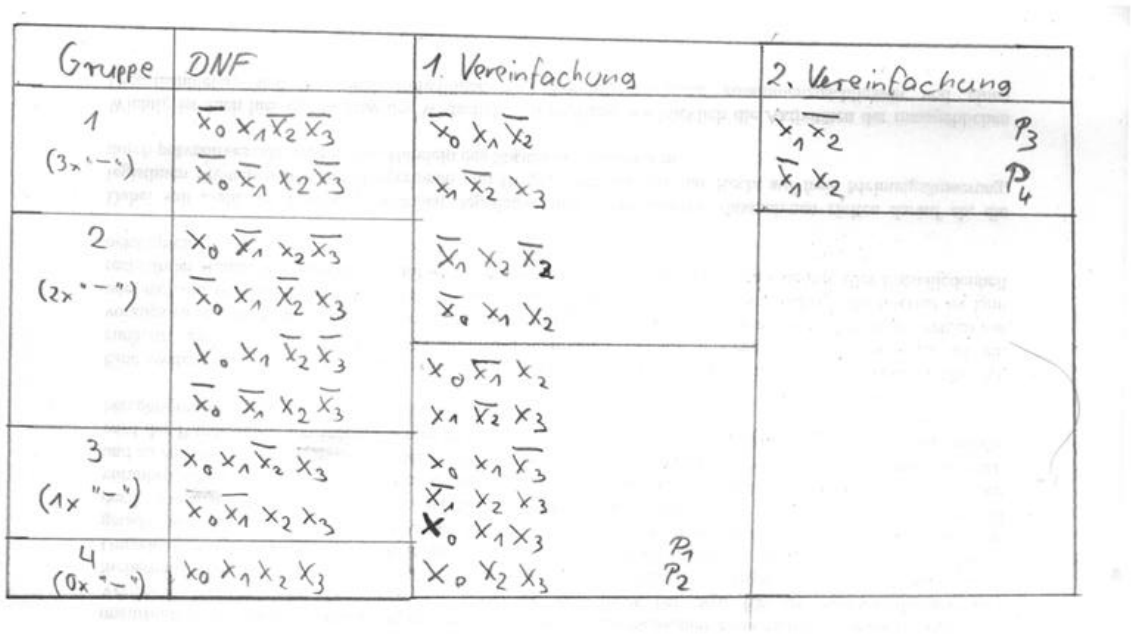
Idee: Die in der DNF auftretenden Konjunktionen werden miteinander kombiniert.
Besteht dann eine Konjunktion (Produkt) aus n Variablen, vion denen m „bejaht“ sind, dann ist nur eine Kombination mit einer Konjunktion sinnvoll, in der m+1 bzw m-1 Variable bejaht sind.

Vollkunjunktion werden daher nach Anzahl der bejahten Variablen in Gruppen zusammengefasst. Lediglich aufeinanderfolgende Gruppen müssen miteinander verglichen werden.

Beispiel:

$f(X_0,X_1,X_2,X_3) = \bar{X}_0 X_1 \bar{X}_2 \bar{X}_3 + X_0 X_1 \bar{X}_2 \bar{X}_3 + \bar{X}_0 \bar{X}_1 X_2 \bar{X}_3 + X_0 \bar{X}_1 X_2 \bar{X}_3 + \bar{X}_0 X_1 \bar{X}_2 X_3 + X_0 X_1 \bar{X}_2 X_3 + \bar{X}_0 \bar{X}_1 X_2 X_3 + X_0 \bar{X}_1 X_2 X_3 + X_0 X_1 X_2 X_3$

⇨ Vereinfachungstabelle aufstellen

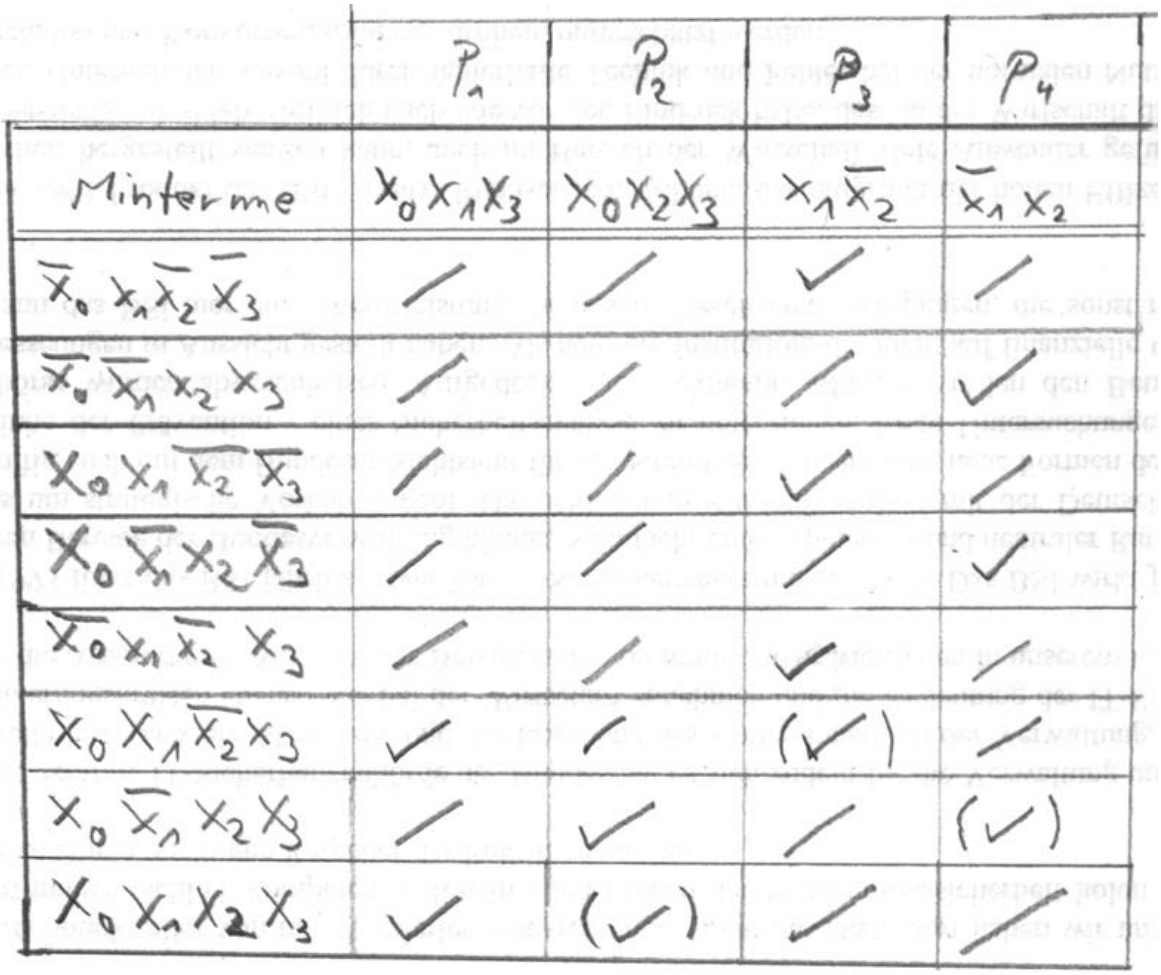


- Nach Anzahl der bejahten Variablen geordnet ergibt sich der 1. Bereich in der Vereinfachungstabelle (Spalte DNF)

- In diesem Bereich wird jede Zeile einer Gruppe mit jeder Zeile der darauffolgenden Gruppe verglichen.
- Lässt sich aus 2 Konjunktionen eine Variable eliminieren, dann werden diese Konjumktionen abgehakt. Das Vereinfachungsergebins wird in die Spalte „1.Vereinfachung“ eingetragen. Die Ergebnsse sind automatisch nach Anzahl der bejahten Variablen geordnet.
- Nich abgehakte Terme werden als „Primterm“ (P) („Primimplicanten“) bezeichnet.

- Usw. bis keine weiteren Vereinfachungen mehr möglich sind.

⇨ Dann Primterm- Minterm-Tabelle aufstellen.



Minterme	P_1 $X_0 X_1 X_3$	P_2 $X_0 X_2 X_3$	P_3 $X_1 \overline{X_2}$	P_4 $\overline{X_1} X_2$
$\overline{X_0} X_1 \overline{X_2} \overline{X_3}$	✓	✓	✓	✓
$\overline{X_0} \overline{X_1} X_2 X_3$	✓	✓	✓	✓
$X_0 X_1 \overline{X_2} \overline{X_3}$	✓	✓	✓	✓
$X_0 \overline{X_1} X_2 \overline{X_3}$	✓	✓	✓	✓
$\overline{X_0} X_1 \overline{X_2} X_3$	✓	✓	✓	✓
$X_0 X_1 \overline{X_2} X_3$	✓	✓	(✓)	✓
$X_0 \overline{X_1} X_2 X_3$	✓	✓	✓	(✓)
$X_0 X_1 X_2 X_3$	✓	(✓)	✓	✓

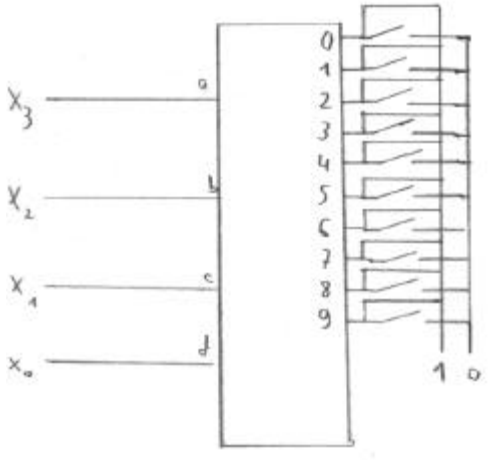
- Die Minimalfunktion erhält man durch disjunktive Verknüpfung derjenigen Primterme durch die <u>alle</u> Minterme erfasst werden.

  Hierfür gibt es häufig mehrere Möglichkeiten!

  $P_1 + P_3 + P_4 = X_0 X_1 X_3 + X_1 \overline{X_2} + \overline{X_1} X_2$
  $P_2 + P_3 + P_4 = X_0 X_2 X_3 + X_1 \overline{X_2} + \overline{X_1} X_2$

<u>d)Codierschaltungen:</u>

Beispiel: Umwandlung eines 1 aus 10-Codes in einen dekadischen (d.h. Basis 10) Code.



Zuordnungstabelle:

	d	c	b	a	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	1	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	1	0	0	0	0	0	0	0
3	0	0	1	1	0	0	0	1	0	0	0	0	0	0
4	0	1	0	0	0	0	0	0	1	0	0	0	0	0
5	0	1	0	1	0	0	0	0	0	1	0	0	0	0
6	0	1	1	0	0	0	0	0	0	0	1	0	0	0
7	0	1	1	1	0	0	0	0	0	0	0	1	0	0
8	1	0	0	0	0	0	0	0	0	0	0	0	1	0
9	1	0	0	1	0	0	0	0	0	0	0	0	0	1

A = 1+3+5+7+9 C = 4+5+6+7
B = 2+3+6+7 D = 8+9

A = □(□1 □3 □5 □7 □9) C = □(□4 □5 □6 □7)
B = □(□2 □3 □6 □7) D = □(□8 □9)

e) fehlererkennende und fehlerkorrigierende Codes

Diese Codes sind durch Anhängen zusätzlicher Prüfbits an die Informationszeichen bestimmt.
Die Codierungsvorschrift ordnet einem gegebenen Zeichensatz nicht nur Informationszeichen, sondern auch Prüfzeichen zu.

1.einfache Paritätskontrolle

Im Rechner werden binär verschlüsselte Codes häufig transportiert (z.B. vom Rechnerwerk zum Arbeitsspeicher, vom Arbeitsspeicher zur Peripherie usw.). Zur Erhöhung der Störsicherheit kann die

Nachricht durch ein sogenanntes *Prüfbit* ergänzt werden , z.B. „die Anzahl der Einsen isr immer gerade“. ⇨ ist nach einem Datentransport die Anzahl der Einsen ungerade, so liegt ein Übertragungsfehler vor.

Aufgaben:

1. Man gebe eine Schaltung an, die jedem dreistelligen Binärwert ein Prüfbit hinzufügt.
2. Man gebe eine Formal zur einfachen Berechnung der Summe (von 1. ?) an!
   [Ergebnis: $(X_0 + X_1 + X_2)\text{MOD } 2$]

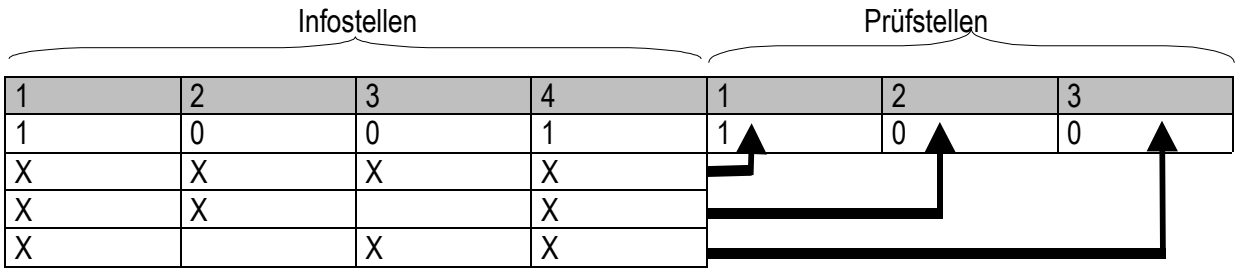
3. Man zeige, daß die Addition MOD 2 durch XOR implementiert werden kann.
   Hinweis: Schaltfunktion aus Aufgabe 1 mit XOR umschreiben!

4. Man beschreibe, wie sich das XOR, das die Addition MOD2 im Prüfbitgenerator bewirkt, durch NICHT, UND und OR beschrieben läßt (Schaltung skizzieren!).

5. Man schreibe ein Pascal- oder C-Programm, das die Schaltung aus Aufgabe 4 für den Prüfbitgenerator realisiert.

2. Das Verfahren von Hamming (1950)

Codewort = Informationsstellen + Prüfstellen
Die einzelnen Prüfstellen werden durch Paritätskontrolle ermittelt. Jedoch erkennen die einzelnen Paritätskontrollen nicht alle Infostellen, sondern nur eine geeignete Auswahl.



1	2	3	4	1	2	3
1	0	0	1	1	0	0
X	X	X	X			
X	X		X			
X		X	X			

Erklärung: Prüfstellen durch even- Parity ermittelt. Jedes Infobit ist an der Bildung mindestens zweier Prüfstellen beteiligt.

Annahme	fehlerhafte Prüfstellen		
Fehler in 1. Infostelle	X	X	X
Fehler in 2. Infostelle	X	X	
Fehler in 3. Infostelle	X		X
Fehler in 4. Infostelle		X	X

Beispile zur 1-Fehler-Korrektur:

Fehler: 1.Stelle: 0001100 ⇨ Paritätsvektor 100
⇨ ber. Paritätsvektor 011
⇨Fehler in erster Infostelle! Korrekt 1001100

Code-Konstruktion nach Hamming:

Annahme:

- Es gibt m Infostellen (mit je 1 Bit)
- Es gibt r Paritätsfunktionen, die auf bestimmten Teilmengen von {1;...;m} definiert sind (Paritätsgruppen)
- Paritätsgruppen erfüllen:

1. Jede Codestelle erscheint in mindestens 2 unterschiedlichen Paritätsgruppen.

2. zwei beliebige Codestellen werden durch mindestens zwei Paritätsgruppen getrennt.

