Fachhochschule München
Fachbereich Informatik

Rechnertechnik WS2000/2001

Dr. Thomas Tensi

Inhaltsverzeichnis

1 Einführung **8**

1.1 Digitale und analoge Signalverarbeitung 8

1.1.1 Signale 8

1.1.2 Vergleich zwischen analoger und digitaler Technik . . . 9

1.1.3 Realisierungsvarianten digitaler Systeme 11

1.1.4 Entwicklung der digitalen Schaltungstechnik 13

1.2 Zahlensysteme 14

1.2.1 Allgemeines 14

1.2.2 Konvertierung zwischen Zahlensystemen 16

1.2.3 Rechnen im Dualsystem 20

1.2.4 Dualbrüche 28

2 Logische Grundfunktionen **30**

2.1 Grundbegriffe 30

2.2 Schaltalgebra 31

2.2.1 Allgemeines 31

2.2.2 Rechenregeln der Schaltalgebra 33

2.2.3 Darstellung von elementaren Schaltfunktionen mittels NAND- oder NOR-Gattern 37

2.2.4 Minimierung logischer Funktionen 38

3 Komplexere digitale Funktionen **49**

3.1 Elementare Schaltnetze 49

3.1.1 Kodewandler 49
3.1.2 Multiplexer und Demultiplexer 53
3.1.3 Addierer 57
3.1.4 Komparatoren 60
3.2 Elementare Schaltwerke 61
3.2.1 Digitale Oszillatoren 62
3.2.2 Monostabile Kippstufen (Monoflops) 65
3.2.3 Bistabile Kippstufen (Flipflops) 67
3.2.4 Digitale Zähler 74
3.2.5 Schieberegister 83

4 Entwurf von Schaltungen mit endlichen Automaten 86
4.1 Rekapitulation der Automatentheorie 86
4.1.1 Allgemeines 86
4.1.2 Zustandsminimierung 90
4.2 Analyse und Synthese von Schaltwerken mit Automaten . . . 93
4.2.1 Analyse 93

5 Technische Realisierung digitaler Schaltungen 101
5.1 Grundlagen 101
5.1.1 Kenngrößen digitaler integrierter Schaltungen 101
5.1.2 Grundlagen zu Halbleitern 106
5.1.3 Schaltkreistechnologien 111
5.2 Digitale Halbleiterspeicher 122
5.2.1 Festwertspeicher 122
5.2.2 Schreib-/Lesespeicher (RAM) 127
5.2.3 Assoziativspeicher 133
5.2.4 Zusammenfassung Speicher 135
5.3 Anwenderspezifische ICs 135
5.3.1 Programmierbare Logikanordnungen (PLD) 136
5.3.2 Gate Arrays 140

5.3.3 Standardzellenschaltkreise 143

6 Mikroprozessoren 144

6.1 Grundlagen 144

6.1.1 Allgemeines 144

6.1.2 Aufbau eines Mikrocomputers 145

6.1.3 Aufbau eines Mikroprozessors 146

6.2 CISC- und RISC-Prozessoren 148

6.2.1 Ein 32Bit-CISC-Prozessor 149

6.2.2 Ein 32Bit-RISC-Prozessor 167

6.2.3 Vergleich zwischen CISC und RISC bzgl. Programmierung 175

6.2.4 Einfache CISC- und RISC-Programme 176

6.3 Prozessoren für höhere Programmiersprachen 182

6.3.1 Abrißder Programmiersprache JAVA 183

6.3.2 Die virtuelle Maschine für Java (JVM) 184

6.3.3 Der picoJava-Prozessor 197

Information

Das vorliegende Dokument ist Grundlage meiner Vorlesung "Rechnertechnik" im WS2000/2001 und SS2001 an der FH München. Da es sich um eine Arbeitsunterlage handelt, enthält das Papier möglicherweise Fehler und in jedem Fall sind Diagramme enthalten, die aus im Literaturverzeichnis zitierten Büchern stammen. Vor einer weiteren Verwendung des Materials muß daher geprüft werden, ob Rechte Dritter damit verletzt werden.

München im September 2000

Thomas Tensi

Kapitel 1

Einführung

1.1 Digitale und analoge Signalverarbeitung

1.1.1 Signale

Definition 1.1 (Signal)
Ein *Signal* ist eine zeitlich veränderliche physikalische Größe, die eine auf sie abgebildete Information trägt. Die physikalische Größe heißt *Informationsparameter*. Sowohl der Informationsparameter als auch der zeitliche Signalverlauf können *kontinuierlich* oder *diskret* sein.

Beispiel 1.2 (Informationsparameter)
Informationsparameter in elektronischen Geräten können sein: Spannung, Frequenz, Strom, Impulsbreite...

Erklärung 1.3 (Signalkategorien)
Signale lassen sich in folgende Kategorien einteilen (Bild 1.1):

1. amplitudenkontinuierlich, zeitkontinuierlich
2. amplitudenkontinuierlich, zeitdiskret
3. amplitudendiskret, zeitkontinuierlich
4. amplitudendiskret, zeitdiskret

Definition 1.4 (Analoge Signale)
Ein analoges Signal ist amplitudenkontinuierlich, d.h. der Informationsparameter kann (innerhalb bestimmter Grenzen) beliebig viele Werte annehmen.

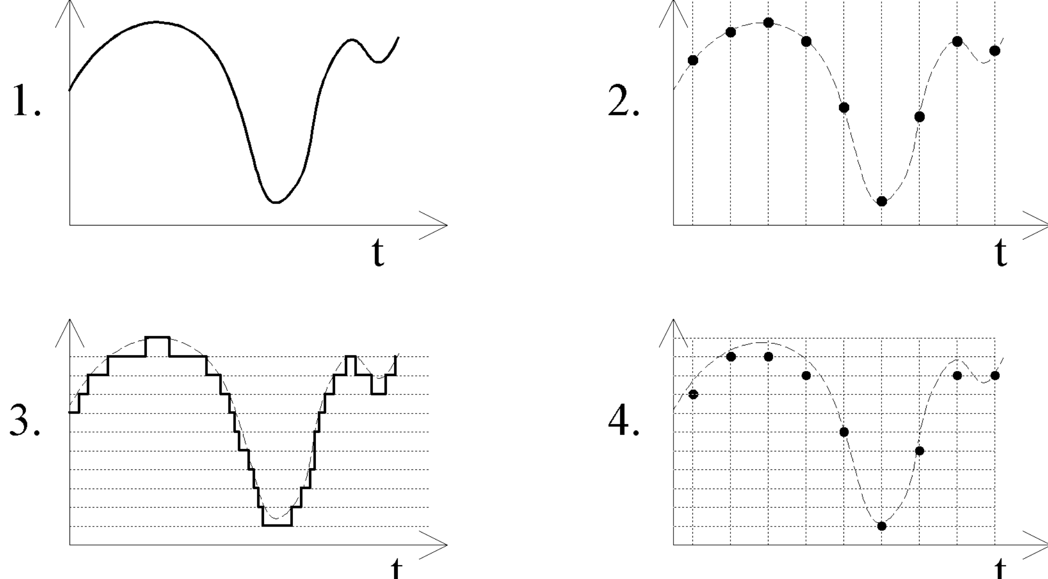


Abbildung 1.1: Arten von Signalen

Die Auflösung ist also theoretisch unendlich hoch. Ein analoges Signal kann zeitkontinuierlich oder zeitdiskret sein.

Definition 1.5 (Digitale Signale)
Ein digitales Signal ist amplitudendiskret und in der Regel zeitdiskret. In der digitalen Schaltungstechnik werden fast ausschließlich zweiwertige Signale verwendet (mit hohem und niedrigen Pegel (High, Low)).

Folgende Gründe sprechen für die Verwendung von binären Signalen:

- großer Störabstand
- größere Toleranzen in den Pegeln der repräsentierenden physikalischen Größe möglich
- leicht realisierbar
- wenige Grundschaltungen nötig

Definition 1.6 (Parallele vs. serielle Darstellung des Informationsparameters)

parallel Die Werte eines n-bit Informationsparameters (mit 2^n möglichen Werten) werden über n Kanäle übertragen.
Vorteile: sofort kompletter Wert, zeitkontinuierlich, störunempfindlicher

seriell Die Werte eines n-bit Informationsparameters werden über 1 Kanal zu n Zeitpunkten aufeinanderfolgend übertragen.
Vorteile: geringerer technischer Aufwand (?!)

1.1.2 Vergleich zwischen analoger und digitaler Technik

Vorteile der Analogtechnik sind

- billig (z.B. Realisierung einer Digitalschaltung mit diskreten Elementen wesentlich aufwendiger als entsprechende Analogschaltung)
- oft einfacher
- für gleichwertige Übertragung eines Analogsignal erheblich geringere Anforderungen an Kanal als bei Digitalsignal (s.u.)
- zeit- und amplitudenkontinuierliche Verarbeitung (?!)
- oft keine Umsetzung der zu verarbeitenden Größe nötig
- robuster bei Überschreitung des spezifizierten Pegelbereichs

Vorteile der Digitaltechnik sind

- bei Einsatz hochintegrierte Schaltungen ebenfalls billig
- beliebige Genauigkeit möglich
- störsicher, zuverlässig
- Signale speicherbar

Anmerkung: Es ist ebenfalls aufschlußreich, zu vergleichen, welche Bandbreite analoge und digitale Signale bei einer Übertragung über einen Kanal benötigen. Dabei wird angenommen, daß das analoge Signal als obere Grenzfrequenz f_{grenz} hat.

Verfahren: Das amplituden- und zeitkontinuierliche analoge Signal wird wie folgt in ein amplituden- und zeitdiskretes digitales Signal umgesetzt und das digitale statt des analogen übertragen (Bild 1.2).

1. Filterung des Analogsignals mit Tiefpaßfilter mit Grenzfrequenz f_{grenz} (Bandbreitenbegrenzung);
2. Abtastung (Sampling) des Analogsignals mit Abtastfrequenz f_s, d.h. Abstand zwischen Abtastungen ist $t_s = 1/f_s$; nach dem Abtasttheorem muß gelten: $f_s \geq 2f_{\text{grenz}}$; (in der Praxis: $f_s \gg f_{\text{grenz}}$)
3. Quantisierung des abgetasteten Signals in 2^n Stufen, d.h. in n-bit-Digitalwert
4. Übertragung des digitalen Signals

notwendige Kanalbandbreite bei Analogsignal : f_{grenz}

notwendige Kanalbandbreite bei Digitalsignal : Die notwendige Bandbreite für Übertragung einer Impulsfolge von R bit/s ist R/2 Hz (Informationstheorie). Die Bitrate aufgrund obiger Abtastung und Quantisierung ist $nf_s \geq 2nf_{\text{grenz}}$.
$\Longrightarrow$ Bandbreite des Kanals$\geq (2nf_{\text{grenz}})/2 = nf_{\text{grenz}}$

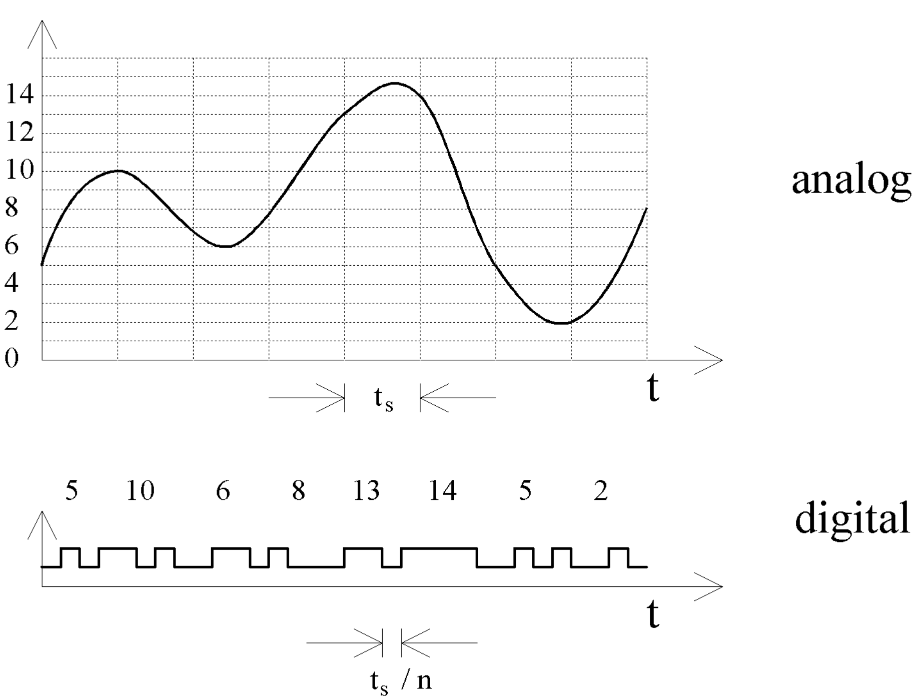


Abbildung 1.2: Analog-Digitalwandlung eines Signals

Anmerkung: Bei einer Genauigkeit der Abtastung von 1 Promille Genauigkeit ($2^10 = 1024$ Abtastwerte) ergibt sich ungefähr die 10fache Bandbreite bei einer reinen Digitalübertragung!

Anmerkung: Weitere Betrachtung: Übertragung von Digitalsignalen über Analogleitungen
Bild 1.3 zeigt am Beispiel der Übertragung des Buchstaben b (= 01100010 im ASCII-Code), wie das Signal bei beschränkter Bandbreite verfälscht wird. Grundlage der Betrachtung: Fourieranalyse[1]Also: Bei reiner Digitalübertragung mit Rate r (in Bits/s) gilt T=8/r bzw. f=r/8. Wenn man diese Signale über Telephonverbindungen (f_{grenz}=3000Hz) überträgt, dann wird für 300bps bis zu 80f übertragen, bei 9600bps bis zu 2f. Im letzten Fall wird das Signal nicht mehr realistisch erkannt.

1.1.3 Realisierungsvarianten digitaler Systeme

Varianten der Verarbeitung:

- verdrahtungsprogrammiert / speicherprogrammiert

  verdrahtungsprogrammiert: Funktion des Systems wird durch feste Verdrahtung einzelner Schaltkreise realisiert; Vorteile: einfache und billigere Bauteile; möglicherweise schneller

  speicherprogrammiert: Funktion des Systems wird durch Festwertspeicher / PLA / Mikroprozessor realisiert; Vorteile: einfach umkonfigurierbar; geringe Bauteilzahl; komplexe Abläufe einfacher realisierbar; geringer Entwicklungsaufwand

  Anmerkung: Ein ROM ist ein Beispiel für eine speicherprogrammierte Schaltung. Für jede Eingangskombination wird der Ausgangswert "eingebrannt" (siehe Kapitel 5.2.1).

- synchrone / asynchrone Systeme

[1]Fourier hat gezeigt, daß jede stetige Funktion g mit Periodenlänge T sich zerlegen läßt in Sinus- und Cosinusfunktionen mit

$$g(t) = c/2 + \sum_k a_k \sin(kf2\pi t) + \sum_k b_k \cos(kf2\pi t), f := 1/T$$

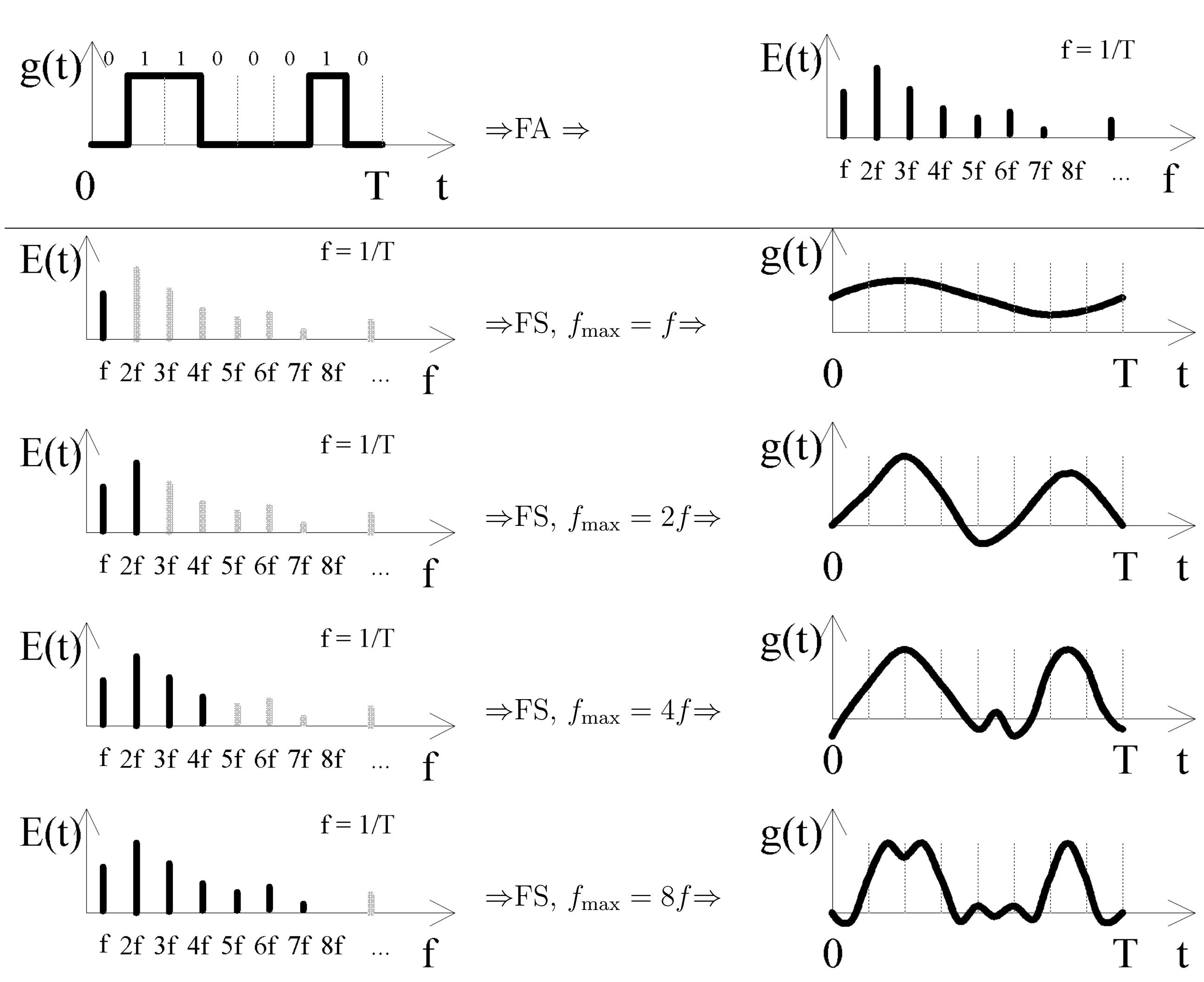
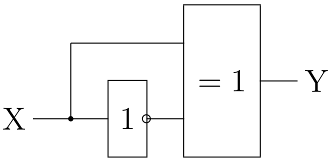


Abbildung 1.3: Signalverfälschung bei Bandbreitenbegrenzung

synchron, getaktet : Diese Systeme werden durch einen Taktgeber gesteuert. Die Information wird nur zu bestimmten Zeitpunkten weitergeleitet. Das System hat eine durch die Taktrate definierte Verarbeitungsgeschwindigkeit. Nachteil: etwas höherer Schaltungsaufwand, höhere Verarbeitungszeit

asynchron, ungetaktet : In diesen Systemen ist jedes Funktionselement zu jedem Zeitpunkt aktiv. Nachteil: Störungen im dynamischen Verhalten (z.B. Hazards)

Erklärung 1.7 (Beispiel eines Hazards)
In folgender Schaltung (Notation siehe Kapitel 2) wird bei jedem Signalwechsel von X am Ausgang Y aufgrund der Schaltzeit der Negation ein kurzer Pegelwechsel nach 0 erzeugt (0-Hazard), obwohl der Ausgang der Antivalenz logisch immer auf 1 liegen müßte.



- Schaltnetz / Schaltwerk

  Schaltnetz: Der Wert der Ausgangsgrößen ist nur vom aktuellen Wert der Eingangsgrößen abhängig (Gatterkaskade). Das System realisiert eine *digitale Funktion*.

  Schaltwerk: Der Wert der Ausgangsgrößen hängt neben den aktuellen Werten der Eingangsgrößen auch von deren früheren Werten zu endlich vielen vorhergehenden Zeitpunkten ab. Das System realisiert eine *digitale Funktion mit Internzustand*.
  Schaltwerke sind fast immer synchron.

- kundenspezifisch / halbkundenspezifisch

  kundenspezifisch: Ein System, deren Bauteile speziell für eine Anwendung gefertigt werden (LSI/VLSI).
  Vorteil: extrem an Anforderungen angepaßt; geringe Produktionskosten pro Stück (?!); hoher Schutz vor Nachbauten

  halbkundenspezifisch: Ein System, deren Bauteile nur teilweise für eine Anwendung gefertigt werden. Vorteil: erheblich geringerer Entwicklungsaufwand

Standardschaltung: Ein System, deren Bauteile überhaupt nicht für eine Anwendung gefertigt werden.
Vorteil: Standardelemente; ohne Fertigung einsetzbar

- parallel / seriell
  dies wurde bereits in Definition 1.6 behandelt...

1.1.4 Entwicklung der digitalen Schaltungstechnik

Zeit	Technik	Integrationsgrad (Gatter/Chip)	Produkt
1940–1960	diskret	—	Röhre/Relais/Transistoren
1960–1965	SSI	3–30	Gatter, Flipflops...
1966–1970	MSI	30–300	Zähler, Addierer...
1970–1980	LSI	300–3000	Mikroprozessor, RAM, ROM
1980–	VLSI	-10^5	kompl. Mikroproz., Kundenschaltungen
1990–	ULSI	$> 10^5$	— -—

Tabelle 1.1: Steigerung des Integrationsgrads

Bauelement	Ausführung	Bauelemente/cm^3
Röhre/Relais	Chassis, diskret	$10^{-3} \dots 10^{-1}$
Transistor	Leiterkarten, diskret	$10^{-1} \dots 10$
Einfache IS	Leiterkarten, Verbindung auf Chip	$10 \dots 10^5$
Großintegr. IS	— -—	$10^2 \dots 10^7$

Tabelle 1.2: Generationen von Bauelementen und Packungsdichte

1.2 Zahlensysteme

1.2.1 Allgemeines

Definition 1.8 (Zahlensystem)
Ein *Zahlensystem* ist ein Regelsystem, in dem Zahlen durch Folgen von Symbolen dargestellt werden. Die Regeln legen fest,

- welche Symbolfolgen als Zahlendarstellungen erlaubt sind und
- wie die Zahl aus der Symbolfolge bestimmt wird.

Anmerkung: Praktisch alle gebräuchlichen Zahlensysteme sind *Stellenwertsysteme*.

Definition 1.9 (Stellenwertsystem)
In einem Stellenwertsystem stellt jedes Symbol einen Zahlenwert dar. Zusätzlich hat jede Stelle in der Symbolfolge eine Gewichtung, die nicht alle gleich 1 sind. Der gesamte Zahlenwert ergibt sich, indem man die Stellengewichtungen multipliziert mit den Symbolwerten und alle diese Produkte addiert.

Mathematisch gilt (für Zahlendarstellungen ohne Komma)

$$\text{Wert} = s_k \cdot g_k + s_{k-1} \cdot g_{k-1} + \cdots + s_0 \cdot g_0 = \sum_{i=0}^{k} s_i \cdot g_i$$

Dabei ist

$$\begin{aligned} s_i &= \text{Wert des Symbols an der } (i+1)\text{-ten Stelle von rechts} \\ g_i &= \text{Gewichtung der } i+1\text{-ten Stelle von rechts} \end{aligned}$$

Definition 1.10 (Stellenwertsystem mit Basis)
Ein *Stellenwertsystem mit Basis* b ist ein Stellenwertsystem mit folgenden zusätzlichen Eigenschaften:

- Es gibt genau b Symbole mit den Symbolwerten $0, 1, \ldots, b-1$.
- Die Gewichtung an der ersten Stelle von rechts ist 1.
- Die Gewichtung einer Stelle ist um den Faktor b höher als die der rechts benachbarten (also: i-te Stelle von rechts hat Gewichtung b^{i-1}).

Mathematisch ergibt sich

$$\text{Wert} = s_{k-1} \cdot b^{k-1} + s_{k-2} \cdot b^{k-2} + \cdots + s_1 \cdot b + s_0 \cdot 1 = \sum_{i=0}^{k-1} s_i \cdot b^i$$

mit

$$s_i = \text{Wert des Symbols an der } i+1\text{-ten Stelle von rechts}$$

Eine Zahlendarstellung zur Basis b wird durch die tiefgestellte Dezimaldarstellung von b abgeschlossen, wenn die Basis nicht aus dem Zusammenhang erschlossen werden kann (z.B. 1101_2).

Anmerkung: Üblicherweise werden als Symbole für die Zahlendarstellungen werden die Ziffern "0" bis "9" und die Buchstaben "A" bis "Z" genommen. Die Symbolwerte sind: Wert("0")=0_{10},..., Wert("9")=9_{10}, Wert("A")=10_{10}, Wert("B")=11_{10} usw.

Beispiel 1.11 (Stellenwerte)

1. Der Wert von "749_{10}" $= 7 \cdot 10^2 + 4 \cdot 10^1 + 9 \cdot 10^0 = 749$.
2. Der Wert von "217_8" $= 2 \cdot 8^2 + 1 \cdot 8^1 + 7 \cdot 8^0 = 143_{10}$.

In der technischen Praxis wichtige Stellenwertsysteme:

- Dualsystem: Basis 2, Symbole aus $\{0, 1\}$ bzw. $\{O, L\}$;
- Oktalsystem: Basis 8, Symbole aus $\{0, 1, \ldots, 7\}$;
- Dezimalsystem: Basis 10, Symbole aus $\{0, 1, \ldots, 9\}$;
- Hexadezimalsystem: Basis 16, Symbole aus $\{0, 1, \ldots, 9, A, \ldots, F\}$;

1.2.2 Konvertierung zwischen Zahlensystemen

Definition 1.12 (Divisionsverfahren zur Zahlenkonversion)
Die Konvertierung einer Zahl z aus System X in System Y (mit Basis b) kann wie folgt passieren:

```
verbleibender_Wert:=z;
Ergebnisfolge:="";
WIEDERHOLE
    Quotient:= verbleibender_Wert DIV b;
    Divisionsrest:=verbleibender_Wert MOD b;
    verbleibender_Wert:=Quotient;
    Ergebnisfolge:=Symbol_Y(Divisionsrest) verkettet_mit Ergebnisfolge;
SOLANGE BIS verbleibender_Wert=0;
```

Dabei wird im System X gerechnet; die Divisionsreste werden im System Y dargestellt.

Beispiel 1.13 (Divisionsverfahren I)
$2110_{10} = x_{16}$?:

verbleibender_Wert	Quotient	Divisionsrest	Symbol_{16}	Ergebnis	fertig?
2110	131	14	E	"E"	nein
131	8	3	3	"3E"	nein
8	0	8	8	"83E"	ja

d.h. das Ergebnis ist $2110_{10} = 83E_{16}$.

Beispiel 1.14 (Divisionsverfahren II)
$26_{10} = x_2?$:

verbleibender_Wert	Quotient	Divisionsrest	Symbol_2	Ergebnis	fertig?
26	13	0	O	"O"	nein
13	6	1	L	"LO"	nein
6	3	0	O	"OLO"	nein
3	1	1	L	"LOLO"	nein
1	0	1	L	"LLOLO"	ja

d.h. das Ergebnis ist $26_{10} = LLOLO_2$.

Erklärung 1.15 (Warum funktioniert das Divisionsverfahren?)
Beweis für die Korrektheit des Verfahrens:
Induktion nach n ($n =$Länge der Darstellung von z)
Induktionsbehauptung: Wenn $0 \leq v \leq b-1$ und $u \geq 0$, dann gilt:

$$\text{Darstellung}_Y(u \cdot b + v) = \text{Darstellung}_Y(u) \text{ verkettet_mit } \text{Symbol}_Y(v)$$

n=1: $z = 0 \cdot b + v$. Der Algorithmus terminiert nach einem Durchlauf mit der Darstellung der Ziffer v mit $0 \leq v \leq b-1$ im neuen System Y.

n$\Longrightarrow$n+1: $z = u \cdot b + v$. Der Algorithmus spaltet durch Division mit b die Ziffer v im neuen System Y ab (mit $0 \leq v \leq b-1$). Da die Darstellung von u kürzer ist als die von z, wird die Darstellung von u nach Induktionsannahme korrekt bestimmt. Konkatenation der Darstellungen von u und v ergibt die Darstellung für z.