Beispiel:

Annahme: $m = 2^r - r - 1$

m=4,r=3

Länge des Codewortes = $m + r = 2^r - 1 = 2^3 - 1 = 7$

Nun werden die Infobits neu numeriert mit aufeinander successierenden Zahlen die keine Zweierpotenz sind.

1 ⇨ 3 ; 2 ⇨ 5; 3 ⇨ 6; 4 ⇨ 7

Regel: Definitionsbereich der i-ten Paritätsfunktioon sei alle umnummerierten Codestellen, in denen an Der i-ten Stelle eine „1“ erscheint.

$$G^* = \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right), G = G^* T^T =$$

$$= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Erzeugematrix:

$$H^* = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Satz: $H^* G^{*T} = 0$

Deutung: $H^* v^T = 0$ für Codeworte v

Mit Hilfe von H^* lässt sich festellen ob z.B. 1101101 ein fehlerloses (1-Fehler) Codewort ist.

1101101 AND 0001100 = 0001100 (EVEN)
1101101 AND 0111010 = 0101000 (EVEN)
1101101 AND 1101001 = 1101001 (EVEN)

aus jeder Zeile ist die Parität zu bilden (I MOD 2)
⇨ Sie ist überall 0 (Nachweis der Zulässigkeit des Codeworts)

Was passiert wenn ein Fehler auftritt?

z.B. Codewort 1101101 wird zu 11911***1*** 1 verfälscht

1101111 AND 0001100 = 0001100 (EVEN/0)
1101111 AND 0111010 = 0101010 (***ODD/1***)
1101111 AND 1101001 = 1101001 (EVEN/0)

Die Paritäten 010 lassen unter Umständen sogar Rückschlüsse zu, *welcher* Fehler aufgetreten ist!

3.Lineare Codes

Grundidee: keine Angabe einer kompletten Übersetzungstabelle, sondern nur Angabe der Übersetzung von Datenwerten, die nur eine Eins und sonst nur Nullen enthalten.

Beispiel:

Übersetzungstabelle:

Datenwert				Codewort						
1	0	0	0	0	0	0	0	0	1	1
0	1	0	0	0	0	0	1	1	1	0
0	0	1	0	0	0	1	1	0	0	0
0	0	0	1	1	1	0	1	0	1	0

Formal gesehen ergibt sich der Code von „Vektor“ 1101 durch die Darstellung des „Vektors“ bezüglich seiner „Basis“:

1x0000011 = 0000011
1x0001110 = 0001110
0x0011000 = 0000000
1x1101010 = 1101010

1100111 (binäre Addition ohne Übertrag (XOR))

Problem: wie soll das Ergebnis interpretiert werden?

Normalform einer Erzeugermatrix: m=4, r=3

1000 XXX
0100 XXX
0010 XXX
0001 XXX (X=0 oder X=1 ⇨ Parity!)

Beispiel:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} = G^*$$

Das Datenwort 11001 wird jetzt zum bitweisen XOR codiert.

1x1000001 = 1000001
1x0100011 = 0100011
0x0010010 = 0000000 | XOR
1x0001111 = 0001111 ↓

1101101

Vorteil: saubere Trennung Daten-XOR und Prüfbits!

Beispiel: Erzeugermatrix für Hamming-Codes (m=4,r=3)

	Codestellen				Prüfbits		
	0	0	0	0	0	0	0
*	0	0	0	1	1	1	1
*	0	0	1	0	0	1	1
	0	0	1	1	1	0	0
*	0	1	0	0	1	0	1
	0	1	0	1	0	1	0
	0	1	1	0	1	1	0
	0	1	1	1	0	0	1
*	1	0	0	0	1	1	0
	1	0	0	1	0	0	1
	1	0	1	0	1	0	1
	1	0	1	1	0	1	0
	1	1	0	0	0	1	1
	1	1	0	1	1	0	0
	1	1	1	0	0	0	0
	1	1	1	1	1	1	1

Berechnung des Hamming-Codes für 1001 per Erzeugermatrix:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

⇨ 1001 =

1x1000110 = 1000110
0x0100101 = 0000000 | XOR
0x0010011 = 0000000
1x0001111 = 0001111 ↓

1001001 (vgl. Tabelle!)

Die Hauptfunktion der Erzeugermatrix ist die Berechnung zulässiger Codes!

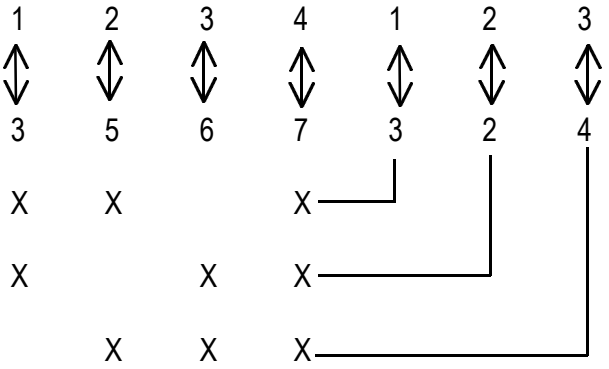
Frage: Wie sind obige Normalformmatrix und die Erzeugermatirx für das (3,5,6,7)-Verfahren mteinander verwandt?

Antwort: Durch eine Transformation T, die ($r_1,r_2,s_1,r_3,s_2,s_3,s_4$) in ($s_1,s_2,s_3,s_4,r_1,r_2,r_3$) abbildet.

$$T=\begin{pmatrix}0&0&0&0&1&0&0\\0&0&0&0&0&1&0\\1&0&0&0&0&0&1\\0&0&0&0&0&0&0\\0&1&0&0&0&0&0\\0&0&1&0&0&0&0\\0&0&0&1&0&0&0\end{pmatrix},T^{-1}=T^T\Rightarrow G^*=GT,G=G^*T^T$$

$$G=\begin{pmatrix}1&1&1&0&0&0&0\\1&0&0&1&1&0&0\\0&1&0&1&0&1&0\\1&1&0&1&0&0&1\end{pmatrix};G^*=GT=\begin{pmatrix}1&0&0&0&1&1&0\\0&1&0&0&1&0&1\\0&0&1&0&0&1&1\\0&0&0&1&1&1&1\end{pmatrix}$$

Wobei die Elementmultiplikation durch AND realisiert wird und die Addition sich als MOD 2 versteht (XOR!)



Codeberechnung für m=4;r=3

	BIN				Codewort							
	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	1	1	1	0	1	0	0	1	*
2	0	0	1	0	0	1	0	1	0	1	0	*
	0	0	1	1	1	0	0	0	0	1	1	
4	0	1	0	0	1	0	0	1	1	0	0	*
	0	1	0	1	0	1	0	0	1	0	1	
	0	1	1	0	1	1	0	0	1	1	0	
	0	1	1	1	0	0	0	1	1	1	1	
8	1	0	0	0	1	1	1	0	0	0	0	*
	1	0	0	1	0	0	1	1	0	0	1	
	1	0	1	0	1	0	1	1	0	1	0	
	1	0	1	1	0	1	1	0	0	1	1	
	1	1	0	0	0	1	1	1	1	0	0	
	1	1	0	1	1	0	1	0	1	0	1	
	1	1	1	0	0	0	1	0	1	1	0	
	1	1	1	1	1	1	1	1	1	1	1	

Gültige Hamming-Codes

Satz: 2 gültige Hamming-Codes müssen sich in mindestens 3 Infostellen voneinander unterscheiden!

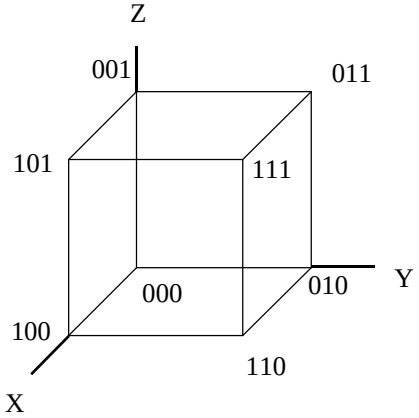
Annahme: nur ein Infobit würde bei der Übertragung verändert, dann ändert sich jede Paritätsfunktion,die dieses Bit erfasst! ⇨ ungültiger Code!

Annahme: nur zwei Infobits würden bei der Übertraguing verändert. Dann wähle man zwei Paritätsgruppen, die diese trennen. Diese Änderung bedutet: Der Wert beider zugeordneter Paritätsfunktionen würde geändert! ⇨ ungültiger Code!

Folgerung: gültige Hamming-Codes sind 1-Fehler korrigierbar!
Ändert sich ein einziges Infobit während der Übertragung, so ist der gültige Hamming-Code, Der sich nur in diesem Bit unterscheidet, der richtige.

Geometrische Deutung des Coderaumes

Jeder Binärstelle wird eine Dimension zugeordnet, d.h. alle n-stelligen Binärwerte liegen auf den Ecken eines n-dimensionalen Würfels.



X	Y	Z	
0	0	0	*
0	0	1	
0	1	0	
0	1	1	*
1	0	0	
1	0	1	*
1	1	0	*
1	1	1	

Von den 8 Codewörtern kann wegen Fehlererkennung bzw. Fehlerkorrektur nur ein Teil verwendet werden.
Dem vorliegenden Code können deshalb nur 2 Infostellen (+ 1 Prüfstelle) zugeordnet werden.
Dann sind beispielsweise nur die in der vorstehenden Tabelle mit „*“ markierten Codewörter zugelassen (im Coderaum hier jeweils 2 Kantenlängen voneinander entfernt).

Definition: Die Anzahl der Stellen in denen sich zwei Codeworte unterscheiden heisst Hamming-Abstand D.

X,Y Codeworte

1) $D(X,Y)=0 \Leftrightarrow X=Y$
2) $D(X,Y)=D(Y,X) \geq 0$
3) $D(X,Y) \leq D(X,Z)+D(Z,Y)$

Der kleinste Wert von D, der zwischen den Werten eines Codes auftritt ist die Mindestdistanz d .
„d“ ist ausschlaggebend für die Erkennbarkeit und Korrigierbarkeit von Fehlern!

Beispiele:

a) n=3 d=1

   Es ist unmöglich hier einen Fehler zu erkennenm da jeder Bitunterschied in einer Binärstelle zu 1 der 3 benachbarten Codewörter führt!

b) n=3 d=2

   Jede Verfälschung der 4 zugelassenen Codeworte (in ener Stelle) führt zu einer der 3 benachbarten nicht zugelassenen Kombinationen.
   ⇨ Fehler sind erkennbar!
   ⇨ Fehler sind nicht korrigierbar, da die nicht zugelassenen Kombinationen gleichwahrschienlich aus den 3 zugelassenen Kombinationen enstanden sein können.

c) n=d=3

   Egal welche 2 Wörter gewählt worden sind, haben alle 3 möglichen Fehlerkombinationen nur 1 Kantenlänge Abstand zu den zugelassenen Codewörtern.
   ⇨ Fehler sind korrigierbar!

4. Hamming-Codes

$G^* = GT$ oder $G = G^* T^T$

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} =$$

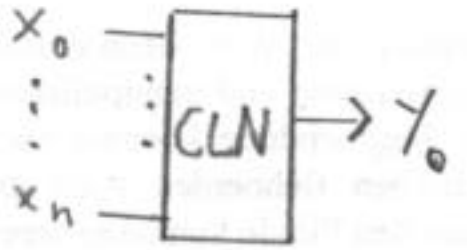
$$= \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} = H^* = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

$H^* v^T = (0/0/0)$ für gültige Codes.
Die Paritätsmatrix H, die auf dem ursprünglichen Codewortsystem definiert ist, soll bestimmt werden, d.h. H^* wird mittels T (d.h. $T^{-1} = T^T$) „rücktransformiert“.

$$H = H^* T^T = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

3.3.Schaltwerke

Bisher: Automaten hatten keine internen Zustände

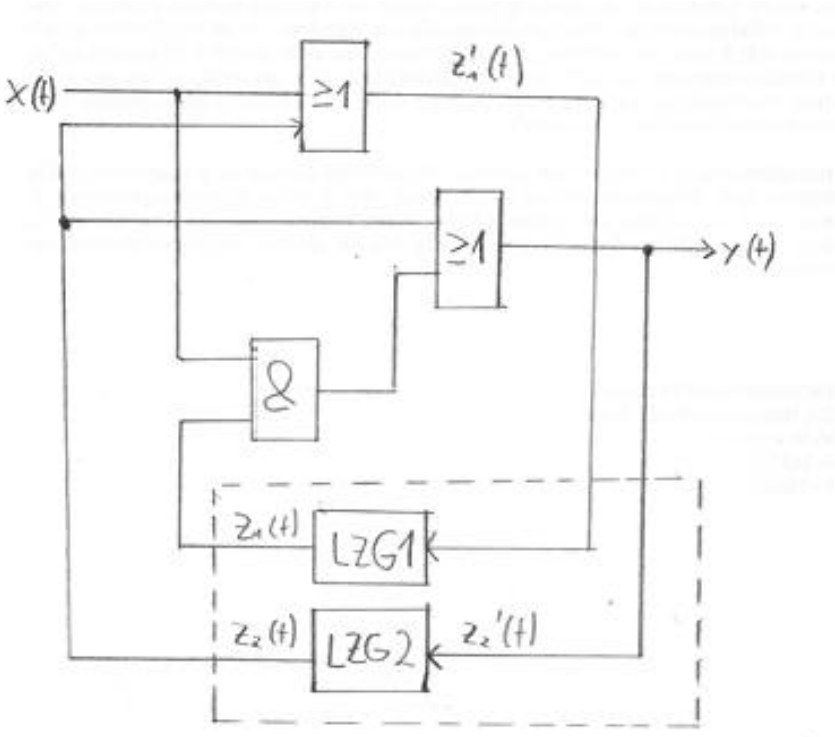


(Ausgabe hängt nur von der Eingabe ab!)

CLN = **C**ombinatorical **L**ogical **N**etwork

Bei Schaltwerken ist die zeitliche Dimension wichtig. Zu gewissen äquidistanten Zeitpunkten geschieht das Lesen eines Eingabezeichens, die Ausgabe einens Ausgabezeichens und der Übergang in den Folgezustand.

Beispiel: Analyse eines sequentiellen Schaltnetzes



LZG = **L**auf**z**eit**g**lied

Zur Zeit t vorliegende Signale:

- Eingangssignal X(t)

- Ausgangssignal Y(t)
- Interne Signale
  - $Z_1'(t)$ am Eingang von LZG 1
  - $Z_1(t)$ am Ausgang von LZG 1
  - $Z_2'(t)$ am Eingang von LZG 2
  - $Z_2(t)$ am Ausgang von LZG 2

Die an den Laufzeitgliedern LZG1 und LZG2 zur Zeit t anliegenden Signale werden um eine Zeiteinheit verzögert und stehen daher zur Zeit t+1 am Ausgang der LZG zur Verfügung.

$Z_1(t+1)=Z_1'(t)$; $Z_2(t+1)=Z_2'(t)$

Über eine Wertetabelle lassen sich die Werte der abhängigen Variablen $Y(t), Z_1'(t), Z_2'(t)$ ermitteln.