Definition 1.16 (Multiplikationsverfahren zur Zahlenkonversion)
Die Konvertierung einer Zahl z aus System X (mit Basis b) in System Y kann wie folgt passieren:

```
verbleibende_Eingabefolge:=z;
Ergebniswert:=0;
WIEDERHOLE SOLANGE verbleibende_Eingabefolge nicht leer
    erste_Ziffer:=Erstes_Symbol_von(verbleibende_Eingabefolge);
    Ziffernwert:=Wert_von(erste_Ziffer);
    verbleibende_Eingabefolge:=Rest_von (verbleibende_Eingabefolge);
    Ergebniswert:=Ergebniswert·b + Ziffernwert;
ENDE WIEDERHOLE;
```

Dabei wird vollständig im System Y gerechnet.

Beispiel 1.17 (Multiplikationsverfahren I)
$D31_{16} = x_{10}$?:

verbl._Eingabe	fertig?	erste_Ziffer	Ziffernwert	Ergebniswert
"D31"	nein	'D'	13	13
"31"	nein	'3'	3	$13 \cdot 16 + 3 = 211$
"1"	nein	'1'	1	$(13 \cdot 16 + 3) \cdot 16 + 1$ $= 13 \cdot 16^2 + 3 \cdot 16 + 1$ $= 3377$
""	ja			

Beispiel 1.18 (Multiplikationsverfahren II)
$21202_3 = x_{10}$?:

verbl._Eingabe	fertig?	erste_Ziffer	Ziffernwert	Ergebniswert
"21202"	nein	'2'	2	2
"1202"	nein	'1'	1	$2 \cdot 3 + 1 = 7$
"202"	nein	'2'	2	$(2 \cdot 3 + 1) \cdot 3 + 2$ $= 2 \cdot 3^2 + 1 \cdot 3 + 2$ $= 23$
"02"	nein	'0'	0	69
"2"	nein	'2'	2	209
""	ja			

Erklärung 1.19 (Warum funktioniert das Multiplikationsverfahren?)
Das in der Definition des Stellenwertsystems mit Basis gegebene Polynom wird mittels des sog. *Hornerschemas* berechnet.
Idee:

$$\begin{aligned}\sum_{i=0}^{k-1} s_i \cdot b^i &= s_{k-1} \cdot b^{k-1} + s_{k-2} \cdot b^{k-2} + \cdots + s_1 \cdot b + s_0 \cdot 1 \\ &= (\ldots(s_{k-1} \cdot b + s_{k-2}) \cdot b + s_{k-3}) \cdot b \ldots + s_1) \cdot b + s_0\end{aligned}$$

Definition 1.20 (Umwandlung, wenn Zielbasis Potenz der Ausgangsbasis ist)
Wenn eine Zahlenumwandlung einer Zahl x von einer Ausgangsbasis b_a in eine Zielbasis b_z mit $b_z = {b_a}^j$ erfolgt, dann kann die Umwandlung dadurch passieren, daß man in der Darstellung von x mit Basis b_a von rechts beginnend jeweils j Stellen zusammenfaßt zu einer Stelle der Zieldarstellung.

Beispiel 1.21 (Basiskonversion)
$1011110_2 = x_8?$. Für die Basen gilt: $b_z = 8 = 2^3 = {b_a}^j$.

$$\begin{array}{ccc} \overbrace{001}^{j} & \overbrace{011}^{j} & \overbrace{110}^{j} \\ \Downarrow & \Downarrow & \Downarrow \\ 1 & 3 & 6 \end{array}$$

d.h. das Resultat ist 136_8.

Erklärung 1.22 (Warum funktioniert das Stellengruppierungsverfahren?)
Das in der Definition des Stellenwertsystems mit Basis gegebene Polynom wird anders geklammert und p benachbarte Koeffizienten werden zu einem neuen zusammengefaßt.
Idee: Wir nehmen vereinfacht an, daß $k = pj$ (ansonsten wird die Zahl im Ausgangssystem mit führenden Nullen ergänzt). Dann ergibt sich:

$$\begin{aligned}&\sum_{i=0}^{k-1} s_i \cdot b_a^i = \\ &= \sum_{m=0}^{p-1} \left(\sum_{n=0}^{j-1} s_{mj+n} \cdot b_a^{mj+n} \right) \\ &= \sum_{m=0}^{p-1} b_a^{mj} \cdot \left(\sum_{n=0}^{j-1} s_{mj+n} \cdot b_a^{n} \right) \\ &= \sum_{m=0}^{p-1} b_z^{m} \cdot \left(\underbrace{\sum_{n=0}^{j-1} s_{mj+n} \cdot b_a^{n}}_{s'_m} \right)\end{aligned}$$

$$= \sum_{m=0}^{p-1} s'_m \cdot b_z^m$$

Dabei sind s'_m die Ziffern bezüglich der neuen Basis b_z. Wie man leicht überlegt, gilt $0 \leq s'_i \leq {b_a}^j - 1$.

Anmerkung: Aufgrund der analogen Überlegung kann man auch Zahlen expandieren, wenn gilt: $b_z^j = b_a$, z.B.:
$8D5F_{16} = x_2$?. Für die Basen gilt: $b_z^j = 2^4 = 16 = b_a$.

$$\underbrace{8}_{4} \quad \underbrace{D}_{4} \quad \underbrace{5}_{4} \quad \underbrace{F}_{4}$$
$$\Downarrow \quad \Downarrow \quad \Downarrow \quad \Downarrow$$
$$1000 \quad 1101 \quad 0101 \quad 1111$$

d.h. das Resultat ist: 1000110101011111_2

1.2.3 Rechnen im Dualsystem

Duale Addition

Definition 1.23 (Standardverfahren für duale Addition)
Es wird stellenweise von rechts nach links addiert gemäß folgender Tabelle (der erste Übertrag ist 0):

1.Summand	2.Summand	alter_Übertrag	Resultat	neuer_Übertrag
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Beispiel 1.24 (Addition im Dualsystem)

1. Summand:	01101100
2. Summand:	01011010
Übertrag:	1111
Resultat:	11000110

Duale Subtraktion

Definition 1.25 (Standardverfahren für duale Subtraktion)
Es wird stellenweise von rechts nach links subtrahiert gemäß folgender Tabelle (der erste Borger ist 0):

Minuend	Subtrahend	alter_Borger	Resultat	neuer_Borger
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Beispiel 1.26 (Subtraktion im Dualsystem)

Minuend:	11110000
Subtrahend:	10011010
Borger:	00111100
Resultat:	01010110

Probleme für Prozessoren durch direkte Subtraktion:

- gesonderte Vorzeichenrechnung nötig
- spezielles Rechenwerk für Subtraktion notwendig

$\rightsquigarrow$Subtraktion auf Addition zurückführen

Erklärung 1.27 (Rechnen mit Komplementen)
Eine negative Zahl $(-b)$ wird beim Rechnen mit Komplementen durch $\bar{b} := (C - b)$ dargestellt (für eine festgelegte Zahl C).
Dann können Differenzen wie folgt bestimmt werden:

$$a - b = a + (C - b) - C = a + \bar{b} - C$$

(Das Abziehen von C und die Berechnung von $(C - b)$ muß sehr einfach erfolgen können.)

Beispiel 1.28 (Rechnen mit Komplement im Dezimalsystem)
Sei $C := 999$. Jeder Ziffer sei die zu 9 ergänzende zugeordnet (also z.B. die '3' der '6' und umgekehrt). Das Komplement einer Zahl ergibt sich, indem jede Ziffer durch die ergänzende ersetzt wird.
Für $365 - 179 =?$ ergibt sich

$$\begin{array}{rr l} & 365 & \\ + & 820 & \\ \hline & 1185 & \\ + & 1 & \text{sog. Einerrücklauf!} \\ \hline & 186 & \end{array}$$

Die Subtraktion von C wird durch eine Addition von 1 realisiert!

Definition 1.29 (Komplemente im Dualsystem)
Wenn mit N-stelligen Dualzahlen gerechnet wird, gibt es zwei Arten von Komplementrechnung:

Einerkomplementrechnung mit $C = 2^N - 1$: früher üblich;

Zweierkomplementrechnung mit $C = 2^N$: heute gebräuchlich

Definition 1.30 (Rechnen im Einerkomplement)
Beim Rechnen im Einerkomplement wird das Komplement einer Zahl dadurch gebildet, daß alle Bits "gekippt" werden (von 0 nach 1 und umgekehrt). Nach der Addition des Minuenden und des Subtrahendenkomplements wird 1 addiert, falls bei der Addition ein Überlauf auftrat (sogenannter *Einerrücklauf*). Ansonsten ist das Ergebnis negativ und dessen Komplement wird bestimmt.

Beispiel 1.31 (Einerkomplement I)
$13 - 7 =?$ im Dualsystem mit $N = 5$

$$\begin{array}{lrcrl} \text{Komplementbildung:} & 7_{10} & = & 00111 & \\ & -7_{10} & = & 11000 & \\ \text{Addition:} & 13_{10} & = & 01101 & \\ & +(-7_{10}) & = & 11000 & \\ \cline{4-4} & & & 100101 & \\ \text{Einerrücklauf:} & & & 1 & \\ \cline{4-4} & & & 00110 & = 6_{10} \\ \cline{4-5} \end{array}$$

Beispiel 1.32 (Einerkomplement II)
$7 - 13 =?$ im Dualsystem mit $N = 5$

Komplementbildung:	13_{10}	=	01101	
	-13_{10}	=	10010	
Addition:	7_{10}	=	00111	
	$+(-13_{10})$	=	10010	
			11001	
kein Einerrücklauf:				
			11001	$= -6_{10}$

Erklärung 1.33 (Analyse der Einerkomplementrechnung)
Bei der Subtraktion von $x-y$ sei $z := x-y$, das Resultat $r := x+(C-y) = C+z$ und im Einerkomplement $C := 2^N - 1$). Es können folgende Fälle auftreten:

$x > y$: Dann ist $r = C + z \geq C + 1 = 2^N$, d.h. bei der Addition tritt ein Überlauf ein. Für das Resultat der Addition r gilt $r = C + z = 2^N + (z-1)$. Die Korrektur um $-C$ wird durch Streichen des Überlaufs und Addition von 1 bewirkt ($z = r - 2^N + 1$).

$x \leq y$: Dann ist $r = C + z \leq C < 2^N$, d.h. bei der Addition tritt kein Überlauf ein. Nun gilt aber $r = C - (-z)$, d.h. r ist das (negative) Resultat von $x - y$ in Einerkomplementdarstellung.

Anmerkung: Wie man bei Subtraktion x-x sehen kann, gibt es im Einerkomplement zwei Darstellungen der 0!

Anmerkung: Der Zahlenbereich im Einerkomplement umfaßt bei N bits die Zahlen $-2^{N-1}+1 \ldots 2^{N-1}-1$.

Zahl	-3	-2	-1	-0	0	1	2	3
Darstellung	100	101	110	111	000	001	010	011

Anmerkung: Bewertung der Einerkomplementrechnung:

- Fallunterscheidung bei Addition nötig;
- doppelte Darstellung der Null;
- einfache Komplementbildung

Definition 1.34 (Rechnen im Zweierkomplement)
Beim Rechnen im Zweierkomplement wird das Komplement einer Zahl dadurch gebildet, daß alle Bits "gekippt" werden und das Resultat um 1 inkrementiert wird.
Nach der Addition des Minuenden und des Subtrahendenkomplements muß keine weitere Operation durchgeführt werden.

Beispiel 1.35 (Zweierkomplement I)
$13 - 7 =?$ im Dualsystem mit $N = 5$

Komplementbildung:	7_{10}	$=$	00111	
	-7_{10}	$=$	11001	
Addition:	13_{10}	$=$	01101	
	$+(-7_{10})$	$=$	11001	
			100110	
Überlauf streichen:			-100000	
			00110	$= 6_{10}$

Beispiel 1.36 (Zweierkomplement II)
$7 - 13 =?$ im Dualsystem mit $N = 5$

Komplementbildung:	13_{10}	$=$	01101	
	-13_{10}	$=$	10011	
Addition:	7_{10}	$=$	00111	
	$+(-13_{10})$	$=$	10011	
			11010	
kein Überlauf:				
			11010	$= -6_{10}$

Erklärung 1.37 (Analyse der Zweierkomplementrechnung)
Bei der Subtraktion von $x - y$ sei $z := x - y$, das Resultat $r := x + (C - y) = C + z$ und im Zweierkomplement $C := 2^N$). Es können folgende Fälle auftreten:

$x \geq y$: Dann ist $r = C + z \geq C$, d.h. bei der Addition tritt ein Überlauf ein. Für das Resultat der Addition r gilt $r = C + z = 2^N + z$). Die Korrektur um $-C$ wird durch Streichen des Überlaufs bewirkt ($z = r - 2^N$).

$x < y$: Dann ist $r = C + z < C$, d.h. bei der Addition tritt kein Überlauf ein. Nun gilt aber $r = C - (-z)$, d.h. r ist das (negative) Resultat von $x - y$ in Zweierkomplementdarstellung.

Anmerkung: Der Zahlenbereich im Zweierkomplement umfaßt bei N bits die Zahlen $-2^{N-1} \dots 2^{N-1} - 1$.

Zahl	-4	-3	-2	-1	0	1	2	3
Darstellung	100	101	110	111	000	001	010	011

Anmerkung: Bewertung der Zweierkomplementrechnung:

- keine Fallunterscheidung bei Addition nötig;
- keine doppelte Darstellung einer Zahl;
- etwas komplizierte Komplementbildung;
- asymmetrischer Zahlenbereich;

Duale Multiplikation

Definition 1.38 (Standardverfahren für duale Multiplikation)
Der Multiplikator wird stellenweise von rechts nach links abgearbeitet. Der Startwert des Ergebnisses ist 0. Wenn eine Stelle des Multiplikators 1 ist, dann wird der Multiplikand multipliziert mit dem Stellenwert der Multiplikatorstelle zum Ergebnis addiert.

Beispiel 1.39 (Standardverfahren für duale Multiplikation)
Die Multiplikation von $3 \cdot 5 = 15$ funktioniert dual wie folgt:

```
0011·   0101
------------
        0011
       00000
      001100
     0000000
    --------
    00001111
```

Erklärung 1.40 (Analyse der Standardmultiplikation)
Die Standardmultiplikation funktioniert durch Anwendung des Distributivgesetzes:

$$x \cdot y \quad =$$

$$
\begin{aligned}
&= x \cdot \sum_{i=0}^{k-1} y_{i+1} \cdot 2^i \\
&= \sum_{i=0}^{k-1} x \cdot y_{i+1} \cdot 2^i \\
&= \sum_{\substack{i=0 \\ y_{i+1}=1}}^{k-1} x \cdot 2^i
\end{aligned}
$$

Anmerkung: Das Standardverfahren erfordert eine getrennte Vorzeichenrechnung; Zahlen in Komplementdarstellung werden nicht korrekt verarbeitet.

Beispiel: $(-3) \cdot 5 = ?$

```
11101·     00101
----------------
           00101
          000000
         0010100
        00101000
       001010000
     -----------
      0010010001
```

$\rightsquigarrow$ Algorithmus, der automatisch Korrekturen für Zahlen in Komplementdarstellung vornimmt

Definition 1.41 (Booth-Algorithmus zur dualen Multiplikation)
Die Multiplikation zweier Dualzahlen x und y (beide in Zweierkomplementdarstellung) kann wie folgt erfolgen:

```
vorherige_Ziffer:=0;
Multiplikator:=y;
Ergebnis:=0;
Faktor:=1;
WIEDERHOLE SOLANGE Multiplikator≠0
    aktuelle_Ziffer:=Multiplikator mod 2;
    Multiplikator:=Multiplikator div 2;
    WENN vorherige_Ziffer≠ aktuelle_Ziffer DANN
        WENN aktuelle_Ziffer=1 DANN
            Ergebnis:=Ergebnis-(x*Faktor);
        SONST
            Ergebnis:=Ergebnis+(x*Faktor);
        ENDE WENN;
    ENDE WENN;
    vorherige_Ziffer:=aktuelle_Ziffer;
```

Faktor:=Faktor*2;
ENDE WIEDERHOLE;

Beispiel 1.42 (Booth-Verfahren)
Berechnung von $(-3) \cdot (-5) = 11101_2 \cdot 11011_2$ mit 5 Dualstellen:

v_Ziffer	Faktor	Multipl.	a_Ziffer	Aktion	Ergebnis	δ
0	1	11011	1	subtr.	0000000011	-x
1	2	1101	1	-	0000000011	
1	4	110	0	add.	0000000011 + 1111110100 = 1111110111	+4x
0	8	11	1	subtr.	1111110111 + 0000011000 = 0000001111	-8x
1	16	1	1	-	0000001111	

Erklärung 1.43 (Analyse des Booth-Verfahrens)
Wir beweisen, daß das Booth-Verfahren für alle Multiplikatoren der Länge n korrekt arbeitet.

$n = 1:$ Wenn $y_0 = 0$ ist, dann ändert das Verfahren den Wert des Ergebnisses nicht ab und läßt ihn 0. Wenn $y_0 = 1$ ist, d.h. der Wert von y gleich -1 ist, dann wird Ergebnis:=Ergebnis-x zu -x berechnet.

$n \rightsquigarrow n + 1:$ Der Produktwert für die letzten n Stellen von y $(y_{n-1}, \ldots, y_0)$ sei korrekt bestimmt. Es gibt zwei Möglichkeiten:

- Wenn die vorderste Stelle y_n mit der zweiten y_{n-1} übereinstimmt, dann sind beide 0 oder beide 1. Der Wert von y ist dann nicht von der vordersten Stelle abhängig, d.h. das Produkt war für das verkürzte y bereits korrekt bestimmt worden.
- Stimmt vorderste und zweite Stelle nicht überein, dann ist das bisherige berechnete Produkt um $2^{n+1} \cdot x$ falsch (je nach Vorzeichen von y). Dies wird durch den Algorithmus entsprechend korrigiert.

1.2.4 Dualbrüche

Definition 1.44 (Dualbrüche)
Ein echter Dualbruch mit n Stellen läßt sich darstellen als $0,y_{-1}\dots y_{-n}$. Er hat den Wert $\sum_{i=1}^{n} y_{-i}\cdot 2^{-i}$.

Umrechnung eines Dezimalbruchs in einen Dualbruch

Anmerkung: Die Umrechnung eines Dezimalbruchs in einen Dualbruch erfolgt analog zum Divisionsverfahren; diesmal findet eine Multiplikation mit 2 statt.

Beispiel 1.45 (Bruchkonversion I)
$0,625_{10} = x_2$?

$$\begin{array}{rlrl} 0{,}625 & \cdot 2 = & 1{,}25 & \Longrightarrow y_{-1}=1 \\ 0{,}25 & \cdot 2 = & 0{,}5 & \Longrightarrow y_{-2}=0 \\ 0{,}5 & \cdot 2 = & 1{,}0 & \Longrightarrow y_{-3}=1 \\ 0 & \cdot 2 = & 0 & \Longrightarrow y_{-4}=0, y_{-5}=0,\dots \end{array}$$

$\Longrightarrow 0,625_{10} = 0,101_2$

Beispiel 1.46 (Bruchkonversion II)
$0,1_{10} = x_2$?

$$\begin{array}{llll} 0{,}1 & \cdot 2 = & 0{,}2 & \Longrightarrow y_{-1}=0 \\ 0{,}2 & \cdot 2 = & 0{,}4 & \Longrightarrow y_{-2}=0 \\ 0{,}4 & \cdot 2 = & 0{,}8 & \Longrightarrow y_{-3}=0 \\ 0{,}8 & \cdot 2 = & 1{,}6 & \Longrightarrow y_{-4}=1 \\ 0{,}6 & \cdot 2 = & 1{,}2 & \Longrightarrow y_{-5}=1 \\ 0{,}2 & \cdot 2 = & 0{,}4 & \Longrightarrow y_{-6}=0 \\ \dots & & & \end{array}$$

$\Longrightarrow 0,1_{10} = 0,0\overline{0011}_2$, d.h. ein periodischer Dualbruch.

Anmerkung: Aufgrund der Tatsache, daß endliche Dezimalbrüche zu periodischen Dualbrüchen werden, sind Rechenoperationen, die im Dezimalsystem unproblematisch sind, bei endlicher Dualarithmetik unexakt, z.B. $\underbrace{0,1+\cdots 0,1}_{10\text{mal}} \neq 1$, wenn im Dualsystem gerechnet wird.

Erklärung 1.47 (Analyse der Dualbruchumrechnung)
Analog zum Divisionsverfahren über das Hornerschema
Idee:

$$\begin{aligned}\sum_{i=1}^{n} s_{-i} \cdot 2^{-i} &= s_{-n} \cdot \frac{1}{2^{-n}} + s_{1-n} \cdot \frac{1}{2^{1-n}} + \cdots + s_{-2} \cdot \frac{1}{2^2} + s_{-1}\frac{1}{2} \\ &= \frac{1}{2} \cdot (s_{-1} + \frac{1}{2} \cdot (s_{-2} + (\cdots \frac{1}{2}(s_{1-n} + \frac{1}{2} \cdot s_{-n}) \cdots)\end{aligned}$$

Kapitel 2

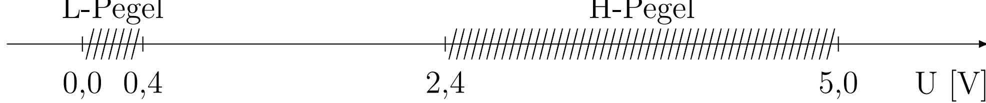
Logische Grundfunktionen

2.1 Grundbegriffe

Definition 2.1 (Logische Funktion, logischer Zustand, logischer Pegel)
Eine logische Funktion hat Parameter, die jeweils nur in einem von zwei logischen Zuständen sein können: 0 oder 1.
Diese Zustände werden technisch durch zwei nichtüberlappende Pegelbereiche einer physikalischen Größe repräsentiert. Der näher an $+\infty$ liegende Pegelbereich heißt H-Pegel, der näher an $-\infty$ liegende heißt L-Pegel. Der zwischen beiden liegende Pegelbereich heißt *verbotene Zone*.

Anmerkung: Es gibt nicht von vorneherein eine Zuordnung von logischem Pegel zu logischem Zustand.

Beispiel 2.2 (Pegelbereich bei TTL)
Bei TTL sieht der Pegelbereich wie folgt aus:



Definition 2.3 (positive Logikvereinbarung, negative Logikvereinbarung)
Bei der Zuordung von Pegel zu Zustand mit Zustand(H)=1 und Zustand(L)=0 spricht man von *positiver Logikvereinbarung* oder *positiver Logik*, bei Zuordnung mit Zustand(H)=0 und Zustand(L)=1 von *negativer Logikvereinbarung* oder *negativer Logik*.

Beispiel 2.4 (Einfluß der Vereinbarung auf die Schaltfunktion)
Die Wahrheitstabelle

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

muß bei positiver Logikvereinbarung implementiert werden durch

A	B	Y
L	L	H
L	H	H
H	L	H
H	H	L

bei negativer Logikvereinbarung implementiert werden durch

A	B	Y
H	H	L
H	L	L
L	H	L
L	L	H

d.h. durch unterschiedliche technische Schaltfunktionen.

Anmerkung: Die *Wahrheitstabelle* beschreibt das logische Verhalten einer Schaltung, die *Pegeltabelle* das Verhalten bezüglich Pegeln.

2.2 Schaltalgebra

2.2.1 Allgemeines

Definition 2.5 (Terme der Schaltalgebra)
Terme der Schaltalgebra beschreiben logische Funktionen zweiwertiger Variablen. Die Variablen werden durch Großbuchstaben repräsentiert, die durch Verknüpfungszeichen verbunden sind. Klammerungen sorgen für vorrangige Berechnung von Teilausdrücken.

Definition 2.6 (Verknüpfungszeichen)
In der Schaltalgebra sind folgende Verknüpfungszeichen zulässig (mit fallender Priorität:

Symbol	Bezeichnung	Stelligkeit	Beispiel
$\overline{\ }$, $\neg$	Negation	1	$\overline{X}$, $\neg$Y
$\wedge$	Konjunktion	2	X$\wedge$Y
$\vee$	Disjunktion	2	X$\vee$Y
$\overline{\wedge}$	NAND	2	$\overline{X \wedge Y}$
$\overline{\vee}$	NOR	2	$\overline{X \vee Y}$
$\rightarrow$	Implikation	2	X$\rightarrow$Y
$\leftrightarrow$	äquivalenz	2	X$\leftrightarrow$Y
$\not\leftrightarrow$	Antivalenz	2	X$\not\leftrightarrow$Y

Beispiel 2.7 (Terme der Schaltalgebra)
$X_1 \wedge \overline{X_3} \vee (X_1 \rightarrow X_4)$ oder $\overline{X} \wedge \overline{X_3} \vee (X_1 \rightarrow X_4)$

Definition 2.8 (Schaltsymbole (nach DIN 40900))

Negation

X —[1]o— Y

X	0	1
Y	1	0

$Y = \overline{X}$

Konjunktion

X_1, X_2 —[&]— Y

X_1	0	1	0	1
X_2	0	0	1	1
Y	0	0	0	1

$Y = X_1 \wedge X_2$

Disjunktion

X_1, X_2 —[≥ 1]— Y

X_1	0	1	0	1
X_2	0	0	1	1
Y	0	1	1	1

$Y = X_1 \vee X_2$

NAND

X_1, X_2 —[&]o— Y

X_1	0	1	0	1
X_2	0	0	1	1
Y	1	1	1	0

$Y = \overline{X_1 \wedge X_2}$

NOR

X_1, X_2 —[≥ 1]o— Y

X_1	0	1	0	1
X_2	0	0	1	1
Y	1	0	0	0

$Y = \overline{X_1 \vee X_2}$

Antivalenz

X_1, X_2 — $=1$ — Y

X_1	0	1	0	1
X_2	0	0	1	1
Y	0	1	1	0

$Y = X_1 \not\leftrightarrow X_2$

2.2.2 Rechenregeln der Schaltalgebra

Definition 2.9 (Axiomenschemata der Schaltalgebra)
Folgende Gleichungen sind für beliebige Einsetzungen von Termen für die Schemavariablen α, β und γ Theoreme der Schaltalgebra:

1. $\alpha \wedge 1 = \alpha;$ $\qquad \alpha \vee 0 = \alpha;$
2. $\alpha \wedge \beta = \beta \wedge \alpha;$ $\qquad \alpha \vee \beta = \beta \vee \alpha;$
3. $\alpha \wedge (\beta \wedge \gamma) = (\alpha \wedge \beta) \wedge \gamma;$ $\qquad \alpha \vee (\beta \vee \gamma) = (\alpha \vee \beta) \vee \gamma;$
4. $\alpha \wedge (\beta \vee \gamma) = (\alpha \wedge \beta) \vee (\alpha \wedge \gamma);$ $\qquad \alpha \vee (\beta \wedge \gamma) = (\alpha \vee \beta) \wedge (\alpha \vee \gamma);$
5. $\alpha \wedge \overline{\alpha} = 0;$ $\qquad \alpha \vee \overline{\alpha} = 1$ und $\overline{\alpha}$ mit dieser Eigenschaft ist eindeutig.

Definition 2.10 (Herleitungsregel der Schaltalgebra)
Wenn $\alpha = \beta$ ein Axiom oder Theorem der Schaltalgebra ist und $F(x) = G(x)$ ebenfalls, dann ist $F(\beta) = G(\alpha)$ ein Theorem.

Beispiel 2.11 (Herleitung von $x \wedge 0 = 0$)

$$\begin{aligned}
x \wedge 0 &= (x \wedge 0) \vee 0 && \text{(Axiom } 1b) : && \alpha = \alpha \vee 0 \\
&= (x \wedge 0) \vee (x \wedge \overline{x}) && \text{(Axiom } 5a) : && 0 = \alpha \wedge \overline{\alpha} \\
&= x \wedge (0 \vee \overline{x}) && \text{(Axiom } 4a) : && (\alpha \wedge \beta) \vee (\alpha \wedge \gamma) = \alpha \wedge (\beta \vee \gamma) \\
&= x \wedge (\overline{x} \vee 0) && \text{(Axiom } 2b) : && \alpha \vee \beta = \beta \vee \alpha \\
&= x \wedge \overline{x} && \text{(Axiom } 1b) : && \alpha \vee 0 = \alpha \\
&= 0 && \text{(Axiom } 5a) : && \alpha \wedge \overline{\alpha} = 0
\end{aligned}$$

Analog kann $x \vee 1 = 1$ gezeigt werden.