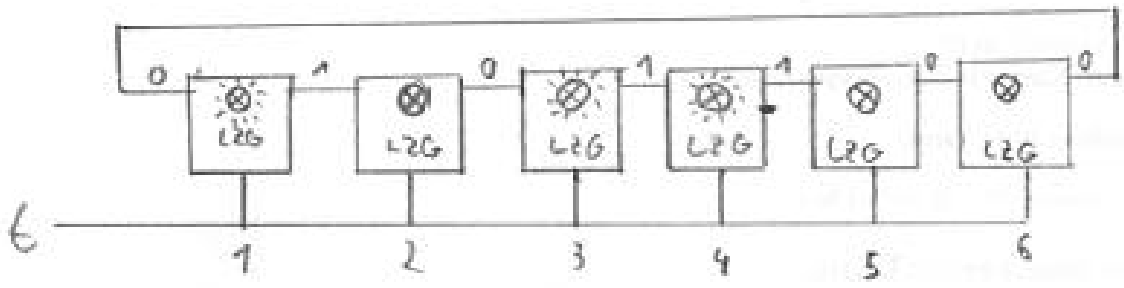
X(t)	$Z_1(t)$	$Z_2(t)$	$Z_1'(t)$	$Z_2'(t)$	Y(t)
0	0	0	0	0	0
0	0	1	1	1	1
0	1	0	0	0	0
0	1	1	1	1	1
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

$Z_1(t)$, $Z_2(t)$ kennzeichnen den gegenwärtigen Zustand des Schaltwerkes. $Z_1'(t)$, $Z_2'(t)$ bestimmen den Folgezustand des Schaltwerkes.
KV-Diagramme können hier verwendet werden.

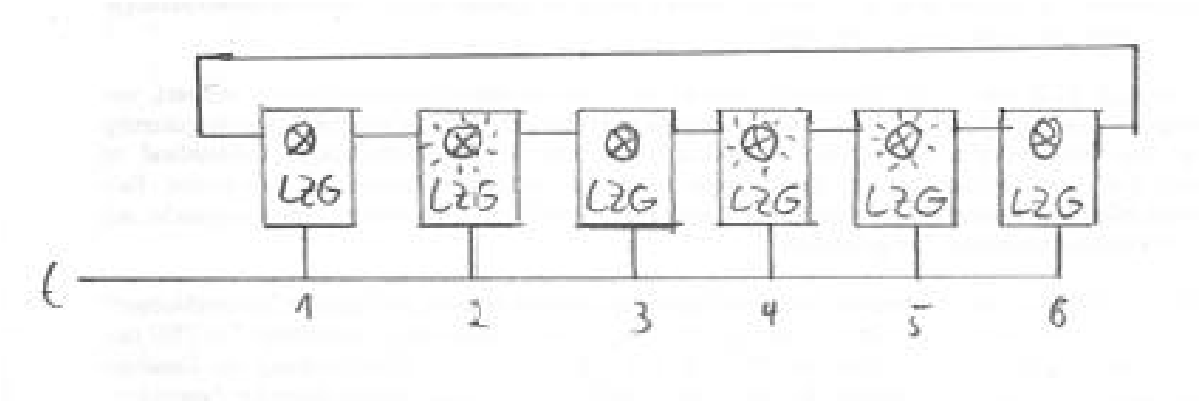
Ein Schaltwerk berücksichtigt zeitliche Vorgänge und kann auf einen Automaten zurückgeführt werden. Jeder endliche Automat lässt sich durch ein geeignetes Schaltwerk realisieren.

Definition:
Ein endlicher Automat heißt binäres Schaltwerk, falls die Eingabemenge und die Zustandsmenge aus binären Werten bestehen.
Ausgabe- und Übergangsfunktionen sind Schaltnetze.

Schaltet man mehrere LZG hintereinander, z.B.

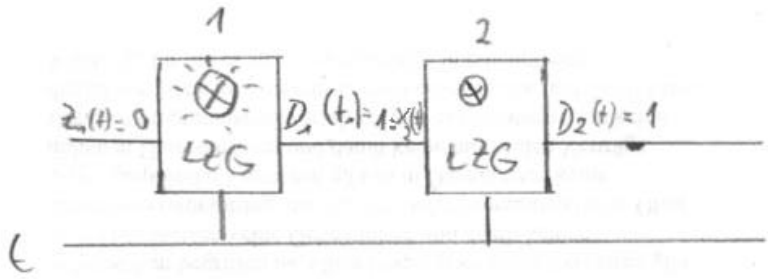


Dann liegt eine Taktperiode später folgende Situation vor:



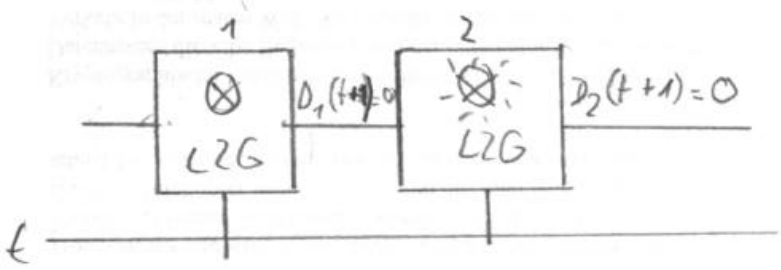
Das Muster aus brennenden und nicht brennenden Lämpchen zeigt den intrnen Zustand des jeweiligen Kippgliedes und ist nach einer Taktperiode um einen Schritt verschoben.

D(t+1)=X(t)



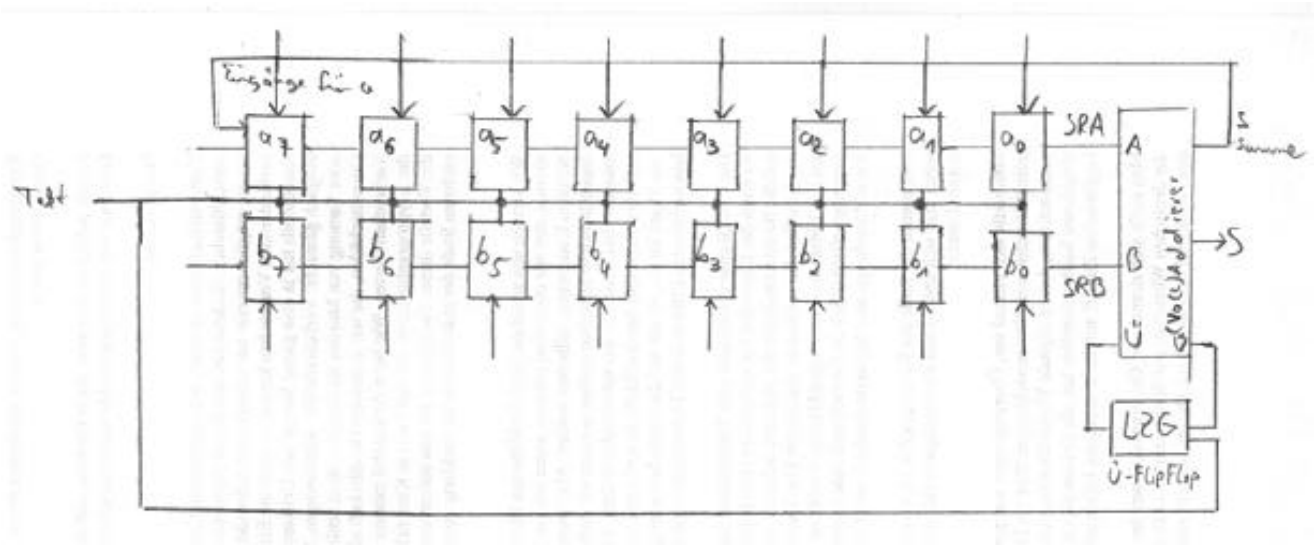
D(t)=1 ⇨ Lampe leuchtet, D(t)=0 ⇨ Lampe dunkel.

Lampe leuchtet ⇨ $D_1(t)=1$ aber 0 liegt am Eingang ⇨ $X_1(t)=0$. Wenn t auf eins geht, geht Kippglied 1 in Zustand 0 ⇨ Lampe erlischt und der Ausgang von Kippglied 1 nimmt den Wert 0 an. Zur gleichen Zeit wird bei Kippglied 2 der Wert $X_2(t)$ durchgeschaltet ⇨ Lampe von Kippglied 2 leuchtet und $D_2(t+1)=1$



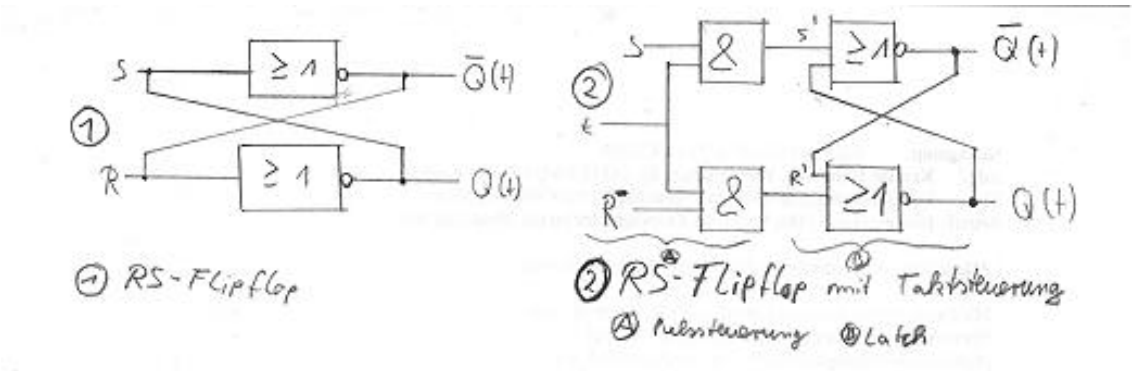
Zur Zeit t+1

Ringschieberegister: serieller Addierer



- Bitweise Addition über Volladdierer in Taktschritten (Die Anzahl der Bits ist hier prinzipiell unbegrenzt!)
- Speicherung des Übertrags im Übertragsflipflop
- Beim nächsten Taktschritt wird der Übertrag über den Carryeingang des Addierers zur nächsten Dualstelle hinzuaddiert.
- Nach 8 Taktschritten steht die Summe der beiden Dualzahlen im Schiebregister SRA zur Verfügung und kann dort seriell abgenommen werden.

Flipflops sind die einfachste Form eines sequentiellen Schaltnetzes, z.B., über NOR/NAND-Gatter realisiert:



Ein RS-Flipflop hat folgende Wahrheitstabelle:

R	S	Q(t)	Q(t+1)	Zustand
0	0	0	0	Speichern
0	0	1	1	Speichern
0	1	0	1	Setzen
0	1	1	1	Setzen
1	0	0	0	Rück-
1	0	1	0	setzen
1	1	0	Diese Konstellationen	
1	1	1	würden zum Osszillieren des FFs führen!	

Ohne Pulssteuerung:

R	S	Q	
0	0	Q(t)	Speichern
0	1	1	Setzen
1	0	0	Rücksetzen
1	1	--	undefiniert

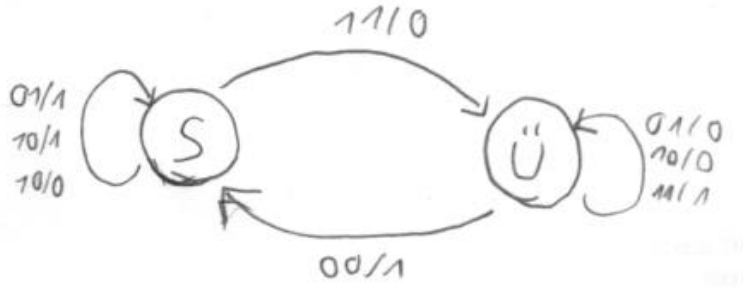
Aus dem KV-Diagramm

Q(t+1):

Q \ R,S	0,0	0,1	1,1	1,0
0	0	1	? undef.	0
1	1	1	? undef.	0

Lässt sich die Übergangsfunktion Q(t+1) schreiben:

Q(t+1)= $\bar{R}S+\bar{R}Q(t)$ S=SR+S $\bar{R}$ = S $\bar{R}$, da SR=NIL ist (nicht definiert!)
⇨ Q(t+1)=S+ $\bar{R}Q(t)$



<u>Automatentafel:</u>

Akt.	00		01		10		11	
Zust.	A	FZ	A	FZ	A	FZ	A	FZ
S	0	S	1	S	1	S	0	Ü
Ü	1	S	0	Ü	0	Ü	1	Ü

Wertetabelle

Eingabe		Zustand	Ausgabe	Folgezustand
X_1	X_2			
0	0	0 (=S)	0	0
0	1	1	1	0
0	1	0	1	0
0	1	1	0	1 (=Ü)
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

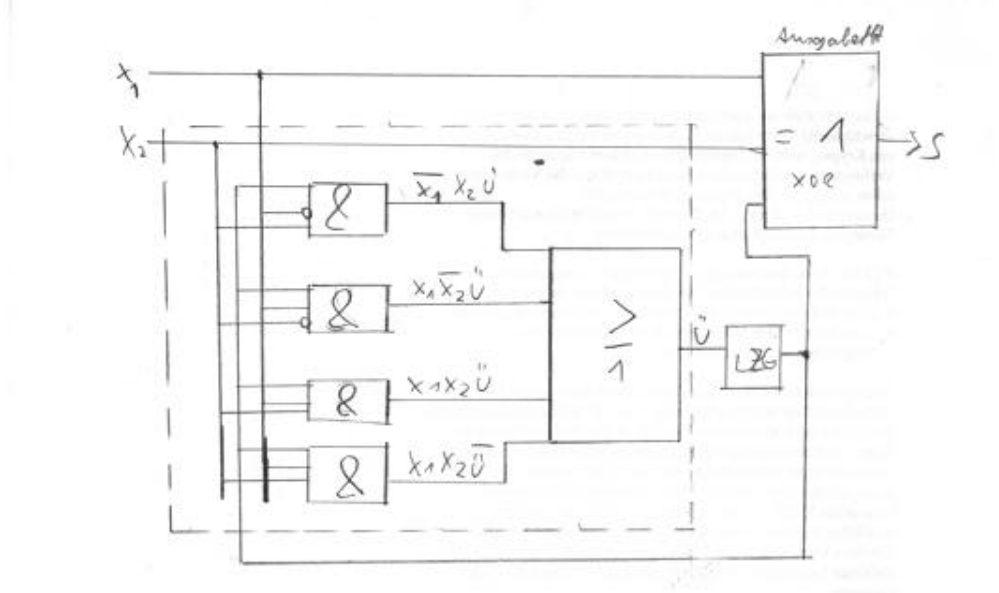
Ausgabefkt.

Übergangsfkt

⇨ $S = \bar{X}_1\,\bar{X}_2\,Ü + \bar{X}_1\,X_2\,\bar{Ü} + X_1\,\bar{X}_2\,\bar{Ü} + X_1\,X_2 Ü = (\bar{X}_1\,X_2 + X_1\,\bar{X}_2)Ü + (\bar{X}_1\,X_2 + X_1\,\bar{X}_2)\bar{Ü}$
$= (\overline{(X_1 \oplus X_2)})Ü + (X_1 \oplus X_2)\,\bar{Ü} = (X_1 \oplus X_2) \oplus Ü$

$Ü = \bar{X}_1\,X_2\,Ü + X_1\,\bar{X}_2\,Ü + X_1\,X_2\,\bar{Ü} + X_1\,X_2\,Ü = (\bar{X}_1\,X_2 + X_1\,\bar{X}_2 + X_1\,X_2)Ü + X_1\,X_2\,Ü$

Schaltung:



Satz: jeder endliche Automat kann über ein sequentielles Schaltnetz realisiert werden

Beispiel:

Zustandsdiagramm:

Akt. Zust.	00		01	
	A	FZ	A	FZ
Z_0	0	Z_1	0	Z_0
Z_1	1	Z_2	1	Z_1
Z_2	1	Z_2	0	Z_0

Die Zustände sind hier geeignet zu codieren.
Da 3 Zustände vorliegen bietet sich an:

(D ist Bezeichnung des LZGs ⇨ D-FlipFlop)

	D_1	D_2
Z_0	0	0
Z_1	0	1
Z_2	1	0

Wertetabelle

	Zust.			Folgezust.	
X(t)	$D_1(t)$	$D_2(t)$	Y(t)	$D_1(t+1)$	$D_2(t+1)$
0	0	0	0	0	1
0	0	1	1	1	0
0	1	0	1	1	0
1	0	0	0	0	0
1	0	1	1	0	1
1	1	0	1	0	0
↑ Eing.			↑ Ausgabfkt	↑ Übergangsfunktionen	

Bem: Hier sind $D_1=D_2=1$ „don't-care"-Terme!