Beispiel 2.12 (Herleitung von $\overline{X \wedge Y} = \overline{X} \vee \overline{Y}$)

$$\begin{aligned}
0 &= 0 \vee 0 && \text{(Axiom } 1b) : && \alpha = \alpha \vee 0 \\
&= (x_2 \wedge 0) \vee (x_1 \wedge 0) && && 0 = \alpha \wedge 0 \\
&= (x_2 \wedge (x_1 \wedge \overline{x_1})) \vee (x_1 \wedge (x_2 \wedge \overline{x_2})) && \text{(Axiom } 5a) : && 0 = \alpha \wedge \overline{\alpha} \\
&= ((x_2 \wedge x_1) \wedge \overline{x_1}) \vee ((x_1 \wedge x_2) \wedge \overline{x_2}) && \text{(Axiom } 3a) : && \alpha \wedge (\beta \wedge \gamma) = (\alpha \wedge \beta) \wedge \gamma \\
&= ((x_1 \wedge x_2) \wedge \overline{x_1}) \vee ((x_1 \wedge x_2) \wedge \overline{x_2}) && \text{(Axiom } 2a) : && \alpha \wedge \beta = \beta \wedge \alpha \\
&= (x_1 \wedge x_2) \wedge (\overline{x_1} \vee \overline{x_2}) && \text{(Axiom } 4a) : && (\alpha \wedge \beta) \vee (\alpha \wedge \gamma) = \alpha \wedge (\beta \vee \gamma
\end{aligned}$$

$$
\begin{array}{rlll}
\text{Ebenso gilt: } 1 & = \overline{x_1} \vee 1 & & 1 = \alpha \vee 1 \\
& = \overline{x_1} \vee (x_2 \vee \overline{x_2}) & (\text{Axiom } 5b): & 1 = \alpha \vee \overline{\alpha} \\
& = (\overline{x_1} \vee x_2) \vee \overline{x_2} & (\text{Axiom } 3b): & \alpha \vee (\beta \vee \gamma) = (\alpha \vee \beta) \vee \gamma \\
& = ((\overline{x_1} \vee x_2) \wedge 1) \vee \overline{x_2} & (\text{Axiom } 1a): & \alpha = \alpha \wedge 1 \\
& = ((\overline{x_1} \vee x_2) \wedge (x_1 \vee \overline{x_1})) \vee \overline{x_2} & (\text{Axiom } 5b): & 1 = \alpha \vee \overline{\alpha} \\
& = ((\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee x_1)) \vee \overline{x_2} & (\text{Axiom } 2b): & \alpha \vee \beta = \beta \vee \alpha \\
& = (\overline{x_1} \vee (x_2 \wedge x_1)) \vee \overline{x_2} & (\text{Axiom } 4a): & (\alpha \vee \beta) \wedge (\alpha \vee \gamma) = \alpha \vee (\beta \wedge \gamma) \\
& = (\overline{x_1} \vee (x_1 \wedge x_2)) \vee \overline{x_2} & (\text{Axiom } 2a): & \alpha \wedge \beta = \beta \wedge \alpha \\
& = ((x_1 \wedge x_2) \vee \overline{x_1}) \vee \overline{x_2} & (\text{Axiom } 2b): & \alpha \vee \beta = \beta \vee \alpha \\
& = (x_1 \wedge x_2) \vee (\overline{x_1} \vee \overline{x_2}) & (\text{Axiom } 3b): & (\alpha \vee \beta) \vee \gamma = \alpha \vee (\beta \vee \gamma)
\end{array}
$$

Also gilt $\overline{x_1} \vee \overline{x_2} = \overline{x_1 \wedge x_2}$. Dieses Theorem heißt *De Morgan-Theorem.*

Anmerkung: Das Shannonsche Dualitätsprinzip besagt:

$$\overline{f(x_1, \ldots, x_n, \wedge, \vee)} = f(\overline{x_1}, \ldots, \overline{x_n}, \vee, \wedge)$$

Beweisskizze: Induktion über den Aufbau des Terms

Anmerkung: Beweise von $x_1 \vee (x_1 \wedge x_2) = x_1$, $x_1 \vee (\overline{x_1} \wedge x_2) = x_1 \vee x_2$ und $(x_1 \wedge x_2) \vee (x_1 \wedge \overline{x_2}) = x_1$;

$$x_1 \vee (x_1 \wedge x_2) \stackrel{4b}{=} (x_1 \wedge 1) \vee (x_1 \wedge x_2) \stackrel{2a}{=} x_1 \wedge (1 \vee x_2) \stackrel{4b}{=} x_1$$

$$x_1 \vee (\overline{x_1} \wedge x_2) \stackrel{3b}{=} (x_1 \vee \overline{x_1}) \wedge (x_1 \vee x_2) \stackrel{4b}{=} 1 \wedge (x_1 \vee x_2) = x_1 \vee x_2$$

$$(x_1 \wedge x_2) \vee (x_1 \wedge \overline{x_2}) \stackrel{3a}{=} x_1 \wedge (x_2 \vee \overline{x_2}) \stackrel{4b}{=} x_1 \wedge 1 = x_1$$

Exkurs: Formale Systeme, Modelltheorie

Anmerkung: **Ausgangsfrage:** Warum benutzt man nicht generell für Beweis von Theoremen eine Wahrheitstabelle?

Definition 2.13 (Formales System)
Ein formales System $\mathcal{F}$ besteht aus

- einer formalen Sprache S (das ist ein Alphabet Σ und ein Algorithmus, mit dem geprüft werden kann, daß ein Wort $x \in \Sigma^*$ zur Sprache S gehört);
- eine Menge A von *Axiomen*, d.h. $A \subseteq S$;

- einer Menge R von *Herleitungsregeln*, d.h. Verfahren, mit denen aus einer Folge $w_1, \ldots, w_k$ eine Folge $w_1, \ldots, w_k, w_{k+1}$ erzeugt werden kann (alle $w_i \in S_\mathcal{F}$); die Regeln müssen so aussehen, daß es überprüfbar ist, ob eine Folge durch Anwendung von Regel r_x in eine andere überführt wurde.

Anmerkung: Das oben definierte System für die Schaltalgebra ist ein formales System. Es gibt eine formale Sprache (zwei Terme mit "=" verbunden), Axiome und Herleitungsregeln.

Definition 2.14 (Beweis in einem formalen System)
Ein Beweis in einem formalen System $\mathcal{F}$ ist eine Folge von $w_1, \ldots, w_n$ mit $w_i \in S_\mathcal{F}$, sodaß

- $w_1 \in A_\mathcal{F}$, d.h. w_1 ist Axiom von $\mathcal{F}$;
- für $j > 1$:$w_j \in A_\mathcal{F}$ oder $w_1, \ldots, w_j$ entsteht aus $w_1, \ldots, w_{j-1}$ durch Anwendung einer Regel $r \in R_\mathcal{F}$.

Definition 2.15 (Beweis für X in einem formalen System)
Ein Beweis $B = w_1, \ldots, w_n$ in einem formalen System $\mathcal{F}$ ist ein Beweis von X, wenn $X = w_n$ ist. X heißt *gültig* oder *beweisbar* in $\mathcal{F}$.

Anmerkung: In formalen Systemen erzeugt man aus Folgen von Worten neue Folgen von Worten. Der Bezug zu "realen Dingen" ist nicht unmittelbar erkennbar. Die Herstellung des Bezugs passiert durch *Interpretationen*.

Definition 2.16 (Interpretation, wahr)
Eine *Interpretation für S in M* ist eine Abbildung I_M von Worten aus S in Objekte aus einer Menge M. Häufig ist diese Abbildung induktiv ("hierarchisch") definiert, wenn S induktiv definiert ist.
Ein Wort $w \in S_\mathcal{F}$ heißt *wahr für eine Interpretation* I_M, wenn $I_M(w)$ in M wahr gemäß einer Vereinbarung in M ist. M heißt dann *Modell von w*.
M heißt dann *Modell von* $\mathcal{F}$, wenn alle Axiome $A_\mathcal{F}$ und Herleitungsregeln für I_M wahr sind.

Beispiel 2.17 (Interpretation)
M sei die durch Wahrheitstabellen definierte Schaltalgebra für die Schaltsymbole. Die Interpretation ordne die Zeichen $\vee$, $\wedge$, $\bar{\ }$, = bestimmten Schaltsymbolen zu. Variablen werden als Eingangsvariablen beibehalten.
"Wahr in unserem System" soll definiert sein als: für jede Belegung der

Eingänge kommt 1 am Ausgang heraus.
$x_1 \vee (x_1 \wedge x_2) = x_1 \rightsquigarrow$ *Bild*
Dann ist die Formel in unserem System wahr.

Beispiel 2.18 (Mengenlehre als Modell)
M' sei die Mengenlehre. Jede Variable wird einer Menge zugeordnet, die Zeichen $\vee$, $\wedge$, $\bar{\ }$ und $=$ werden den Mengenoperationen Vereinigung, Schnitt, Komplementierung und Mengenvergleich zugeordnet. "Wahr in diesem System" soll definiert sein als "mengentheoretisch wahr". $x_1 \vee (x_1 \wedge x_2) = x_1 \rightsquigarrow$ *Bild*
Dann ist die Formel in zweiten System wahr.

Anmerkung: Es ist nicht klar, daß ein in $\mathcal{F}$ beweisbarer Satz auch notwendigerweise in einem Modell von $\mathcal{F}$ wahr ist.
Für die Schaltalgebra läßt sich beweisen, daß wenn M ein Modell der Schaltalgebra ist,

- jeder in der Schaltalgebra beweisbare Satz auch in M wahr ist (Korrektheit der Schaltalgebra), und
- jeder in M wahre Satz auch in der Schaltalgebra beweisbar ist (Vollständigkeit der Schaltalgebra).

Das gilt nicht im Allgemeinen! Häufig sind Kalküle unvollständig.

Anmerkung: Abgrenzung formaler Beweis/Prüfung der Wahrheit im Modell

> Vorteil des formalen Beweises: Theoreme gelten unbesehen in jedem Modell des formalen Systems; möglicherweise läßt sich die Wahrheit in einem Modell nicht überprüfen (Geometrie...), aber ein Beweis kann einfach geprüft werden;
>
> Vorteil der Prüfung im Modell: bei kleinen Modellen einfacher; Beweise sind oft schwierig zu erstellen;

Anmerkung: Warum benutzt man nicht generell für Beweis von Theoremen eine Wahrheitstabelle?
Antwort: Prinzipiell ist dieser Ansatz (Beweis durch Testen aller Kombinationen) in der Schaltalgebra möglich, weil die Zahl der zu prüfenden Fälle

oft klein ist.
Auch in der Schaltalgebra funktioniert der Ansatz nicht immer: 32bit-Multiplizierer mit 64 Eingängen hat = 2^{64} Eingangskombinationen $\approx 10_{19}$; Kontrollzeit per Computer$\approx 10_{13}s \approx 600.000a$).

2.2.3 Darstellung von elementaren Schaltfunktionen mittels NAND- oder NOR-Gattern

Anmerkung: Jede logische Funktion läßt sich ausschließlich unter Verwendung der NAND-Funktion bzw. der NOR-Funktion realisieren. Es genügt zu zeigen, daß mit diesen Funktionen die Grundfunktionen Negation, Disjunktion und Konjunktion realisierbar sind.

NAND: $\overline{A}: \; \overline{A} = \overline{A \wedge A};$

X ∨Y: $X \vee Y = \overline{\overline{X \vee Y}} = \overline{\overline{X} \wedge \overline{Y}};$

X ∧Y: $X \wedge Y = \overline{\overline{X \wedge Y}}$

NOR: $\overline{A}: \; \overline{A} = \overline{A \vee A}$

X ∨Y: $X \vee Y = \overline{\overline{X \vee Y}}$

X ∧Y: $X \wedge Y = \overline{\overline{X \wedge Y}} = \overline{\overline{X} \vee \overline{Y}};$

	Negation	Disjunktion	Konjunktion
NOR	A ≥ 1 $\overline{A}$	X, Y ≥ 1 ≥ 1 X∨Y	X ≥ 1, Y ≥ 1, ≥ 1 X∧Y
NAND	A & $\overline{A}$	X &, Y &, & X∧Y	X, Y & & X∨Y

2.2.4 Minimierung logischer Funktionen

Allgemeines

Definition 2.19 (Stufenzahl)
Eine Schaltung heißt n-stufig, wenn n die maximale Zahl der Gatter ist, die zwischen einem Eingang und einem Ausgang liegen. Dabei bleiben Negationen an Ein- oder Ausgang unberücksichtigt.

Anmerkung: Die in diesem Abschnitt beschriebene Minimierung erfolgt nach Stufenzahl, nicht nach Gatterzahl.

Definition 2.20 (Normalform einer logischen Gleichung)
Eine *Normalform einer logischen Gleichung* ist eine standardisierte Form einer logischen Gleichung, in der nur Negationen, Konjunktionen und Disjunktionen vorkommen. Bedeutsam sind

- disjunktive Normalform und
- konjunktive Normalform

Definition 2.21 (Minterm, Maxterm)
Ein *Minterm* ist eine konjunktive Verknüpfung aller Eingangsvariablen, wobei jede Variable in negierter oder nichtnegierter Form vorkommt.
Ein *Maxterm* ist eine disjunktive Verknüpfung aller Eingangsvariablen, wobei jede Variable in negierter oder nichtnegierter Form vorkommt.

Anmerkung:

1. Die Negation eines Minterms ist ein Maxterm und umgekehrt.
2. Ein Minterm wird genau für eine Belegung der Eingangsvariablen gleich 1, ein Maxterm genau für eine Belegung der Eingangsvariablen gleich 0.

Anmerkung: Ab jetzt wird als Schreibkonvention $x_i x_j$ für $x_i \wedge x_j$ verwendet, wenn dadurch keine Mehrdeutigkeit entsteht.

Beispiel 2.22 (Minterm, Maxterm)
Minterm $\overline{x_1}\,\overline{x_2}\,x_3$ wird 1 für $x_1 = 0, x_2 = 0, x_3 = 1$. Maxterm $\overline{x_1} \vee x_2 \vee x_3$ wird 0 für $x_1 = 1, x_2 = 0, x_3 = 0$.

Definition 2.23 (Disjunktive Normalform)
Eine disjunktive Normalform einer Funktion ist eine disjunktive Verknüpfung von Mintermen, eine konjunktive Normalform einer Funktion eine konjunktive Verknüpfung von Maxtermen.

Anmerkung: Die Normalformen sind — bis auf die Reihenfolge der Terme — jeweils eindeutig.

Definition 2.24 (Aufstellung der disjunktiven und konjunktiven Normalform)
Wenn eine Schaltfunktion durch eine Wahrheitstabelle gegeben ist, dann können die Normalformen wie folgt gewonnen werden:

DNF: Alle Minterme, für die die logische Funktion gleich 1 ist, werden durch Disjunktionen verknüpft.

KNF: Alle Maxterme, für die die logische Funktion gleich 0 ist, werden durch Konjunktionen verknüpft.

Beispiel 2.25 (KNF, DNF)
KNF und DNF von folgender Funktion:

Minterm	Maxterm	x_1	x_2	x_3	y
$\overline{x_1}\,\overline{x_2}\,\overline{x_3}$	$x_1 \vee x_2 \vee x_3$	0	0	0	0
$\overline{x_1}\,\overline{x_2}\,x_3$	$x_1 \vee x_2 \vee \overline{x_3}$	0	0	1	1
$\overline{x_1}\,x_2\,\overline{x_3}$	$x_1 \vee \overline{x_2} \vee x_3$	0	1	0	1
$\overline{x_1}\,x_2 x_3$	$x_1 \vee \overline{x_2} \vee \overline{x_3}$	0	1	1	0
$x_1\,\overline{x_2}\,\overline{x_3}$	$\overline{x_1} \vee x_2 \vee x_3$	1	0	0	1
$x_1\,\overline{x_2}\,x_3$	$\overline{x_1} \vee x_2 \vee \overline{x_3}$	1	0	1	0
$x_1 x_2\,\overline{x_3}$	$\overline{x_1} \vee \overline{x_2} \vee x_3$	1	1	0	0
$x_1 x_2 x_3$	$\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}$	1	1	1	0

Disjunktive Normalform: $\overline{x_1}\,\overline{x_2}\,x_3 \vee \overline{x_1}\,x_2\,\overline{x_3} \vee x_1\,\overline{x_2}\,\overline{x_3}$
Konjunktive Normalform: $x_1 \vee x_2 \vee x_3 \wedge x_1 \vee \overline{x_2} \vee \overline{x_3} \wedge \overline{x_1} \vee x_2 \vee \overline{x_3} \wedge \overline{x_1} \vee \overline{x_2} \vee x_3 \wedge \overline{x_1} \vee \overline{x_2} \vee \overline{x_3}$

Vereinfachung von Schaltfunktionen

Anmerkung: Die disjunktiven und konjunktiven Normalformen liefern zweistufige Schaltungen, aber in der Regel nicht solche mit minimaler Gatterzahl.

Anmerkung: Maximale Vereinfachung ist heutzutage nicht mehr so bedeutsam wie früher.

Definition 2.26 (Ordnung eines Terms, Primterm, Hauptterm)
Die *Ordnung eines Terms* ist die Anzahl der nicht im Term vorkommenden Variablen.
Ein Term, der in einer Darstellung nicht mit einem anderen Term gleicher Ordnung vereinfacht werden kann, heißt *Primterm*.
Ein Term *erfaßt* einen anderen, wenn der erste durch Streichung von Variablen aus dem zweiten hervorgeht.
Ein Primterm heißt *wesentlich* oder *Hauptterm*, wenn es mindestens einen Minterm gibt, der von keinem anderen Primterm erfaßt wird.

Erklärung 2.27 (Prinzipielle Struktur von Vereinfachungsverfahren)

1. Bildung vereinfachte Terme möglichst hoher Ordnung, d.h. Primterme;

2. Hinzunahme passender Minterme, sofern die Schaltfunktion für diese Minterme nicht festgelegt ist;

3. Erkennung der Hauptterme;

4. Hinzunahme möglichst weniger nichtwesentlicher Primterme zu dem Haupttermen, um Schaltfunktion darzustellen;

Die Bildung der vereinfachten Terme erfolgt durch Finden von Termen, die sich in genau einer Variablen unterscheiden.

Definition 2.28 (Minimalformen)
Bei den Verfahren entstehen häufig als Resultate folgende vier Formen:

> DMF: disjunktive Minimalform; Disjunktion von mit Konjunktionen verknüpften (ev. negierten) Eingangsvariablen (jeweils möglichst wenig Junktoren);
>
> KMF: konjunktive Minimalform; Konjunktion von mit Disjunktionen verknüpften (ev. negierten) Eingangsvariablen (jeweils möglichst wenig Junktoren);
>
> NDMF: DMF für Negation der Schaltfunktion;
>
> NKMF: KMF für Negation der Schaltfunktion;

Anmerkung: Zur Vereinfachung von Schaltungen können folgende Verfahren verwendet werden:

- schaltalgebraische Umformungen;
- graphische Verfahren (Karnaugh-Veitch-Diagramme);
- algorithmische Verfahren (Quine-McCluskey-Verfahren);

Vereinfachung mittels Schaltalgebra

Beispiel 2.29 (Vereinfachung mittels Schaltalgebra)
Vereinfachung mittels Schaltalgebra von Schaltung in DNF in DMF:

$$\begin{aligned}
& x_1 x_2 \overline{x_3}\, x_4 \vee \overline{x_1}\, x_2 \overline{x_3}\, x_4 \vee x_1 \overline{x_2}\, x_3 x_4 \vee x_1 \overline{x_2}\, x_3 \overline{x_4} \vee \overline{x_1}\, \overline{x_2}\, x_3 x_4 \vee \overline{x_1}\, \overline{x_2}\, x_3 \overline{x_4} \\
= \; & (x_1 \vee \overline{x_1})x_2 \overline{x_3}\, x_4 \vee x_1 \overline{x_2}\, x_3(x_4 \vee \overline{x_4}) \vee \overline{x_1}\, \overline{x_2}\, x_3(x_4 \vee \overline{x_4}) \\
= \; & \underbrace{x_2 \overline{x_3}\, x_4}_{\text{Term1.Ordnung}} \vee x_1 \overline{x_2}\, x_3 \vee \overline{x_1}\, \overline{x_2}\, x_3 \\
= \; & x_2 \overline{x_3}\, x_4 \vee (x_1 \vee \overline{x_1})\, \overline{x_2}\, x_3 \\
= \; & \underbrace{x_2 \overline{x_3}\, x_4}_{\text{Hauptterm}} \vee \underbrace{\overline{x_2}\, x_3}_{\text{Hauptterm}}
\end{aligned}$$

Anmerkung: Schaltalgebraische Vereinfachung ist in der Regel nur bei Schaltfunktionen mit geringer Zahl von Variablen gut durchführbar.
Meistens dient sie nur zur teilweisen und nicht zur vollständigen Vereinfachung.

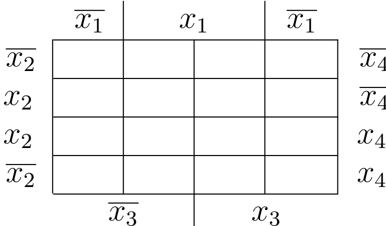
Vereinfachung mittels KV-Diagramm

Definition 2.30 (Karnaugh-Veitch-Diagramm)
Ein *Karnaugh-Veitch-Diagramm* ist eine zweidimensionale Abwicklung einer Wahrheitstabelle. Es besteht bei einer Schaltfunktion mit n Eingangsvariablen aus 2^n Feldern, die jeweils einen Minterm repräsentieren. Die Minterme zweier benachbarter Felder unterscheiden sich in genau einer Variable (diese Beziehung gilt auch über die Ränder des KV-Diagramms!).

Definition 2.31 (Aufbau eines kanonischen KV-Diagramms)
Die Felder sind in einem Rechteck des Formats $2^{\lceil n/2 \rceil} \times 2^{\lfloor n/2 \rfloor}$ angeordnet.
Ein KV-Diagramm für x_1 besteht aus zwei nebeneinanderliegenden Feldern.
Ein KV-Diagramm für $x_1, \ldots x_{n+1}$ entsteht aus einem für $x_1, \ldots x_n$ dadurch,

daß dasselbe Diagramm gespiegelt angeklebt wird. Die Minterme des alten Teils werden um $\overline{x_{n+1}}$, die des neuen Teils x_{n+1} ergänzt.

Beispiel 2.32 (KV-Diagramm)
Das KV-Diagramm für 4 Eingangsvariablen sieht wie folgt aus:



Erklärung 2.33 (Benachbarte Felder im KV-Diagramm)
Im KV-Diagramm differieren benachbarte Felder in nur einer Variablen. Das kann man wie folgt überlegen:
Die Eigenschaft gilt für KV_2. Wenn man KV_{n+1} aus KV_n zusammensetzt, dann gilt die Eigenschaft für jede Hälfte für sich, weil

- die Terme des alten und neuen Teil werden — jeder für sich — um dieselbe Variable ergänzt und
- die Spiegelung des neuen Teils beeinflußt die Eigenschaft nicht.

Die ursprünglichen Terme in den Feldern des alten und des neuen Teils unterscheiden sich an den Nahtstelle nicht (wegen der Spiegelung). Nach der Ergänzung um x_{n+1} unterscheiden sie sich genau in dieser Variablen.

Erklärung 2.34 (Vereinfachung einer Schaltfunktion mittels KV-Diagramm)

1. Feststellen der Zahl der Eingangsvariablen und Erstellen des KV-Diagramms;
2. Eintragen der Werte der Ausgangsvariablen als '0', '1' oder '*' ("don't care" = egal) in entsprechende Felder;
3. Zusammenfassung benachbarter 1-Felder zu Blöcken eventuell unter Einbeziehung von "don't cares"; ein 1-Feld darf in mehreren Blöcken vorkommen;
4. iterative Zusammenfassung gleichgroßer benachbarter Blöcke; ein Block darf in mehreren umfassenden Blöcken vorkommen;

5. Aufstellung der vereinfachten Schaltfunktion: jeder Block ist Primterm und entspricht einer Konjunktion von Eingangsvariablen; Hauptterme und eine minimale Anzahl weiterer Primterme werden für die Schaltfunktion benutzt;

Anmerkung: Wenn man in obigem Verfahren 0-Blöcke bildet, bestimmt man die disjunktive Minimalform der negierten Funktion, d.h. die NDMF. Daraus kann durch den Shannonschen Dualitätssatz die KMF gewonnen werden.

Beispiel 2.35 (Minimierung einer Schaltfunktion I)
Folgende Schaltfunktion sei zu minimieren:

Minterm	Maxterm	x_1	x_2	x_3	y_1	y_2
$\overline{x_1}\,\overline{x_2}\,\overline{x_3}$	$x_1 \vee x_2 \vee x_3$	0	0	0	1	1
$\overline{x_1}\,\overline{x_2}\,x_3$	$x_1 \vee x_2 \vee \overline{x_3}$	0	0	1	1	1
$\overline{x_1}\,x_2\,\overline{x_3}$	$x_1 \vee \overline{x_2} \vee x_3$	0	1	0	1	1
$\overline{x_1}\,x_2 x_3$	$x_1 \vee \overline{x_2} \vee \overline{x_3}$	0	1	1	0	0
$x_1\,\overline{x_2}\,\overline{x_3}$	$\overline{x_1} \vee x_2 \vee x_3$	1	0	0	0	*
$x_1\,\overline{x_2}\,x_3$	$\overline{x_1} \vee x_2 \vee \overline{x_3}$	1	0	1	0	*
$x_1 x_2\,\overline{x_3}$	$\overline{x_1} \vee \overline{x_2} \vee x_3$	1	1	0	0	0
$x_1 x_2 x_3$	$\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}$	1	1	1	0	0

Das KV-Diagramm für y_1 sieht wie folgt aus:

	$\overline{x_1}$	x_1	x_1	$\overline{x_1}$
$\overline{x_2}$	1	0	0	1
x_2	1	0	0	0
	$\overline{x_3}$	$\overline{x_3}$	x_3	x_3

Resultierende Formen: (DMF) $y_1 = \overline{x_1}\,\overline{x_2} \vee \overline{x_1}\,\overline{x_3}$ bzw. (NDMF) $y_1 = \overline{x_1 \vee x_2 x_3}$.

Das KV-Diagramm für y_2 sieht wie folgt aus:

	$\overline{x_1}$	x_1	x_1	$\overline{x_1}$
$\overline{x_2}$	1	*	*	1
x_2	1	0	0	0
	$\overline{x_3}$	$\overline{x_3}$	x_3	x_3

Resultierende Formen: (DMF) $y_2 = \overline{x_2} \vee \overline{x_1}\,\overline{x_3}$ bzw. (NDMF) $y_2 = \overline{x_1 \vee x_2 x_3}$.

Beispiel 2.36 (Minimierung einer Schaltfunktion II)
Mittels Schaltalgebra vereinfachtes Beispiel (von oben):
$x_1x_2\overline{x_3}\,x_4 \vee \overline{x_1}\,x_2\,\overline{x_3}\,x_4 \vee x_1\,\overline{x_2}\,x_3x_4 \vee x_1\,\overline{x_2}\,x_3\,\overline{x_4} \vee \overline{x_1}\;\overline{x_2}\,x_3x_4 \vee \overline{x_1}\;\overline{x_2}\,x_3\,\overline{x_4}$

x_1	x_2	x_3	x_4	y
1	1	0	1	1
0	1	0	1	1
1	0	1	1	1
1	0	1	0	1
0	0	1	1	1
0	0	1	0	1
Rest:				0

Das KV-Diagramm für y sieht wie folgt aus:

	$\overline{x_1}$	x_1	x_1	$\overline{x_1}$	
$\overline{x_2}$	0	0	1	1	$\overline{x_4}$
x_2	0	0	0	0	$\overline{x_4}$
x_2	1	1	0	0	x_4
$\overline{x_2}$	0	0	1	1	x_4
	$\overline{x_3}$	$\overline{x_3}$	x_3	x_3	

Resultierende Formen: (DMF) $y = x_2\,\overline{x_3}\,x_4 \vee \overline{x_2}\,x_3$ bzw. (NDMF) $y = \overline{\overline{x_3}\;\overline{x_4} \vee x_2x_3 \vee \overline{x_2}\;\overline{x_3}}$.

Beispiel 2.37 (Minimierung einer Schaltfunktion III)
Mehr als vier Eingangsvariablen:

x_1	x_2	x_3	x_4	x_5	y
0	0	1	0	0	1
0	0	1	0	1	1
0	0	1	1	0	1
0	0	1	1	1	1
1	0	0	0	0	1
1	1	0	0	0	1
Rest:					0

Das KV-Diagramm für y sieht wie folgt aus:

	$\overline{x_5}$				x_5				
	$\overline{x_1}$	x_1		$\overline{x_1}$		x_1		$\overline{x_1}$	
$\overline{x_2}$	0	1	0	1	1	0	0	0	$\overline{x_4}$
x_2	0	1	0	0	0	0	0	0	$\overline{x_4}$
x_2	0	0	0	0	0	0	0	0	x_4
$\overline{x_2}$	0	0	0	1	1	0	0	0	x_4
	$\overline{x_3}$		x_3				$\overline{x_3}$		

Resultierende Formen: (DMF) $y = \overline{x_1}\,\overline{x_2}\,x_3 \vee x_1\,\overline{x_3}\,\overline{x_4}\,\overline{x_5}$ bzw. (NDMF) $y = \overline{\overline{x_1}\,\overline{x_3} \vee \overline{x_1}\,x_2 \vee x_1x_3 \vee x_1x_5 \vee x_1x_4}$.

```
Vorsicht: Blöcke über 4 Spalten lassen sich nicht immer
zusammenfassen!
```

Vereinfachung mittels Quine-McCluskey-Verfahrens

Anmerkung: Als durchlaufendes Beispiel für diesen Abschnitt wird die Schaltfunktion

x_1	0	0	0	0	0	1	1	1	1	
x_2	0	0	1	1	1	0	1	1	1	
x_3	0	1	0	0	1	1	0	0	1	
x_4	1	0	0	1	0	0	0	1	0	
y	1	1	1	1	1	1	1	1	1	Rest=0

verwendet.