⇨ $Y(t) = \bar{X}\bar{D}_1D_2 + \bar{X}D_1\bar{D}_2 + X\bar{D}_1D_2 + XD_1\bar{D}_2 =$
$= \bar{D}_1D_2(\bar{X} + X) + D_1\bar{D}_2(\bar{X} + X) = \bar{D}_1D_2 + D_1\bar{D}_2$

⇨ Ausgabe hängt nicht direkt von der Eingabe ab!

$D_1(t+1) = \bar{X}\bar{D}_1D_2 + \bar{X}D_1\bar{D}_2$
$D_2(t+1) = \bar{X}xx\bar{D}_1\bar{D}_2 + X\bar{D}_1D_2$

Minnimierung:

Y(t)

$D_2(t)$ \ X	0	1	1	0
1	X	X	1	

$D_1(t)$

$Y(t)=D_1 + D_2$

$D_1(t+1)$

$D_2(t)$ \ X	0	1	0	0
1	X	X	0	

$D_1(t)$

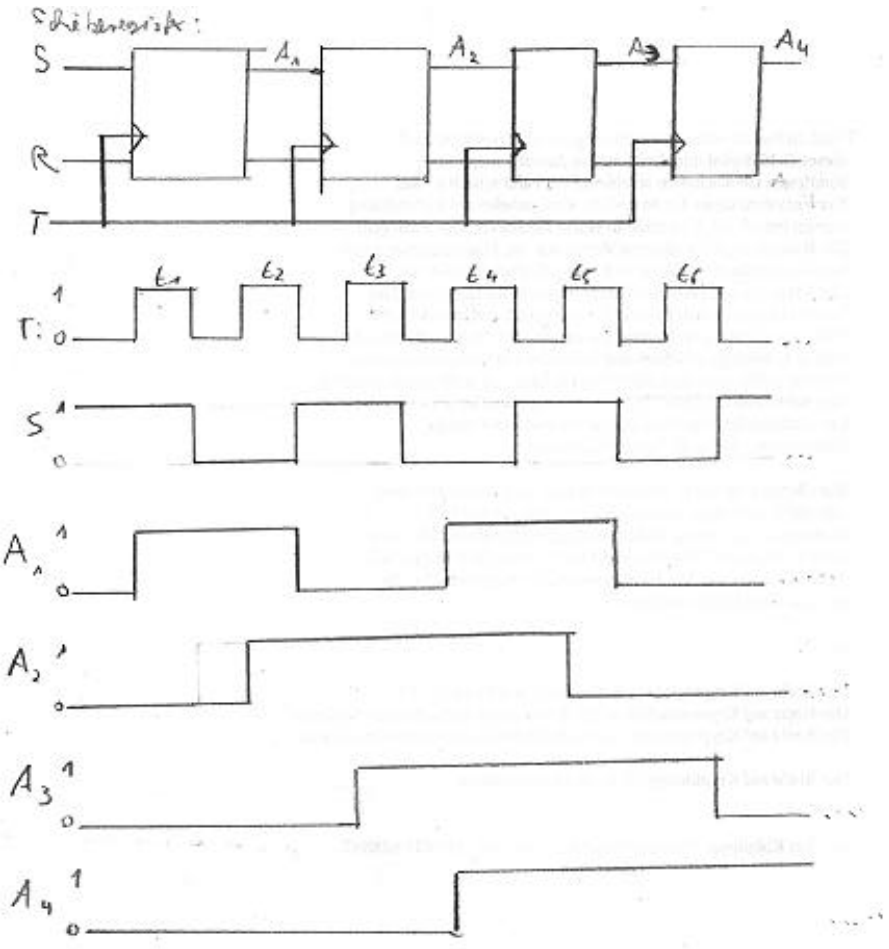
$D_1(t+1) = \square XD_1 + \square XD_2 = \square X(D_1+D_2)$

$D_2(t+1)$

$D_2(t)$ \ X	1	0	0	0
0	X	X	1	

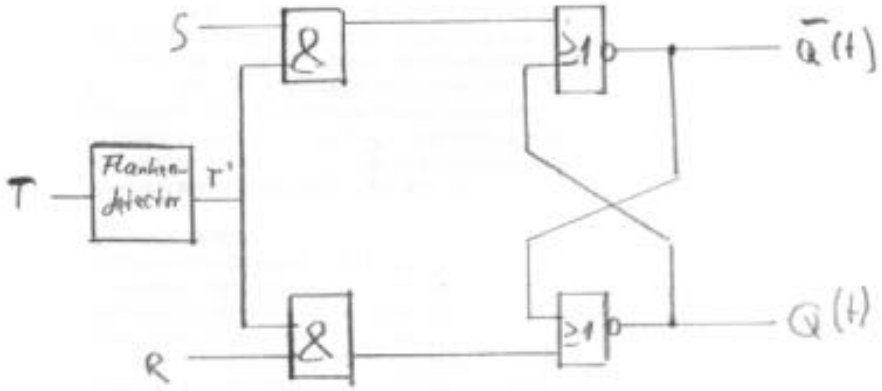
$D_1(t)$

$D_2(t+1) = XD_2 + \square X\square D_1\square D_2$

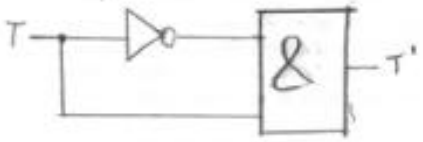


Problem:
Der Inhalt jedes Speicherelements muß genau um eine Position pro Taktimpuls verschoben werden!
RS-FF nicht geeignet, da Eingabe sich um mehrere Stellen pro Taktimpuls propagieren ann.

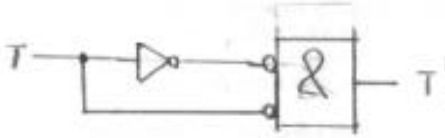
Lösung: Taktflankengesteuerte FFs, oder Master_Slave-FFs (FFs mit Zwischenspeichern)



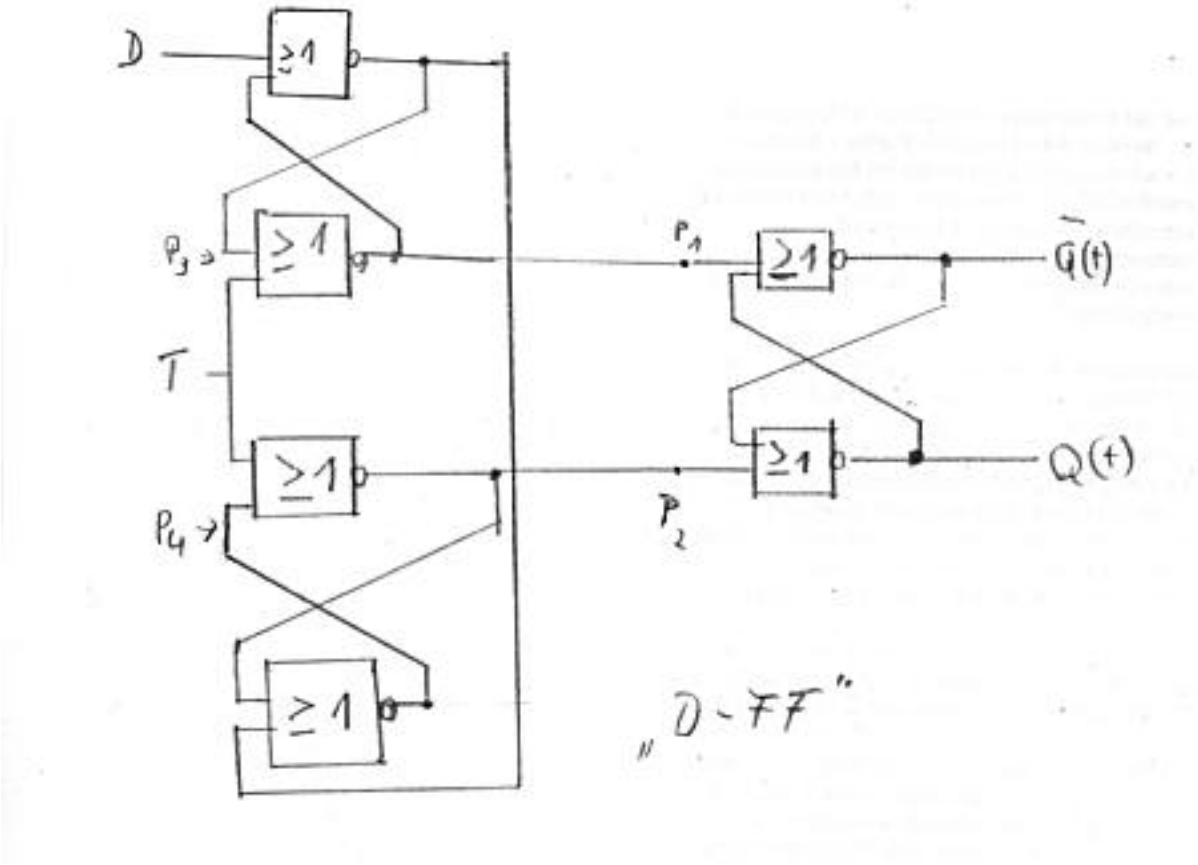
1) positive Flanken-Trigger



2) negative Flanken-Generator



Negative Flankengesteuerte FFs



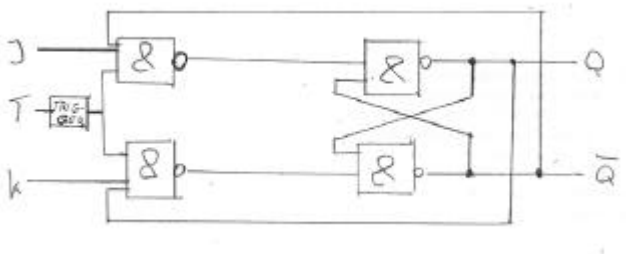
Aanalyse

1. T=1 ⇨ $P_1 = P_2 = 0$ ⇨ FF speichert, ausserdem ist $P_3 = \bar{D}$ und $P_4 = D$ solange T= 1 bleibt.

   Annahme: D ändert sich nicht während eines kürzeren Zeitintervalls vor („Setup Time") und nach („Hold Time") dem Übergang T ⇨ 0.

2. T ⇨ 0 : $P_1 = \bar{P}_3 = D$; $P_2 = \bar{P}_4 = \bar{D}$ ⇨ FF speichert D
3. Während T=0 werden Änderungen an D am Ausgangs-FF nicht gespürt!
4. T ⇨ 1 : $P_1 = P_2 = 0$, d.h. Ausgangs-FF speichert wieder, aber 0 an 2. Eingang von NOR-Glied Nr. 1 wieder Aufnahme von D (d.h. $P_3 = \bar{D}$ und $P_4 = D$)

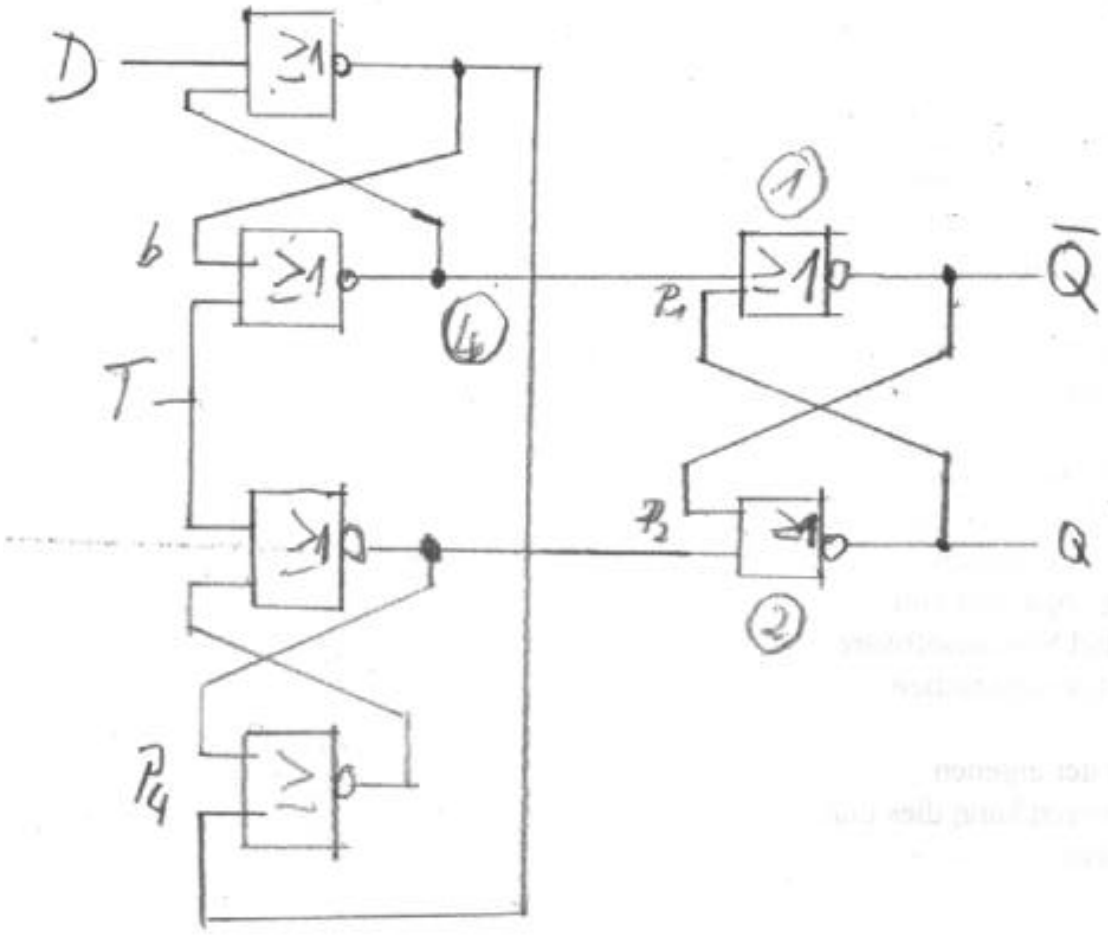
JK-FF – definiert auch für J=K=1, sonst RS



J	K	Q
0	0	Speichern
1	0	Setzen
0	1	Rücksetzen
1	1	Kippen

Übung:

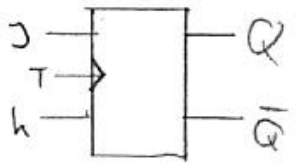
1) Tabelle verifizieren
2) Warum darf dieses FF nicht im untaktflankierten Zustand betrieben werden (d.h. Eingabe geht direkt an Ausgabe)



Behauptung: Während T=0 werden Änderungen an D am Ausgangsff nicht gespürt.