Anmerkung: Naive Zusammenfassung von Mintermen führt nicht immer zu einer Schaltfunktion mit minimaler Gatterzahl z.B. ergibt sich für obige Schaltfunktion bei ungeschickter Zusammenfassung

$$
\begin{aligned}
y &= \overline{x_1}\,\overline{x_2}\,\overline{x_3}\,x_4 \vee \overline{x_1}\,\overline{x_2}\,x_3\,\overline{x_4} \vee \overline{x_1}\,x_2\,\overline{x_3}\,\overline{x_4} \vee \overline{x_1}\,x_2\,\overline{x_3}\,x_4 \vee \overline{x_1}\,x_2x_3\,\overline{x_4} \vee \\
&\quad \vee x_1\,\overline{x_2}\,x_3\,\overline{x_4} \vee x_1x_2\,\overline{x_3}\,\overline{x_4} \vee x_1x_2\,\overline{x_3}\,x_4 \vee x_1x_2x_3\,\overline{x_4} \\
&= \overline{x_1}\,\overline{x_2}\,\overline{x_4}\,(\overline{x_3} \vee x_3) \vee \overline{x_1}\,x_2\,\overline{x_3}\,(\overline{x_4} \vee x_4) \vee \overline{x_1}\,x_2x_3\,\overline{x_4} \vee \\
&\quad \vee x_1x_3\,\overline{x_4}\,(\overline{x_2} \vee x_2) \vee x_1x_2\,\overline{x_3}\,(\overline{x_4} \vee x_4) \\
&= \overline{x_1}\,\overline{x_2}\,\overline{x_4} \vee \overline{x_1}\,x_2\,\overline{x_3} \vee \overline{x_1}\,x_2x_3\,\overline{x_4} \vee x_1x_3\,\overline{x_4} \vee x_1x_2x_3
\end{aligned}
$$

Dies läßt sich nur noch mit komplexeren schaltalgebraischen Umformungen vereinfachen, aber nicht mit Ausklammern von Variable und ihrer Negation wie im ersten Schritt.
$\rightsquigarrow$gleichzeitig alle Terme bestimmen, die durch Zusammenfassung entstehen können

Erklärung 2.38 (Quine-McCluskey-Verfahren)

1. Notiere alle Minterme der Funktion und sortiere sie nach der Zahl der nichtnegierten Variablen.
2. Für i:=0, 1, 2,... tue:
   (a) Fasse Terme i-ter Ordnung, die sich bezüglich Negation in derselben Variablen unterscheiden, zu einem Term $(i+1)$-ter Ordnung zusammen. Markiere die zusammengefaßten Terme als "erfaßt" und füge den neuen Term in eine Liste (von Termen $(i+1)$-ter Ordnung) ein.
   Führe diesen Schritt so durch, daß alle Paare von zueinander passenden Termen abgedeckt werden.
   (b) Wenn keine Terme mehr zusammengefaßt werden können, brich die Schleife ab, ansonsten sortiere die neue Liste nach der Zahl der nichtnegierten Variablen.
3. Die markierten Terme (beliebiger Ordnung) werden durch einfachere Terme (höherer Ordnung) erfaßt. In der weiteren Betrachtung spielen nur noch unmarkierte ein Rolle. Es ist aber möglich, daß Minterme durch mehrere unmarkierte abgedeckt werden. Zuerst werden also die unmarkierten Terme bestimmt, die exklusiv einen Minterm abdecken (Hauptterme).
4. Zu den Hauptermen wird eine minimale Zahl von unmarkierten Termen hinzugefügt, sodaß sämtliche Minterme abgedeckt werden. (durch systematisches Probieren).

Beispiel 2.39 (Anwendung des QuineMcCluskey-Verfahrens)
Das QM-Verfahren für obige Schaltfunktion wird wie folgt durchgeführt (Terme sind zur einfachen Referenz mit dem Dezimalwert der Belegungen kommentiert):

0. Ordnung:

- 1: $\overline{x_1}\,\overline{x_2}\,\overline{x_3}\,x_4$, 2: $\overline{x_1}\,\overline{x_2}\,x_3\,\overline{x_4}$, 4: $\overline{x_1}\,x_2\,\overline{x_3}\,\overline{x_4}$,
- 5: $\overline{x_1}\,x_2\,\overline{x_3}\,x_4$, 6: $\overline{x_1}\,x_2x_3\,\overline{x_4}$, 10: $x_1\,\overline{x_2}\,x_3\,\overline{x_4}$, 12: $x_1x_2\,\overline{x_3}\,\overline{x_4}$,

- 13: $x_1x_2\,\overline{x_3}\,x_4$, 14: $x_1x_2x_3\,\overline{x_4}$

1. Ordnung:

- 1,5: $\overline{x_1}\;\overline{x_3}\,x_4$, 2,6: $\overline{x_1}\,x_3\,\overline{x_4}$, 2,10: $\overline{x_2}\,x_3\,\overline{x_4}$, 4,5: $\overline{x_1}\,x_2\,\overline{x_3}$, 4,6: $\overline{x_1}\,x_2\,\overline{x_4}$, 4,12: $x_2\,\overline{x_3}\;\overline{x_4}$,
- 5,13: $x_2\,\overline{x_3}\,x_4$, 6,14: $x_2x_3\,\overline{x_4}$, 10,14: $x_1x_3\,\overline{x_4}$, 12,13: $x_1x_2\,\overline{x_3}$, 12,14: $x_1x_2\,\overline{x_4}$

alle Terme 0. Ordnung wurden erfaßt

2. Ordnung:
2,6;10,14: $x_3\,\overline{x_4}$, 2,10;6,14: $x_3\,\overline{x_4}$, 4,5;12,13: $x_2\,\overline{x_3}$, 4,6;12,14: $x_2\,\overline{x_4}$, 4,12;5,13: $x_2\,\overline{x_3}$, 4,12;6,14: $x_2\,\overline{x_4}$
alle Terme 1. Ordnung bis auf $\overline{x_1}\;\overline{x_3}\,x_4$ wurden erfaßt

3. Ordnung:
es ist keine Vereinfachung mehr möglich$\Longrightarrow$Abbruch

Primterme, d.h. Kandidaten für Hauptterme, sind daher $\overline{x_1}\;\overline{x_3}\,x_4$, $x_3\,\overline{x_4}$, $x_2\,\overline{x_3}$ und $x_2\,\overline{x_4}$. Für alle Minterme wird geprüft, welche dieser Primterme sie abdecken:

x_1	0	0	0	0	0	1	1	1	1
x_2	0	0	1	1	1	0	1	1	1
x_3	0	1	0	0	1	1	0	0	1
x_4	1	0	0	1	0	0	0	1	0
$\overline{x_1}\;\overline{x_3}\,x_4$	*			*					
$x_3\,\overline{x_4}$		*			*	*			*
$x_2\,\overline{x_3}$			*	*			*	*	
$x_2\,\overline{x_4}$			*		*		*		*

Hauptterme sind also $\overline{x_1}\;\overline{x_3}\,x_4$ (wegen 0001), $x_3\,\overline{x_4}$ (wegen 0010 und 1010) und $x_2\,\overline{x_3}$ (wegen 1101). $x_2\,\overline{x_4}$ ist kein Hauptterm!
Die Hauptterme decken die Funktion vollständig ab, also muß kein weiterer Term hinzugefügt werden. Also gilt: $y = \overline{x_1}\;\overline{x_3}\,x_4 \vee x_3\,\overline{x_4} \vee x_2\,\overline{x_3}$.

Anmerkung: Generelle Tricks zur Vereinfachung der Überdeckungsmatrix:

- Hauptterme identifizieren und die Minterme streichen, die davon überdeckt werden: im Beispiel ist p1 Hauptterm (wegen m4)

	m1	m2	m3	m4	...
p1	0	0	1	1	...
p2	1	1	0	0	...
p3	1	1	1	0	...
p4	0	1	1	0	...

- Primterme entfernen, die von anderen Primtermen überdeckt werden: im Beispiel werden p2 und p4 von p3 überdeckt

	m1	m2	m3	m4	m5	...
p1	1	1	0	1	0	...
p2	1	1	0	0	1	...
p3	1	1	1	0	1	...
p4	0	1	1	0	1	...

- Mintermspalten entfernen, wenn sie von anderen überdeckt werden: im Beispiel wird m2 von m3 überdeckt

	m1	m2	m3	m4	m5	...
p1	1	0	0	0	0	...
p2	0	1	0	0	1	...
p3	0	1	1	0	0	...

Anmerkung: Das Überdeckungsproblem ist NP-schwer (hat also vermutlich exponentielle Laufzeit), weil es äquivalent ist zum Erfüllbarkeitsproblem. Man sucht nämlich eine minimale Zahl von Primtermen, die alle Minterme überdecken.
Das Beispiel

	m1	m2	m3	m4	...
p1	0	0	1	1	...
p2	1	1	0	0	...
p3	1	1	1	0	...
p4	0	1	1	0	...

läßt sich formulieren als

$$(p2 \vee p3) \wedge (p2 \vee p3 \vee p4) \wedge (p1 \vee p3 \vee p4) \wedge (p1) = 1$$

Kapitel 3

Komplexere digitale Funktionen

3.1 Elementare Schaltnetze

3.1.1 Kodewandler

Definition 3.1 (Kode)
Ein *Kode* ist in der Digitaltechnik eine Zuordnungsvorschrift, die einem Zeichen aus einem Zeichenvorrat eine Bitkombination zuordnet.

Definition 3.2 (Arten von Kodes)
Es gibt

alphanumerische Kodes: dienen zur Darstellung von Buchstaben Ziffern und Sonderzeichen;
Beispiele: 7bit-ASCII, 7bit-EBCDIC, 8bit-ANSI/ISO, 16bit-Unicode...

numerische Kodes: dienen zur Darstellung von Zahlen; unterschieden wird, ob einzelne Ziffern einzeln kodiert werden (BCD-Darstellung) oder als Binärwort;
Beispiele: Dualkode, Graykode, BCD-Dualcode...

Definition 3.3 (Kodewandler)
Ein *Kodewandler* ist eine Schaltung, das die Aufgabe hat, Worte aus verschiedenen Kodes ineinander umzuwandeln. Sie realisieren die Funktion eines speziellen *Festwertspeichers*.

Anmerkung: Kodewandler können seriell sein (d.h. die Bits des Eingangsworts fallen nacheinander an) oder parallel (alle Bits des Eingangsworts liegen sofort an). Wir betrachten im folgenden parallele Wandler.

Abbildung 3.1: Paralleler Kodewandler

Beispiel eines Kodewandlers zwischen Graykode und Dualkode

Problem: Kodestreifen für eindimensionale Positionsaufnahme z.B. auf einer Trommel

Wert	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
									X	X	X	X	X	X	X	X
Dualkode					X	X	X	X					X	X	X	X
			X	X			X	X			X	X			X	X
		X		X		X		X		X		X		X		X

⇝normaler Dualkode unbrauchbar, weil beim Übergang zwischen zwei Positionen falsche Kodes entstehen (z.B. bei Übergang von "7" auf "8" entsteht "15");
⇝Abhilfe durch Kode der bei jedem Wechsel genau an einer Stelle ein Bit ändert = Graykode

Wert	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
									X	X	X	X	X	X	X	X
Graykode					X	X	X	X	X	X	X	X				
			X	X	X	X					X	X	X	X		
		X	X			X	X			X	X			X	X	

Definition 3.4 (Graykode)
Der (n+1)bit-Graykode für $0, \ldots 2^{n+1}$ entsteht wie folgt: Die Kodefolge des n-bit-Graykodes für $0, \ldots 2^n$ wird um das höchstwertige Bit 0 ergänzt und als (n+1)bit-Graykode für diese Zahlen genommen. Dieselbe Kodefolgedes n-bit-Graykodes für $0, \ldots 2^n$ wird um das höchstwertige Bit 1 ergänzt und für $2^{n+1}, 2^{n+1} - 1, \ldots, 2^n + 1$ verwendet.

Anmerkung: Die Minterme benachbarter Zellen in einem KV-Diagramm unterscheiden sich in genau einer Variablen. Ein Graykode läßt sich dadurch gewinnen, daß man die Felder eines KV-Diagramms so durchläuft, daß immer nur benachbarte Felder besetzt werden und ein geschlossener Kurvenzug entsteht. Daher gibt es mehr als einen Graykode.