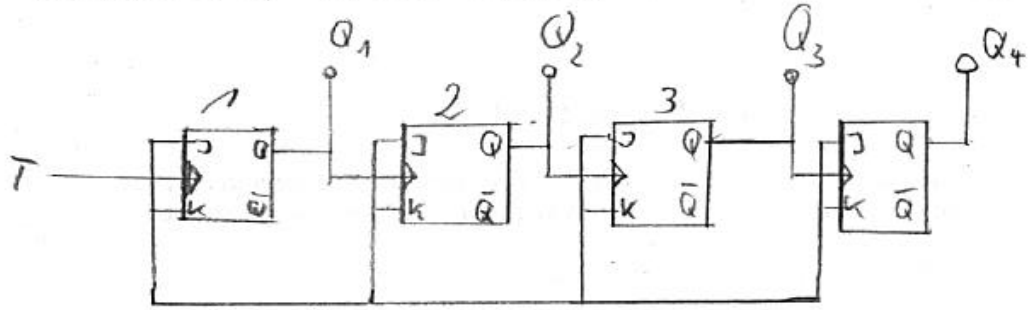
Fall A: D=1 während T ➔ 0 → P_1 = 1 → Eingänge zu NOR-Glied (1) ist dann sets 1 → D blockiert,

Fall B: D=0 während T ➔ 0 → Eingänge zu den NOR-Glied (2) und (4) sind 1, d.h. P_1 bleibt 0 auch während T=0
Auch P_4 = 0, T=0 → P_2 ist 1 auch während T=0

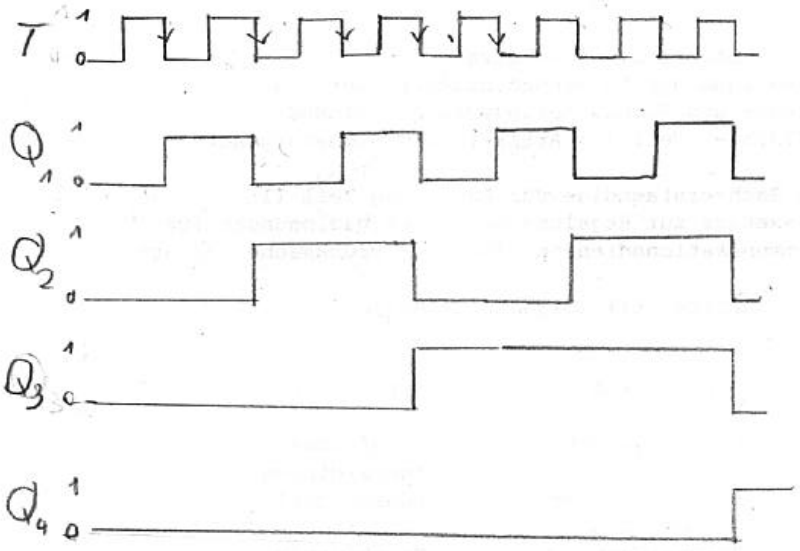


JK-FF als "Black-Box"

Anwendung: (Dual-) Zähler



Erklärung: J=K=1 → FF#1 funktioniert wie ein Kippglied (Toggle), t.h. sin Zustand Q(t) ändert sich mit jedem Taktpuls → nach 2 Taktpulsen ändert sich FF#1 vom ursprünglichen Zustand Q(t) in seinen gekippten Zustand Q(t+1)= Q(t) und liefert dann seinen ursprünglichen Zustand Q(t+2)=Q(t) zurück.



Da FF#2 von FF#1 gesteuert wird benötigt er 4 Taktpulse für einen Zyklus.

t = 0 → 0000 ; t=1 → 1000; t=2 → 0110; t=3 → 1100; t=4 → 0010; t=5 → 1010

Nachteil:langsam weil sequenteiller Zähler!!
In der Praxis werden Zähler "simultan", d.h. bei gemeinsamen Takt inkrementiert.

3.4.Sprachen endlicher Automaten

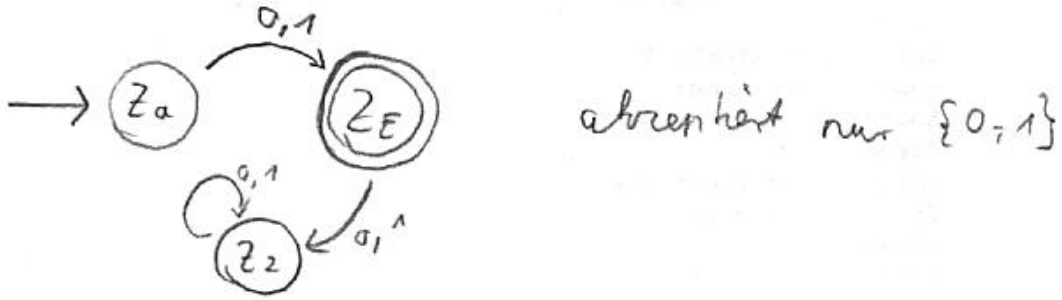
3.4.1. Grundlagen

Relevant ist nur der Endzustand und die Eingabefolge, die diesen Zustand erzeugt hat. Die Ausgabe ist unwichtig.
Zweckmässig: Einem Automaten einen Anfangszustand Z_a und einen (oder mehrere) Endzustand Z_E Zuzuordnen.
Eine Eingabezeichenfolge (Wort), die der Automat akzeptiert führt von Z_a nach Z_E

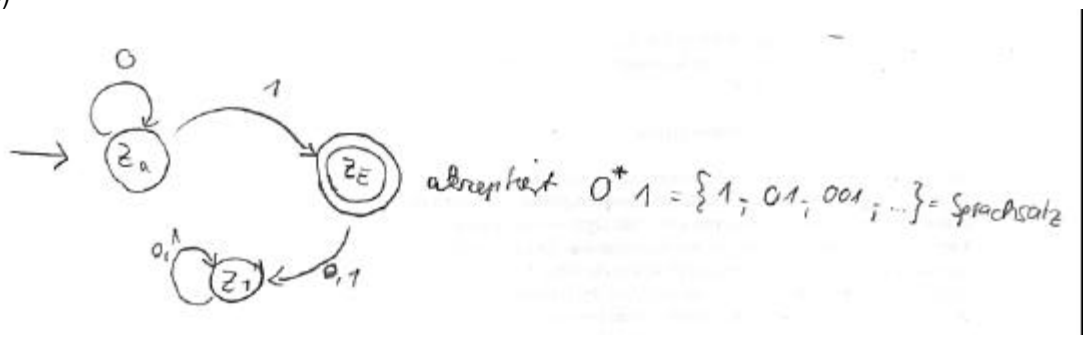
Beispiele:

a)

b)



c)



Ein endlicher Automat ist bestimmt durch

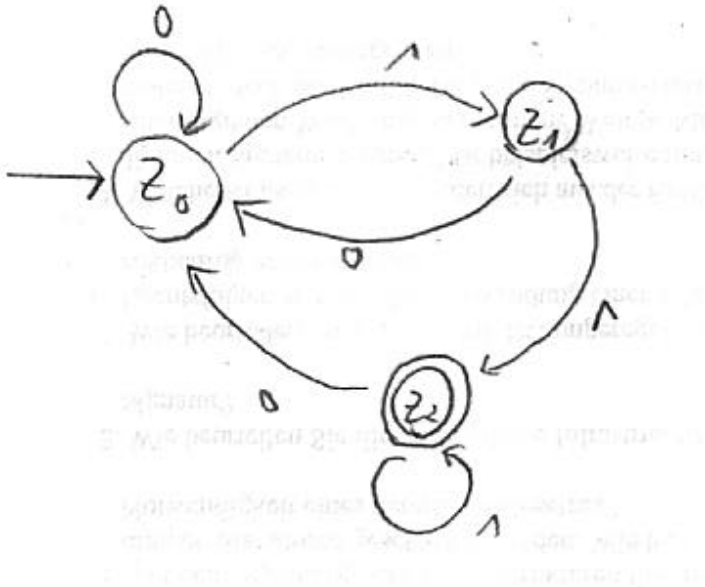
- Menge X („Alphabet“)
- Menge Z (Zustände)

- Übergangsfunktion f:X=Z→Z
- Anfangszustand Z_a aus Z
- Eine Endzustandsmenge Z_E die eine Teilmenge von Z ist

Beispiel:

Automat $A(X,Z,f,Z_A,Z_E)$

$X=\{0;1\}$ $Z=\{Z_0,Z_1,Z_2\}$ $Z_e=\{Z_2\}$ $Z_A = Z_0$



f	0	1
Z_0	Z_0	Z_1
Z_1	Z_0	Z_2
Z_2	Z_0	Z_2

Eingabe	Endzustand
1.01**11**	Z_2
2.001101	Z_1
3.110100	Z_0
4.0110**11**	Z_2

Erkennbar: alle Zeichenketten die auf ,

Erweiterung von f auf Zeichenketten über X

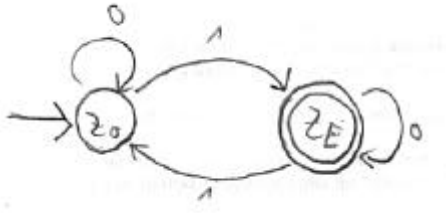
X Alphabet → X^* = Menge aller Zeichenketten über X
X^* umfasst alle Wörter über X (auch das leere Wort ë)

Für W ª X^* : w=X □X; X^a X, □X^a X^*;z ª Z

→ f(w,z) = f(□X,f(x,z)); f(ë,z)=z (rekursiv!)
x^a X^* heißt Wort

Geht ein Automat bei der Verarbeitung eines Eingabewortes in Einen Endzustand (aus Z) über, dann hat der Automat dieses Wort aktzeptiert .
Die Menge aller Eingabewörter, die der Automat akzeptiert, heißt die Sprache L(A) des Automaten.

Beispiel:

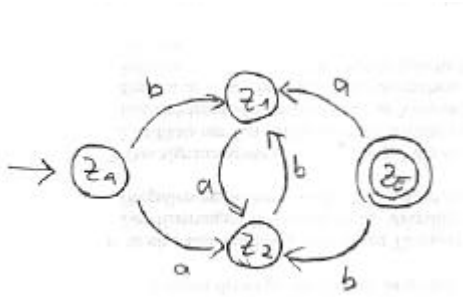


L(A) = alle Zeichenketten mit ungerader Anzahl von Einsen.

Aufgaben:

1. Man zeichne das Zustandsdiagramm eines Automaten mit X={a,b}, bei dem alle Eingaben in den Endzustand Z_E führen, die mit a beginnen und mit a enden und aus mindestens 2 Zeichen bestehen

2. Welche Eingaben führen in Z_E?



3. Wie sieht das Zustandsdiagramm eines Automaten aus, mit X={0,1}, bei dem nur solche Eingaben in den Endzustand führen, bei denen eine gerade Anzahl von Nullen und eine gerade Anzahl von Einsen vorkommt und Z_A + Z_E ?
4. Wie sieht ein Automat A aus, der bei X={a,b,c} genau dann in Z_E ist, wenn bei der Eingabe genau an der 1,3,5,7,... Stelle ein „a“ ist?
5. Man konstruiere A mit X={a,b} und L(A)={b}
6. X={a,b,c}, A ist so zu konstruieren, daß L(A)={w|enthält nach jedem a genau ein b}

   z.B. w^a L(A) = abcccab
   w = bcbb
   w = bcab

   w nicht aus L(A) = abbc
   w = bccaa
   w = aabb

Beispiel:
Zahlendarstellung in Pascal

Gültig: 12; -1.42; 2.3E+4
Ungültig: .6 ; 6. ; 4,23E-6

Beschreibung mittels Bachus-Naur-Form (BNF):

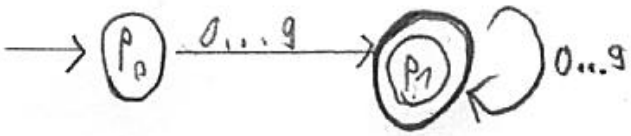
<number> ::= <unsigned>|<signed><unsigned number>
<unsigned number> ::= <unsigned integer>|<unsigned Real>
<unsigned integer> ::=<digit>{<digit>}
<unsigned real> ::= <unsigned Integer>.<digit>{<digit>}|<unsigned integer>.<digit>{<digit>}E<scale factor>|<unsigned integer>E<scale factor>

<scale factor> ::= <unsigned integer>|<sign><unsigned integer>
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<sign> ::= + | -

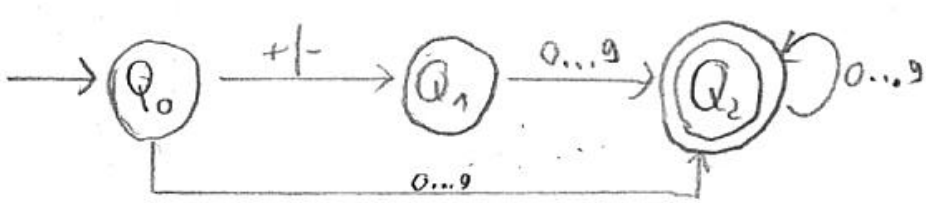
Alphabet = {0,1,...,9,.,E,+,-}

< > = „nicht terminale Symbole“
::= = „ist“
| = oder
{...} = beliebige Wiederholung (auch 0x !)

1) Erkennen von ganzen Zahlen

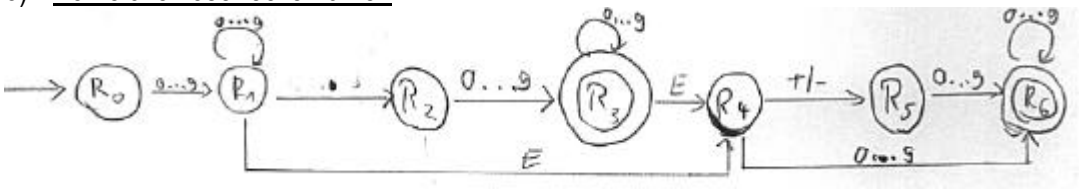


2) <u>Exponent</u>

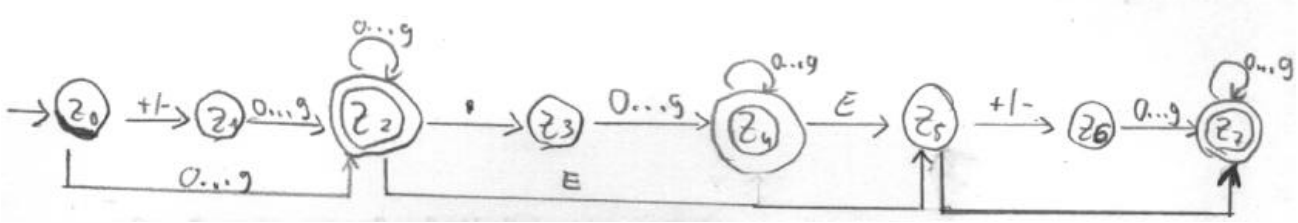


<u>Achtung:</u> Fehlerzustand sollte beigefügt werden!

3) <u>Vorzeichenlose reelle Zahlen</u>



4) beliebige Pascal-Zahlen



Formale Sprachen

Formale Sprachen bestehen aus

- einem Alphabet (eine endliche,geordnete Menge von Zeichen)
- Zeichenketten/Wörter über dem Alphabet („Tokens“)
- Einem System von Regeln: bestimmen ob eine Konstruktion zur Sprache gehört, oder nicht. Wird auch Grammatik genannt.

Beispiel: eine einfache formale Sprache:

X = {a,b} <s> Startsymbol

(1) <s>::=a (d.h. für ‚<s>‘ darf ‚a‘ eingesetzt werden)
(2) <s>::=a<s>a (d.h. für ‚<s>‘ darf ‚a<s>a‘ gesetzt werden)
(3) <s>::=<s>b (d.h. für ‚<s>‘ darf ‚<s>b‘ gesetzt werden)

→ Wörter:

a) <s> —(2)→ a<s>a —(2)→ aa<s>aa —(1)→ aaaaa (alle Wörter mit ungerader Anzahl von ‚a‘)
b) <s> —(3)→ <s>bb —(3)→ <s>bbb —(1)→ abbb (alle Wörter mit 1x ‚a‘ vorne und beliebig vielen b)
c) <s> —(3)→ <s>b —(2)→ a<s>b —(3)→ a<s>bab —(1)→ aabab

aba nicht erzeugbar!

3.4.2. reguläre Sprachen

Die Menge aller Sprachen, die von endlichen Automaten akzeptiert werden, heißen „reguläre Sprachen“.

Definition: Eine grammatik heisst regulär, wenn alle Produktionen von der Form

R) <A>::=a<B>|a (Rechtslinearität)
L) <A>::=<B>a|a (Linkslinearität)

sind. Die zugehörige Sprache heißt regulär.

Satz (v.Kleane): Jede von einem endlichen Automaten erkannte Sprache ist regulär. Zu jeder regulären Sprache gibt es einen endlichen Automaten, der sie erkennt.

reguläre Ausdrücke bezeichnen Mengen von Wörtern über einem Alphabet X

X^* = {Zeichenketten über X eischl. Des leeren Wortes} = {alle Wörter über X}

z.B. $\{0,1\}^*$ = {ë,0,1,00,01,10,11,000,...}

Eine Sprache L über X ist eine Menge solcher Wörter
L_1 und L_2 seien solche Mengen

Solche Mengen sind

- die Verkettung (Produkt) von L_1 und L_2 : $L_1 * L_2 = \{w_1w_2 \in X^* \mid w_1 \in L_1 , w_2 \in L_2\}$
- die Potenz $L^{(i)}$ von L : $L^{(i)} = \{ë\}$, $L^{(i+1)} = L^{(i)} + L$ ($i \geq 1$)
- die Vereinigung (Closure) $L^* = \bigcup_{i=0}^{\infty} L^{(i)}$

von Sprachen. L^* besteht also aus der Menge aller Wörter, die durch Hintereinanderschaltung von Wörtern aus L entstehen.

Definition: reguläre Ausdrücke:

1) {} (leere Menge) ist ein regulärer Ausdruck über X und L({}) = {}
2) ë (leeres Wort) ist ein regulärer Ausdruck über X und L(ë) = {ë}
3) Für jeden Buchstabe $x \in X$ ist x ein regulärer Ausdruck über X und L(x)={x}
4) Sind a und b reguläre Ausdrücke über X, dann sind auch reguläre Ausdrücke:

   a^*,(ab),$(a \cup b)$,b^*
   $L(a^*) = L^*(a)$; L(ab) = L(a)*L(b),$L((a \cup b)) = L(a) \cup L(b)$

5) Nur die nach 1)-4) erzeugten Ausdrücke sind regulär!

Reguläre Ausdrücke a und b sind äquivalent falls L(a) = L(b) ist!
Aussenklammern können weggelassen werden!
‚*' hat höchsten Vorrang, dann Verkettung, dann Vereinigung.

Beispiel:

X={a,b}
$((a \cup b)^*((aa) \cup (bb)(a \cup b)^*))$ ist ein regulärer Ausdruck.
Er kann so geschrieben werden:
$(a \cup b)^*(aa \cup bb)(a \cup b)^*$
$0(0 \cup 1)^*$ = Menge aller Bitfolgen, die mit 1 anfangen
$(1 \cup 2 \cup 3 \cup 4 \cup 5 \cup 6 \cup 7)(0 \cup 1 \cup 2 \cup 3 \cup 4 \cup 5 \cup 6 \cup 7)^*$ = Menge aller natürlichen Zahlen in Oktaldarstellung
$(0 \cup 1)^*0(0 \cup 1)(0 \cup 1)$= Menge aller Bitfolgen, die als dritletztes Bt 0 haben.
$1(0 \cup 1)^*1 \cup 1$ = Menge aller Bitfolgen, die mit 1 beginnen und enden.

3..4.3. Erkennende Automaten

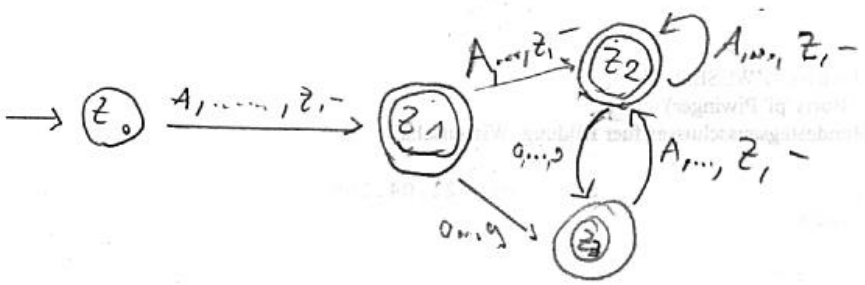
Beispiel: Bezeichner in C und Pascal:

Können verwendet werden für (1) Variablennamen, (2) Prozedur-
/Funktionsnamen,(3)Programmnamen,(4)reservierte Wörter

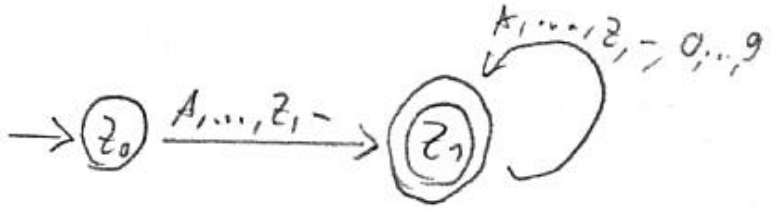
Gestalt:

<identifier> ::= <letter>{<letter>|<digit>}
<letter> ::= A|B...Z|a|b|...|z|_
<digit> ::= 0|1|...|9

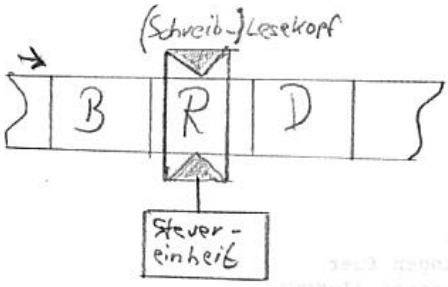
Erkennender Automat:



Kann so umgeschrieben werden:



Arbeitsweise eines erkennenden Automaten:



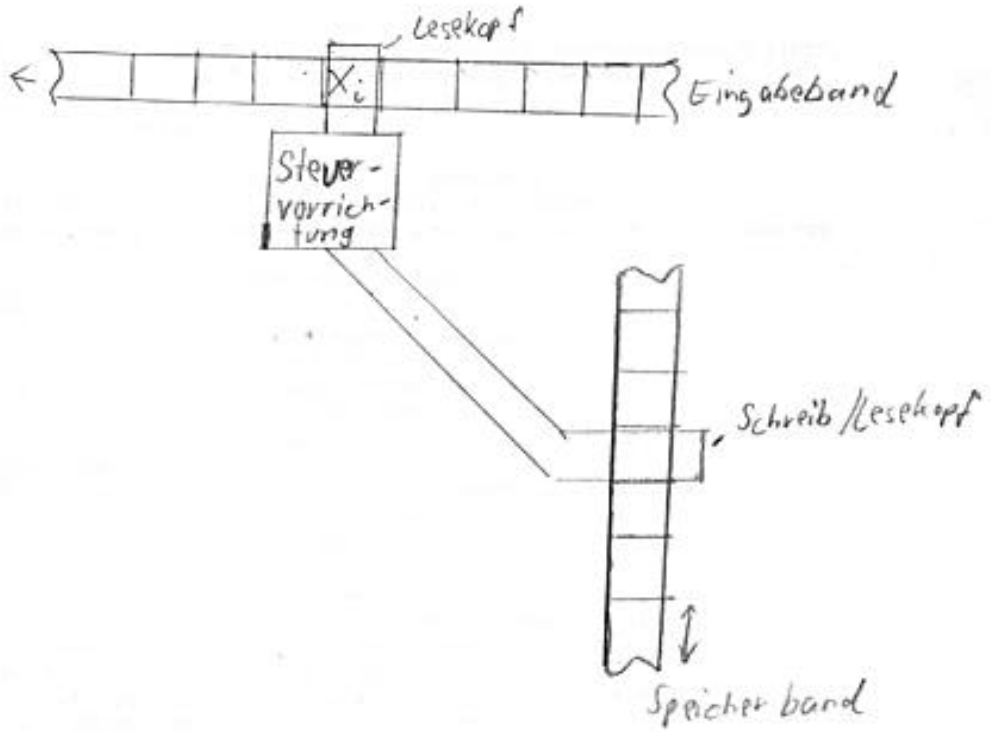
Die zu prüfende Zeichenkette steht auf einem Band, das in Felder eingeteilt ist und auf dem Lesekopf von links nach rechts bewegt werden kann.
Es gibt Mengen von Wörtern (über X), die von keinem Automaten akzeptiert werden.
X={a,b} L(A)={$a^m b^m$|m 0}

Kapitel IV Kellerautomaten (Push-down-Automaten)

4.1. Das Automatenmodell

Kellerautomat

- Eingabeband
- Steuervorrichtung
- Speicherband (Keller/Stack)



Eingabeband:

- Kann gelesen und jeweils um ein Feld weiterbewegt werden

Speicherband:

- Hat ein unteres Ende
- Kann in beide Richtungen bewegt werden
- Kann gelesen und beschrieben werden
- Zeichen unter Schrieb/Lesekopf ist oberstes Zeichen („TOC“)

Definition: ein (deterministischer) Kellerautomat $KA=(X,Z,K,f,Z_A,K_A,F)$ besteht aus

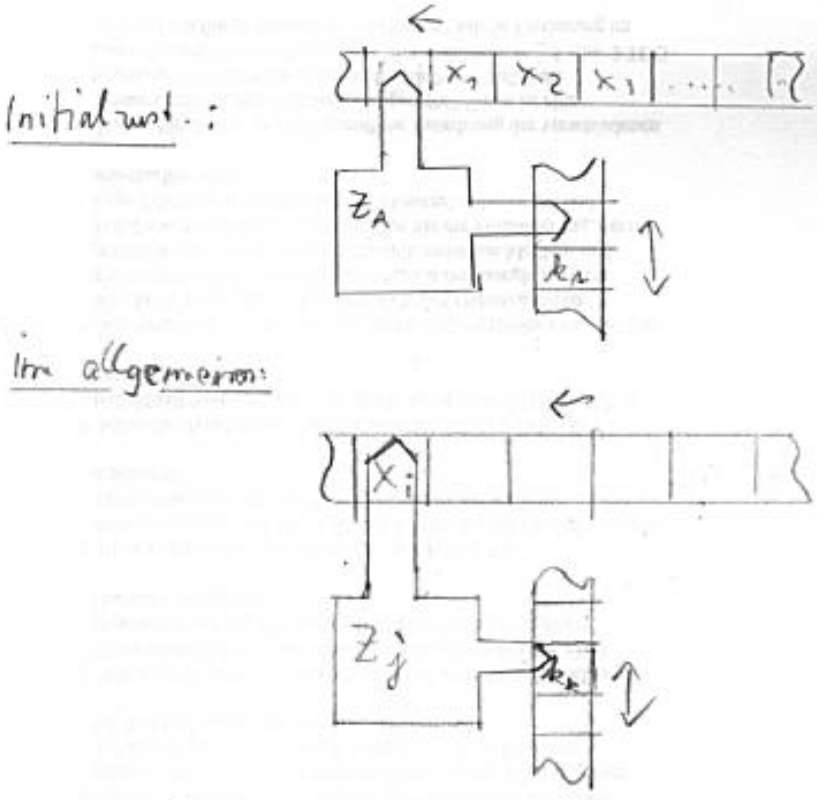
- X Alphabetmenge $\{x_1,...,x_n\},\lambda \in X$
- Z Zustände $Z=\{Z_1,...,Z_n\}$
- K Kellerzeichen $K=\{k_0,...,k_r\}$, $\lambda \in K$
- Einer Übergangsfunktion $f::(X \cup\{\lambda\})xZxK \rightarrow ZxK^*$

  $(x_i,Z_j,k_k) \rightarrow (Z_l,k)$
  Z_l = „neuer Zustand"; $k \in K^*$ (Wort aus Kellerzeichen)

  Auch möglich: (λ,Z_j,k_k) ⇨
  - Keine Berücksichtigung des Eingabezeichens unter dem Lesekopf
  - Keine Bewegung des Eingabebandes

- Anfangszustand Z_A
- Kellerstartzeichen $k_A \in K$
- $F \leq Z$ Endzustände

Arbeitsweise:



5 Möglichkeiten:

1) ist (x_i, Z_j, k_1) bzw (λ, Z_j,k_1) nichtr als Übergangsfunktion definiert ⇨ Maschine bleibt stehen!
2) Die Zusweisung (X_i,Z_j,k_1) ist definiert ⇨ (Z_j,k) ($k \neq \lambda$) ⇨ KA geht in Zustand Z_1 und k_k wird durch Wort ersetzt (PUSH)
   Eingabeband wird ein Feld nach links bewegt
   - SL-Kopf steht über einem Zeichen von X

3) Für die Zuordnung (λ,Z_j,k_a) →(Z,k) ($k \neq \lambda$) verhält sich Ka wie in 2) aber ohne Bewegung des Eingabebandes.

4) Gibt es eine Zuordnung (X_i,Z_j,k_k) → (Z_l,λ) ⇨
   - KA geht in Zustand Z_l
   - K_l wird mit λ ersetzt (POP)
   - S/L-Kopf bewegt sich um 1 Feld nach oben

5) Ist $Z_p \in$ F aus f erreicht oder das Speicherband leer, dann bleibt der KA stehen.

Sprache L(KA) eines Kellerautomaten

Ein Eingabewert w gehört zu L(KA), wenn die Anfangsgegebenheiten (s.u.) erfüllt sind, und die Verarbeitung erfolgt und die Endbedingungen (s.u.) erreicht werden.

Anfangsgegebenheiten:

- KA ist in Zustand Z_A
- Der S/L-Kopf steht über dem untersten Kellerfeld (k_A) des ansonsten leeren Speicherbandes
- Der L-Kopf befindet sich über dem ersten Zeichen von w

Endbedingungen:

- Der L-Kopf steht hinter dem letzten Zeichen von w
- Das Speicherband ist leer
- KA hat einen Endzustand $Z_E \in$ F erreicht.

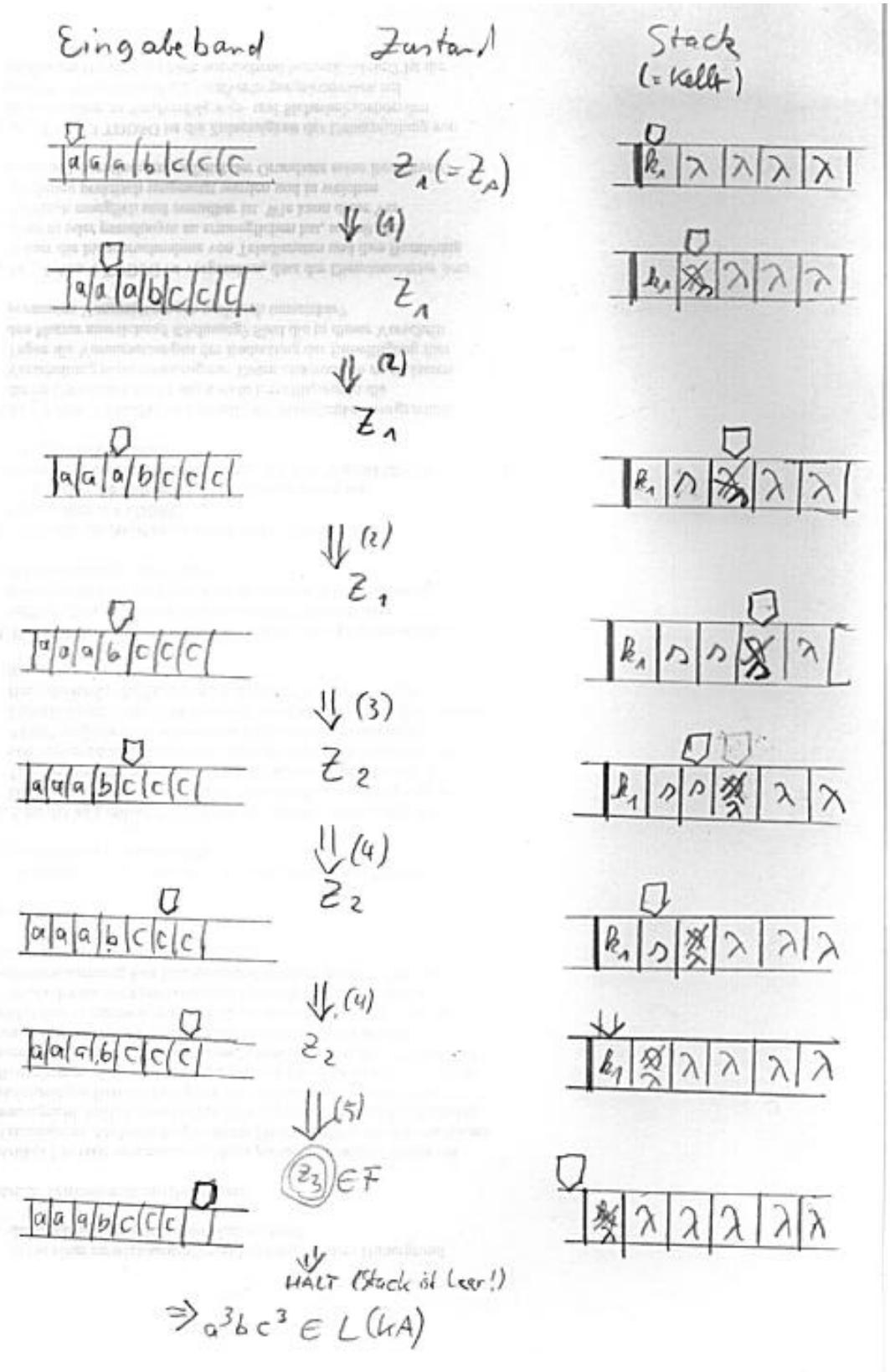
Beispiel: KA: X={a,b,c} ; Z={Z_A,Z_1,Z_2,Z_2}; K = {k_0,k_1,s,λ}, F={Z_3}

f: (a,Z_1,k_1) → (Z_1,k_1,s) (1)
(a,Z_1,s) → (Z_1,ss) (2)
(b,Z_1,s) → (Z_2,λ) (3)
(c,Z_2,s) → (Z_2,λ) (4)

$(c, Z_2, k_1) \rightarrow (Z_3, \lambda)$ (5)

$L(KA)=\{a^n bc^n | n \geq 1\}$

z.B. $w=a^3bc^3$



Mit Grammatik:

<s> ::= a<s‘>c
<s‘>::=a<s‘>c|b

(n ≥1)

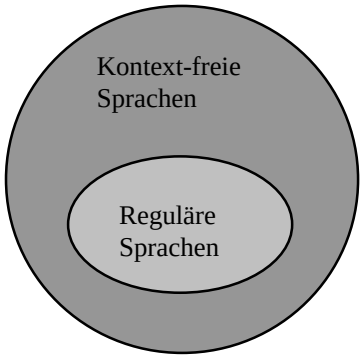
Definition: Eine Grammatik hießt kontextfrei, wenn alle Produktionen der Form a ::= b sind.
Dabei ist a ein nicht-terminales Symbol und b eine beliebige Zeichenkette, bestehend aus terminalen Und nicht-terminalen Symbolen.

Intuitiv: Ein nicht-terminales Symbol lässt sich ohne Berücksichtigung seiner Nachbarn ersetzen.

Beispiel: <s> → a<s‘>c → aa<s‘>cc → aaa<s‘<ccc → aaabccc

Satz: Eine Sprache L wird von einer kontextfreien Grammatik erzeugt ⇔ es gibt einen Kellerautomaten KA mit L(KA)=L

Bemerkung: Die Klasse regulärer Sprachen bildet eine *Unterklasse* der kontext-freien Sprachen!



Übung:

Es ist die Übergangsfunktion eines Kellerautomaten gesucht, die bei X={0,1,m},Z={Z_A,Z_1,Z_3}, K={k_A,0,1,λ} F={Z_2} die Worte der Sprache L(KA)={wm w | m ∈ w} akzeptiert, wobei w die Umkehrung von w ist.
Gib auch die zugehörige Grammatik an.
Ist diese Grammatik kontextfrei?

4.2.Beispiel für KA: Compilierung von Pascalausdrücken

4.2.1.

Quellcode → COMPILER → Maschinencode (Übersetzer)

- Erfolgt in zwei Stufen:

  1.Analyse:
  - Lexikalische Analyse (erkennt Wörter der Sprache)
  - Syntaxanalyse (Parser – erkennt grammatikalische Struktur)

  2.Synthese: Codeerzeugung (Semantik)

4.2.2. Syntax von arithmetischen Ausdrücken

Scanner liest Text und analysiert Wörter (Tokens) und legt alle Bezeichner in einer Symboltabelle ab.

4.2. Eigenschaften kontextfreier Grammatiken

- Sind $L_1(G_1)$ und $L_2(G_2)$ (G_1,G_2 kontextfrei) kontextfrei, so sind ebensu

  (1) $L_1 \cup L_2$
  (2) $L_1 * L_2$
  (3) L_1^*

  Kontextfreie Sprachen!

Übung: Beweis dafür

Hinweis: ohne Verlust der Allgemeinheit kann man annehmen, dass die nichtterminalen Symbole In beiden Grammatiken unterschiedlich sind (sonst umbenennen!)

- Es gibt Algorithmen, die die folgenden Fragen beantworten:

  a) G kontextfreie Grammatik, w Zeichenkette, ist $w \in L(G)$?
  b) Ist $L(G)=\{\}$?

- Satz über „Aufpumpen“ (*pumping up*): G sei eine kontextfreie Grammatik, dann existiert eine ganze Zahl k=k(G), so daß für jedes $w \in L(G)$ mit einer Länge > k gilt:

  W = uvxyz

Wobei

(1) $v \neq \lambda$ oder $y \neq \lambda$ (oder beide $\neq \lambda$)

(2) $uv^nxy^nz \in L(G)$ für alle $n \geq 0$

Beweis: siehe Hopcraft und Ullmann (Buch)

Anwendung: $L=\{a^nb^nc^n|n\geq 0\}$ ist nicht kontextfrei!

Beweis: Angenommen L sei kontextfrei für eine Grammatik G
K sei wie im Satz, n >(k/3), Zeichenkette $w=a^nb^nc^n$ hat Länge >k

Man schreibe $w=a^nb^nc^n = uvxyz$ $v\neq\lambda$ | $y\neq\lambda$ |v und $y \neq \lambda$
⇨ $uv^2xy^2z \in L(G)$

Fall 1: v oder y enthalten 2 Symbole aus {a,b,c}=X ⇨ uv^2xy^2z enthält mindestens ein b vor a oder c
Vor a ⇨ unmöglich!
⇨ L ist nicht kontextfrei! (da G nicht kontextfrei!)

Beispiel:$\{a^p|p$ primzahl$\}$ ist nicht kontextfrei

Übung: kontextfreie Sprachen sind unter Durchschnitt und Komplementierung nicht abgeschlossen.
Hinweis: $L_1=\{a^nb^nc^m|n,m\geq 0\}$ ist kontextfrei (Beweisen) → 2. Solche Sprache nehmen und den Durchschnitt bilden.

Kapitel V: Theorie der Berechenbarkeit

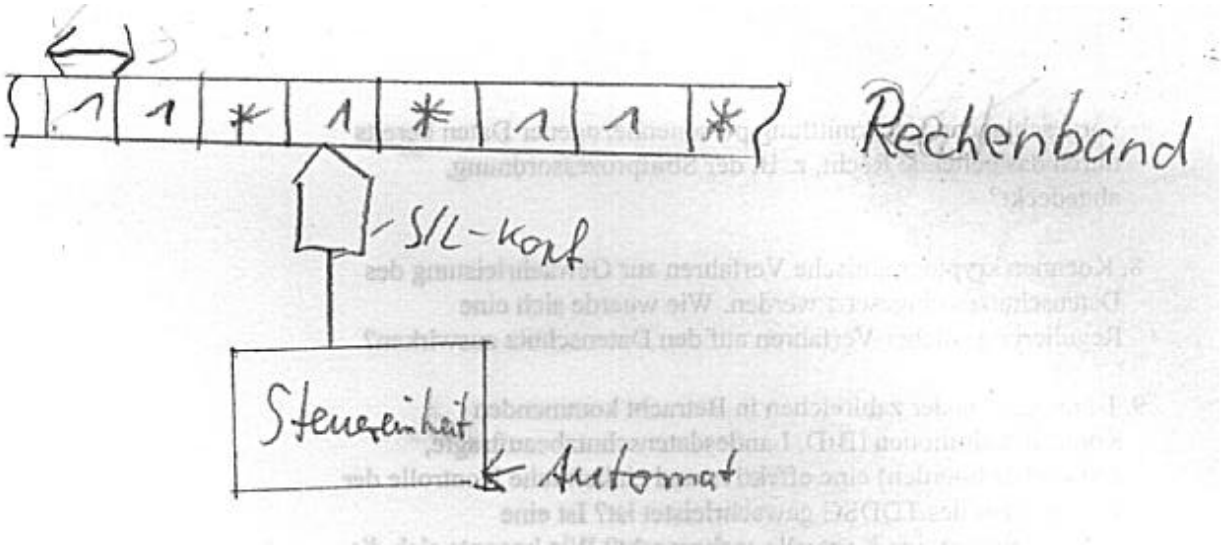
5.1. Algorithmen

Ziel: Algorithmen als (Turing-)Maschinen zu modellieren
Algorithmus: eine endliche Folge von Maschinenanweisungen zur Lösung eines Problems.

Methode: Turingmaschinen (nach Alan Turing, 1930)

5.2. Turing-Maschinen

5.2.1. Aufbau und Definition:

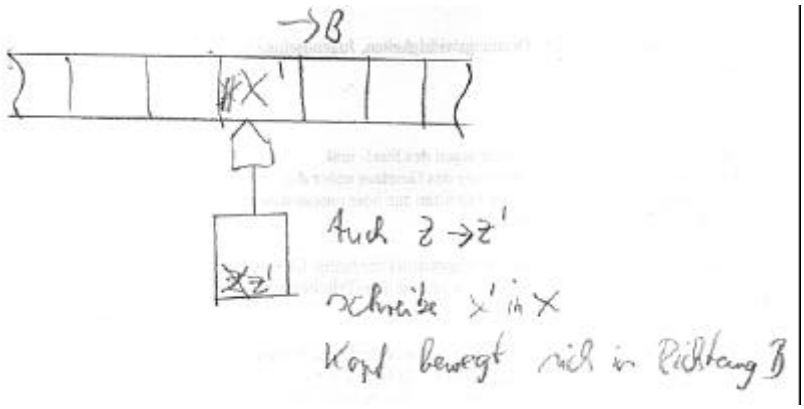


Definition: $T=(X,Z,Z_A,f,g,h,k)$ heißt Turing-Maschine:

(1) X endliche Menge von Bandzeichen (‚*' Extrazeichen z.B. Leerzeichen)
(2) Z Zustandsmenge
(3) Z_A Anfangszustand
(4) $F \leq Z$ Menge der Endzustände
(5) $g : X \times Z \rightarrow X$ Ausgabefunktion
(6) $h : X \times Z \rightarrow Z$ Übergangsfunktion
(7) $k : X \times Z \rightarrow \{L,R,0\}$ Kopfbewegungen (S/L-Kopf)

Übergang wird so geschrieben:

(X,Z,X',Z',B (Richtung)) X' = g(X,Z) Z'=h(X,Z) B=k(X,Z)



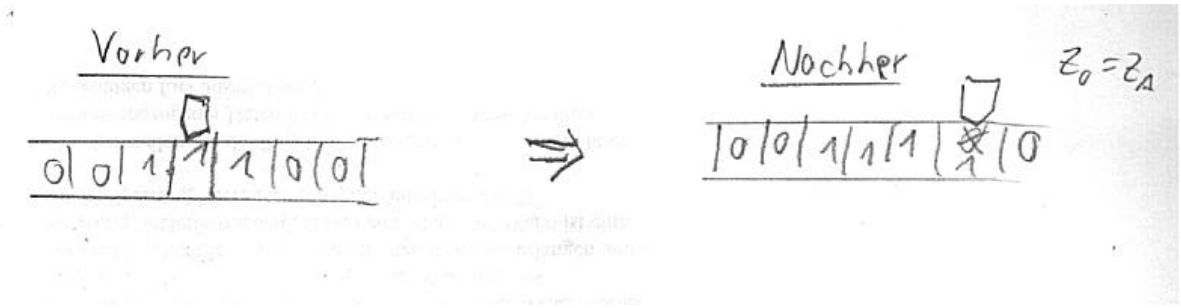
Übung: Grammatik für $L=\{a^n b^n c^n | n \geq 1\}$

TM: $T=(X,Z,Z_A,E,f,g,k)$

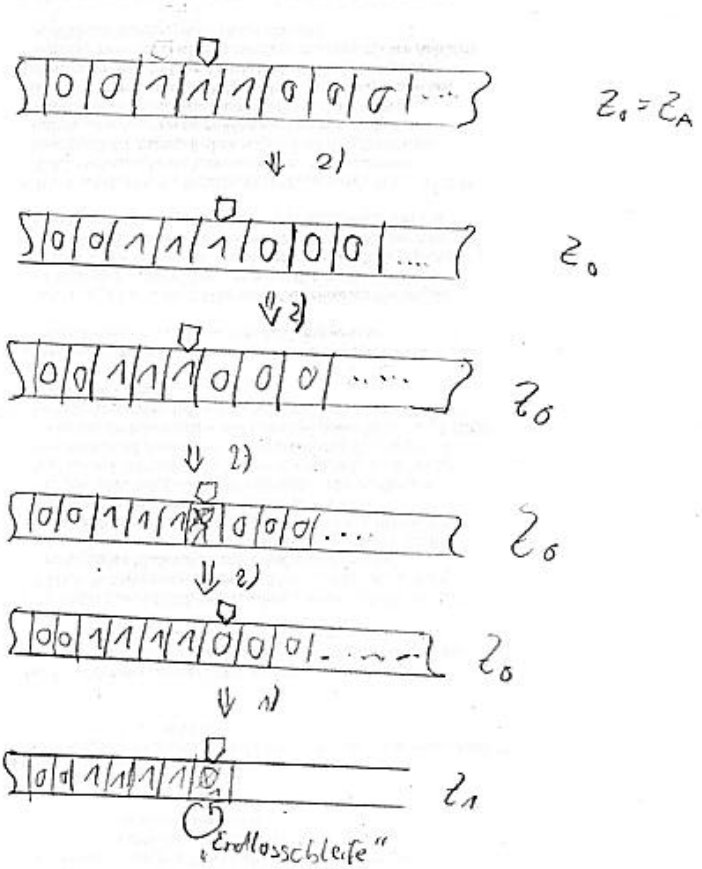
Beispiel: $X=\{0,1\}$, $Z=\{Z_A,Z_1\}$, $F=\{Z_1\}$

Aufgabe:

Baue eine TM, deren S/L-Kopf auf einem Feld mit einer 1 stehend, nach rechts das erste Feld mit einer 0 sucht, in dieses eine 1 schriebt und darauf stehenbleibt.



Übergangsfunktion Ausgabefunktion	Überführungsfunktion	Kopfbewegung
1) $(0,Z_0) \rightarrow 1$	$(0,Z_0) \rightarrow Z_1$	$(0,Z_0) \rightarrow 0$
2) $(1,Z_0) \rightarrow 1$	$(1,Z_0) \rightarrow Z_0$	$(1,Z_0) \rightarrow R$
3) $(0,Z_1) \rightarrow 0$	$(0,Z_1) \rightarrow Z_1$	$(0,Z_1) \rightarrow 0$
4) $(1,Z_1) \rightarrow 1$	$(1,Z_1) \rightarrow Z_1$	$(1,Z_1) \rightarrow 0$



Unendliche Schleife kann vermieden werden, wenn 3) und 4) entfallen.

1') $(0,Z_0,1,Z_1,0)$
2') $(1,Z_0,1,Z_0,R)$
} Umschreibungen von 1) und 2)

Definition: $T=(X,Z,Z_A,F,f,g,k)$ sei eine TM

T erkennt (oder akzeptiert) $s \in X^*$ wenn T sich am Anfang in folgender initialer Konfiguration befindet:

1) Zustand Z_1
2) Zeichenkette s steht auf dem Band geschrieben
3) S/L-Kopf befindet sich über dem letztenb Zeichen von s

Und nach endlich vielen Schritten hält die Maschine in einem Endzustand $Z_E \in F$ an.

T erkennt $s \in X^*$ falls T s für alle $s \in S$ erkennt.

Bemerkung:

1) $s \in X^*$ wird nicht von T erkannt kann bedeuten:

a) Die TM hält im Zustand $Z \in F$ an , oder
b) Die TM hält überhaupt nicht an.

2) Die Funktion f ist meistens nicht definiert für $Z_E \in F$ ⇨ wenn Z_E erreicht, hält die TM automatisch an.

Beispiel: TM die alle Zeichenketten, die aus einer geraden Anzahl von Einsen bestehen, erkennt, soll Gebaut werden.

$Z = \{Z_A, Z_1, Z_2\}$ $F=\{Z_2\}$ $X=[*,1\}$

5.2.2. TM-Programme

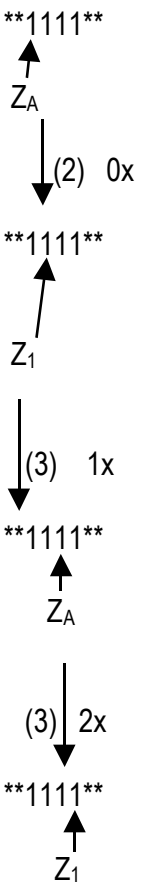
Arbeitsweise einer TM wird durch die Übergangsfunktionen f,g,k beschrieben.
Problem: etwas umständlich

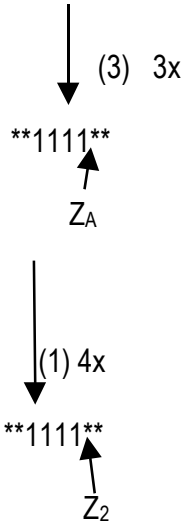
Programmiersprachen für TM vereinfachen die Arbeitsweisen, insbesondere die Formulierung der Algorithmen.

- Bandalphabet besteht aus {|,*}. Keine Einschränkung!
- Programmiersprache besteht aus 7 Anweisungen:

(1) schreib *
(2) schreib |
(3) Kopf links
(4) Kopf rechts
(5) Wenn * gehe zu i (i-te Anweisung ⇨ bedingter Sprung!)
(6) Wenn | gehe zu k (k-te Anweisung ⇨ bedingter Sprung)
(7) Halt

1) (*,Z_A,*,Z_2,0) Band ist leer oder keine Einsen (mehr) vorhanden
2) (1,Z_A,1,Z_1,R) Ungerade viele Einsen gelesen
3) (1,Z_1,1,Z_A,R) Gerade viele Einsen gelesen





Halt

Beispiel: Addierer, der zwei natürliche Zahlen (in unärer Darstellung) addiert. Es soll z.B. aus der Bandinschrift ***||*|||** die Inschrift ***|||||** erzeugt werden. Der S/L-Kopf soll zu Beginn und am Ende auf dem am weitesten rechts stehenden | des beschrifteten Bandstücks stehen.

Lösungsidee: Das die Zahlen(-darstellung) trennende Leerzeichen (*) wird durch ein | ersetzt und einer der Summanden um ein | gekürzt.

Algorithmus: 1. Suche das linke Ende des rechten Summanden
2. Überschreibe das links folgende * mit |

⇨ Turing-Programm:

1. Kopf links	***||*|||***
2. wenn | gehe zu 1	***||*|||***
3. schreibe |	***|||||***
4. Kopf rechts	***|||||***
5. wenn | gehe zu 4	***|||||***
6. Kopf links	***|||||***
7. schreibe *	***||||*****
8. Kopf links	***||||*****
9, halt	

Aufgabe:Man gebe ein Turingprogramm an, dass eine einfache Kopiermaschine realisiert. Die Maschine erzeugt rechts eine Gruppe von | und eine durch * davon getrennte Kopie dieser Gruppe.

Vorher: ...**||||***... Nachher: ---**||||*||||***...

Eine Turing-Maschine mit Programm P berechnet die n-stellige Funktion f, wenn die Maschine auf der Bandinschrift **|||*||||*...*||||*** (= x_1 x_2 .. x_n) mit den Argumenten x_1,...,x_n (in unärer Darstellung) nach endlich vielen (Programm)Schritten anhält und als Bandinschrift in folgender Weise ausgibt:

||...|... (wobei ||...| die unäre Darstellung des Funktionswertes ist).

Dabei steht der S/L-Kopf auf dem rechten „Ende“ des Funktionswertes ⇨ rechts von ihm befinden sich nur Leerzeichen.
Eine Funktion f heißt Turing-berechenbar , wenn es eine Touringmaschinde bzw ein Turingprogramm gibt, das f berechnet.

Aufgabe: Man gebe ein Turing-Programm für eome TM an, die die Fumktion f(x)=2x berechnen kann.
Auf dem Band steht vorher x in unärer Darstellung, danach soll rechts daneben der Wert von f(x) stehen. Der S/L-Kopf steht am rechten Ende von f(x).

5.2.3. rekursive Funktionen

Die Funktion f ist rekursiv und folgendermaßen definiert:

$$f(x) = \begin{cases} f(x-1)+3 & x \geq 1 \\ 2 & x=0 \end{cases}$$

z.B.:
f(0)=2
f(1)=5
f(2)=8
f(3)=11

```
int f(int x)
{
  if (x==0) return 2;
  else
    return f(x-1)+3;
}
```

Aufgabe: Gib ein TM-Programm an,das f(x) berechnet.

Das Turing-Programm beitet direkt die Arbeitsweise zur Berechnug rekrusiver Funktionen.
Das Argument x bildet am Anfang die Bandinschrift

|||||

am Schluss, falls das TM-Programm korrekt gearbeitet hat, besteht die Bandinschrift aus 17 aufeinanderfolgenden | .

Programm:

(1) Kopf rechts **|||||** (Beginn x?5)

(2) Kopf rechts **||||****
(3) schreibe | **|||||*|**
(4) Kopf rechts **|||||*||**
(5) schreibe | **|||||*||*
(6) Kopf links **|||||*||*
(7) wenn | gehe zu 6 **|||||*||*
(8) Kopf links **|||||*||*
(9) wenn * gehe zu 19 [**|||||***...] Endbedingung!
(10) schreibe * **|||**||*
(11) Kopf rechts **|||**||*
(12) schreibe | **||||*|||*
(13) Kopf rechts **||||*|||*
(14) wenn | gehe zu 13 **||||*|||**
(15) schreibe | **||||*||||*
(16) Kopf rechts **||||*||||**
(17) schreibe | **||||*|||||*
(18) wenn | gehe zu 6 **||||*|||||*
(19) Kopf rechts **||||*|||||*
(20) wenn | gehe zu 19 **||||*|||||*
(21) Kopf links **||||*|||||*
(22) halt

Satz: Jede rekursive Funktion ist Turing-berechenbar! Jede Turing-berechenbare Funktion *kann* als eine rekursive Funktion realisiert werden.

Chomsky Sprachenhierarchie

Eine Grammatik des Typs 0 ist ein 4-Tupel
$G=(X,X_N,<S>,P)$

1) X Alphabet
2) X_N Menge von Nichtterminalsystemen ($X_N \neq \{\}$)
3) $<S> \in X_N$ Startsymbol
4) P endliche Menge von Produktionen P der Form p::=c (BNF) (p,c Terminal- bzw. Nichtterminalsymbole – p enthält mindestens ein Element aus X_N)

Nullbare Konvention: alle Produktionen a,b mit $b=\lambda$ sind der Form $<S<::=\lambda$!

Definition:

a) G Typ-0-Grammatik ⇨ L(G) heißt Typ-0-Sprache
b) G heißt Typ-1-Grammatik, wenn
   1) die Nullbare Konventi0on gilt
   2) für Produktionen a::=b ist die Länge von b $\geq$ Länge von a

   L(G) heißt dann Typ-1-Sprache

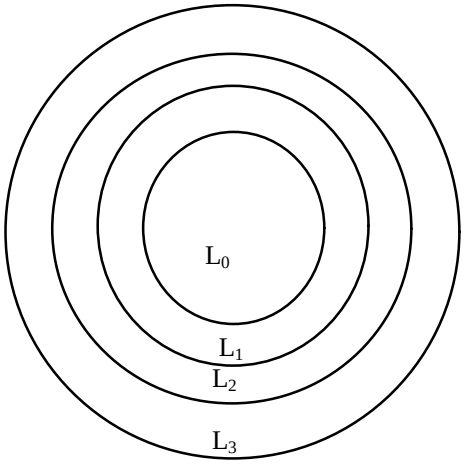
c) G heißt Typ-2-Grammatik, wenn
   1) die Nullbare Konvention gilt
   2) Für Produktionenn a,b ist die Länge von a=1 (=kontextfreie Grammatik!)
   L(G) heißt dann Typ-2-Sprache ode kontextfreie Sprache

d) G heißt Typ-3-Grammatik, oder regulär, wenn
   1) die Nullbare Konvention gilt
   2) Die Produktion a,b sind rechts- oder linkslinear.
      L(G) heißt dann Typ-3-Sprache oder reguläre Sprache.

Satz (v.Clomsky): L_i = Klasse der Typ-i-Sprachen (i=0,1,2,3)

Dann gilt:

$L_3 \subset L_2 \subset L_1 \subset L_0$



Satz des Aufpumpens für reguläre Grammatiken

Sei L=L(G) reguläre Sprache. Dann gibt es int n > 6 mit Z $\in$ L(G) mit Länge (Z) >n
⇨ Z=uvw v$\neq\lambda$, Länge(uv) $\leq$ n und uvw $\in$ L für alle i $\geq$ 0

Übung:

Beweisen Sie, daß L={$a^n b^n$|n$\geq$1} nicht regulär ist.

5.3. Das Halteproblem

Es gibt Turing-Programme, bei denen die Maschinen nach endlich vielen Schritten anhält
⇨ Endlosschleife (sonst!)

Halteproblem für TM

Gibt es einen Algorithmus, der zu jedem Turing-Programm und jeder Inschrift entscheidet , ob die Maschine nach endlich vielen Schrittern anhält?

Diesen Algorithmus gibt es nicht!

Beweis:

1) Konstruktion einer TM, die Programme und Bandinschriften anderer Turing-Programme verarbeiten Kann („universelle TM“).

   a) Codierung von Programmanweisungen

Anweisung	Codewort
Schreibe *	***
Schreibe |	**|
Kopf links	*|*
Kopf rechts	*||
halt	|**
Wenn * gehe zu	|*|**...* (i Sternchen)
Wenn | gehe zu	||*|....| (i mal |)

   b) sequentielles Aufzeichnen der Codewörter

      Programmanfang: |
      Programmende: |||

      Beispiel:

      (1) Kopf links
      (2) Schreibe *
      (3) Kopf links
      (4) Wenn | gehe zu (3)
      (5) Halt

      ⇨ sequentielle Codierung ⇨ ***|*|*****|*||*|||*|**|||***

2) Selbstanwendbarkeit:

   Ein Turing-Programm P heißt selbstanwendbar, wenn es auf Code(P) (dem Turing-Programm P zugeordnete Zeichenkette) nach endlich vielen Schritten hält.

   Reduktion H ⇨ P auf Selbstanwendbarkeit

   Gibt es ein Turing-Programm, das zu einem beliebig gegbenen Turingprogrgramm entscheidet, ob dieses sebstanwendbar ist oder nicht?

Annahme: Es gibt ein solches Programm S, d.h. S druckt ein | auf das Arbeitsfeld, falls P (ein beliebiges Turing-Programm) auf Code(P) hält, sonst nur * und bleibt dann stehen.

Konstruktion eines neuen Programmes S* :
Die letzte Anweisung in S((n) halt) wird ersetzt durch
(n) wenn | gehe zu (n+2)
(n+1) wenn * gehe zu (n+4)
(n+2) Kopf rechts
(n+3) wenn * gehe zu (n+2)
(n+4) halt

Falls P selbstanwendbar ist, läuft der S/L-Kopf nach rechts weg.

Ist P dagegen nicht selbstanwendbar, so hält S* an.

Ist S* selbstanwendbar?

a) Annahme: S* Selbstanwendbar, S* hält auf Code(S*) ⇨ S druckt | ⇨ S* hält nicht an.
⇨ S* hält auch nicht nach (S*)

⇨ S* kann es nich tgeben! ⇨ Das Halteproblem für TM ist unlösbar!

Referenz: M.Harrison, W. Ruzzo, I.Ollman: Protection in Operating Systems (ACM 1918)
Ausgabe 1976: S461-470