Erklärung 3.5 (Konstruktion des Kodewandlers für 4bit-Graykode in 4bit-Dualk
Folgende Schaltfunktion muß realisiert werden:

x_3	x_2	x_1	x_0	y_3	y_2	y_1	y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	0	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Das KV-Diagramm für y_0 sieht wie folgt aus:

	$\overline{x_0}$	x_0	x_0	$\overline{x_0}$	
$\overline{x_1}$	0	1	0	1	$\overline{x_3}$
x_1	1	0	1	0	$\overline{x_3}$
x_1	0	1	0	1	x_3
$\overline{x_1}$	1	0	1	0	x_3
	$\overline{x_2}$	$\overline{x_2}$	x_2	x_2	

Schaltfunktion:(DMF)

$$\begin{aligned} y_0 &= x_0\,\overline{x_1}\;\overline{x_2}\;\overline{x_3} \vee \overline{x_0}\,x_1\,\overline{x_2}\;\overline{x_3} \vee x_0x_1x_2\,\overline{x_3} \vee \overline{x_0}\;\overline{x_1}\,x_2\,\overline{x_3} \\ &\quad \vee x_0\,\overline{x_1}\,x_2x_3 \vee \overline{x_0}\,x_1x_2x_3 \vee \overline{x_0}\;\overline{x_1}\;\overline{x_2}\,x_3 \vee x_0x_1\,\overline{x_2}\,x_3 \\ &= (x_0 \not\leftrightarrow x_1) \not\leftrightarrow (x_2 \not\leftrightarrow x_3) \end{aligned}$$

Das KV-Diagramm für y_1 sieht wie folgt aus:

	$\overline{x_0}$	x_0		$\overline{x_0}$	
$\overline{x_1}$	0	0	1	1	$\overline{x_3}$
x_1	1	1	0	0	$\overline{x_3}$
x_1	0	0	1	1	x_3
$\overline{x_1}$	1	1	0	0	x_3
	$\overline{x_2}$		x_2		

Schaltfunktion: (DMF) $y_1 = \overline{x_1}\,x_2\,\overline{x_3} \vee x_1\,\overline{x_2}\,\overline{x_3} \vee x_1x_2x_3 \vee \overline{x_1}\,\overline{x_2}\,x_3 = (x_1 \not\leftrightarrow x_2) \not\leftrightarrow x_3$

Das KV-Diagramm für y_2 sieht wie folgt aus:

	$\overline{x_0}$	x_0		$\overline{x_0}$	
$\overline{x_1}$	0	0	1	1	$\overline{x_3}$
x_1	0	0	1	1	$\overline{x_3}$
x_1	1	1	0	0	x_3
$\overline{x_1}$	1	1	0	0	x_3
	$\overline{x_2}$		x_2		

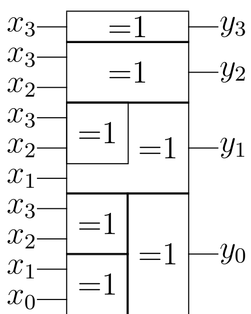
Schaltfunktion: (DMF) $y_2 = x_2\,\overline{x_3} \vee \overline{x_2}\,x_3 = x_2 \not\leftrightarrow x_3$

Das KV-Diagramm für y_3 sieht wie folgt aus:

	$\overline{x_0}$	x_0		$\overline{x_0}$	
$\overline{x_1}$	0	0	0	0	$\overline{x_3}$
x_1	0	0	0	0	$\overline{x_3}$
x_1	1	1	1	1	x_3
$\overline{x_1}$	1	1	1	1	x_3
	$\overline{x_2}$		x_2		

Schaltfunktion: (DMF) $y_3 = x_3$

Resultierende Schaltung:



Erklärung 3.6 (Weitere Beispiele für Kodewandler)

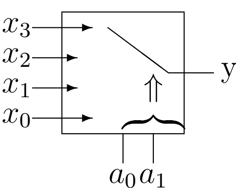
- 1-aus-N-Kodierer: von N Eingängen ist immer genau einer mit 1-Signal belegt; die Nummer dieses Eingangs wird als $\log_2 N$bit-Dualzahl ausgegeben;

- 1-aus-M-Dekodierer: an den Eingängen liegt eine Dualzahl an, die die Nummer eines von M Ausgängen ist; dieser wird auf 1-Signal gelegt; wenn ein Aktivierungseingang vorhanden ist, entspricht diese Schaltung einem 1-zu-M-Multiplexer (siehe unten);

- Anzeigentreiber (BCD-zu-7Segment-Dekodierer)

3.1.2 Multiplexer und Demultiplexer

Definition 3.7 (Multiplexer)
Eine Schaltung, die unter kN Eingängen (durch $\log_2 N$ Eingänge bestimmt) k Eingänge mit k Ausgängen direkt verbindet, heißt *kN-zu-k-Multiplexer*.
Ein Multiplexer ist ein durch eine duale Adresse gesteuerter Wahlschalter.

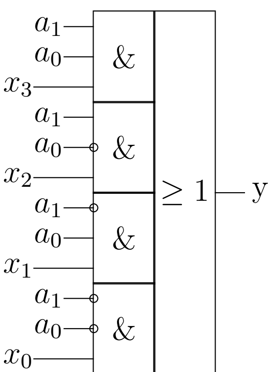


Beispiel 3.8 (Entwurf eines 4-zu-1-Multiplexers)
Von 4 Eingängen soll einer auf einen Ausgang durchgeschaltet werden.

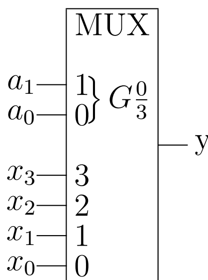
a_1	a_0	y
0	0	x_0
0	1	x_1
1	0	x_2
1	1	x_3

Aus dieser Wahrheitstabelle ergibt sich folgende Schaltfunktion: $y = \overline{a_1}\,\overline{a_0}\,x_0 \vee \overline{a_1}\,a_0x_1 \vee a_1\,\overline{a_0}\,x_2 \vee a_1a_0x_3$

Resultierende Schaltung:

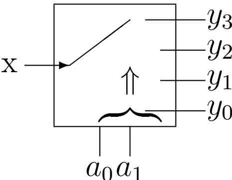


Anmerkung: Schaltsymbol für Multiplexer nach DIN 40900:



Anmerkung zum Schaltsymbol: Die Steuereingänge sind mit 0 und 1 gekennzeichnet. Dadurch ist die Wertigkeit definiert. "G" steht für "Gate", d.h. eine UND-Verknüpfung. Dadurch wird eine UND-Abhängigkeit zu den entsprechend markierten Eingängen gekennzeichnet.

Definition 3.9 (Demultiplexer)
Eine Schaltung, die k Eingänge mit k von kN Ausgängen direkt verbindet (durch $\log_2 N$ Eingänge bestimmt), heißt *k-zu-kN-Demultiplexer*.
Ein Demultiplexer ist ein durch eine duale Adresse gesteuerter Wahlschalter.



Beispiel 3.10 (Entwurf eines 1-zu-4-Demultiplexers)
Ein Eingang soll auf einen von vier Ausgängen durchgeschaltet werden.

a_1	a_0	y_0	y_1	y_2	y_3
0	0	x	0	0	0
0	1	0	x	0	0
1	0	0	0	x	0
1	1	0	0	0	x

Aus dieser Wahrheitstabelle ergeben sich folgende Schaltfunktionen:

$$
\begin{aligned}
y_0 &= \overline{a_0}\,\overline{a_1}\,x \\
y_1 &= a_0\,\overline{a_1}\,x \\
y_2 &= \overline{a_0}\,a_1 x \\
y_3 &= a_0 a_1 x
\end{aligned}
$$

Resultierende Schaltung:

Anmerkung: Schaltsymbol für Demultiplexer nach DIN 40900:

Anmerkung zum Schaltsymbol: Die Steuereingänge sind mit 0 und 1 gekennzeichnet. Dadurch ist deren Wertigkeit definiert. "G" steht für "Gate", d.h. eine UND-Verknüpfung. Dadurch wird eine UND-Abhängigkeit zu den entsprechend markierten Ausgängen gekennzeichnet.

Anmerkung: Häufig sind digitale Schaltelemente mit einem Aktivierungseingang (i.Z. "EN"=enable) ausgestattet. Wenn dieser auf 0 gelegt wird, liegen alle gewöhnlichen Ausgänge auf 0. Sogenannte Tristate-Ausgänge (mit ∇ gekennzeichnet, nehmen einen dritten Zustand an, in dem dieser Ausgang mit keinem Potential verbunden ist ("abgetrennt" oder "hochohmig"). Damit sind ODER- bzw. UND-Verknüpfungen durch direkte Verbindung derartiger Ausgänge möglich.

Erweiterung von Multiplexern und Demultiplexern

Erklärung 3.11 (Erweiterung von Multiplexern)
Eine Erweiterung eines n-zu-1-Multiplexers (Bild 3.2) kann erfolgen

a) zu einem kn-zu-k-Multiplexer durch Parallelschaltung von k n-zu-1-Multiplexern; die Adreßleitungen werden an alle Multiplexer geführt;

b) zu einem kn-zu-1-Multiplexer durch $\log_2 k$ zusätzliche Adreßleitungen, die über einer 1-aus-k-Dekodierer auf die Aktivierungseingänge von k n-zu-1-Multiplexern geführt werden; die k Ausgänge werden mit einer ODER-Schaltung verknüpft;

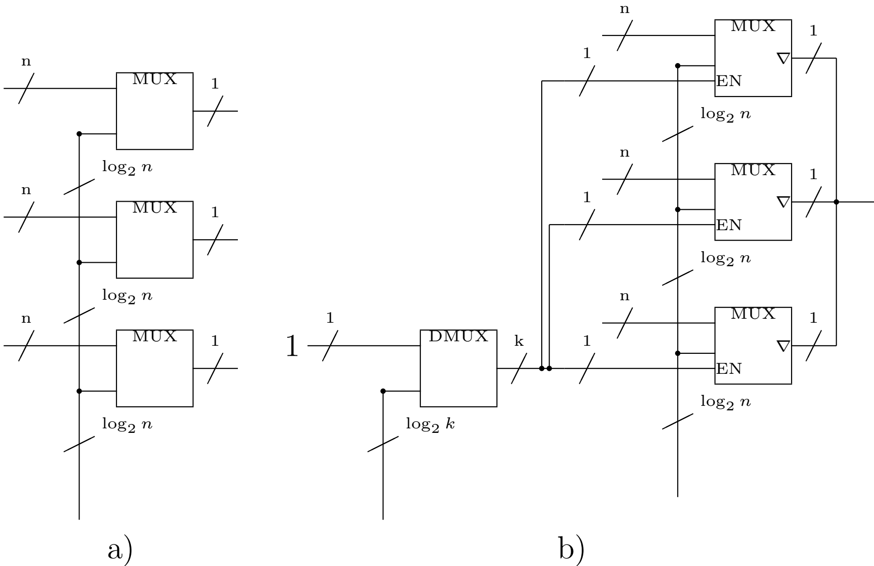


Abbildung 3.2: Kaskadierung von Multiplexern

Erklärung 3.12 (Erweiterung von Demultiplexern)
Eine Erweiterung eines 1-zu-n-Demultiplexers (Bild 3.3) kann erfolgen

a) zu einem k-zu-kn-Demultiplexer durch Parallelschaltung von k 1-zu-n-Demultiplexern; die Adreßleitungen werden an alle Demultiplexer geführt;

b) zu einem 1-zu-kn-Demultiplexer durch $\log_2 k$ zusätzliche Adreßleitungen, die über einer 1-aus-k-Dekodierer auf die Aktivierungseingänge von k 1-zu-n-Demultiplexern geführt werden; die k Eingänge sind direkt verbunden;

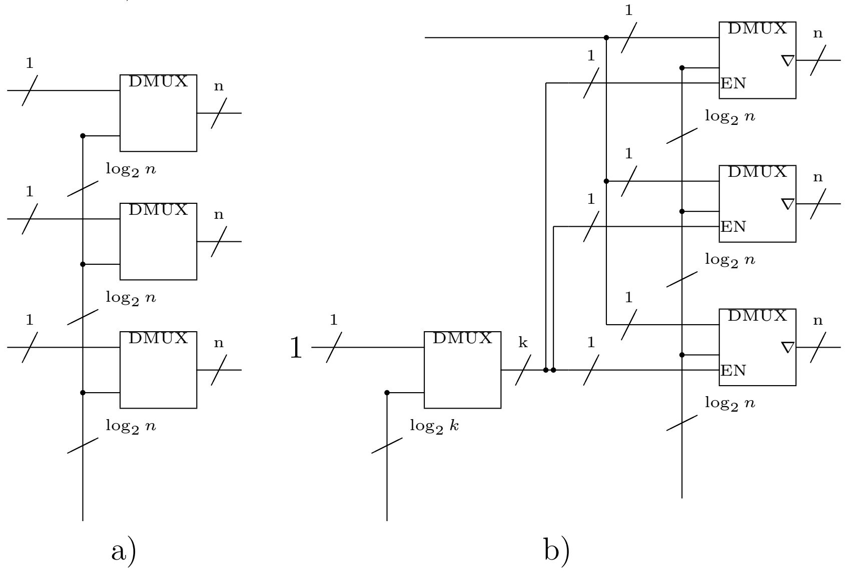


Abbildung 3.3: Kaskadierung von Demultiplexern

Anwendungen von Multiplexern und Demultiplexern

- einfache Festwertspeicher (durch MUX);
- ungetaktete Schieberegister (durch MUX);
- Parallel-Seriell-Parallel-Wandler (durch MUX);

3.1.3 Addierer

Halbaddierer, Volladdierer

Definition 3.13 (Halbaddierer, Volladdierer)
Ein *Halbaddierer* realisiert die Addition zweier einstelliger Dualzahlen.
Ein *Volladdierer* realisiert die Addition zweier einstelliger Dualzahlen inklu-

sive Übertrag.

Erklärung 3.14 (Aufbau eines Halbaddierers)
Wahrheitstafel eines Halbaddierers

a_i	b_i	S_i	C_i
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Resultierende Schaltfunktionen:

$$\begin{aligned} S_i &= a_i \not\leftrightarrow b_i \\ C_i &= a_i \wedge b_i \end{aligned}$$

Schaltsymbol:

a_i =1 S_i
b_i & C_i

Erklärung 3.15 (Aufbau eines Volladdierers)
Wahrheitstafel eines Volladdierers

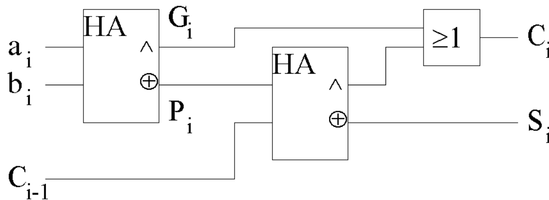
a_i	b_i	C_{i-1}	S_i	C_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Schaltfunktion:

$$\begin{aligned} S_i &= C_{i-1}\,\overline{a_i}\,\overline{b_i} \vee C_{i-1}a_ib_i \vee \overline{C_{i-1}}\,\overline{a_i}\,b_i \vee \overline{C_{i-1}}\,a_i\,\overline{b_i} \\ &= C_{i-1} \not\leftrightarrow (a_i \not\leftrightarrow b_i) \\ C_i &= C_{i-1}\,\overline{a_i}\,b_i \vee C_{i-1}a_ib_i \vee \overline{C_{i-1}}\,a_ib_i \vee C_{i-1}a_i\,\overline{b_i} \\ &= C_{i-1}(a_i \not\leftrightarrow b_i) \vee a_ib_i \end{aligned}$$

$$P_i^* = a_i \not\leftrightarrow b_i, G_i = a_i b_i$$
$$S_i = P_i^* \not\leftrightarrow C_{i-1}$$
$$C_i = P_i^* C_{i-1} \vee G_i$$

Anmerkung: Man kann einen Volladdierer aus zwei Halbaddierern und einem Oder-Gatter zusammensetzen.



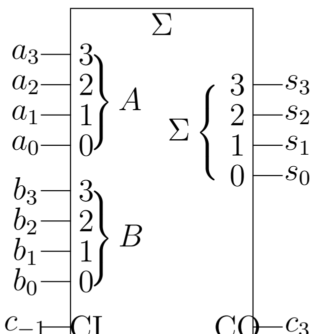
Erklärung 3.16 (Kaskadierung von Addierern)

- durch serielle Schaltung; Problem: durchlaufender Übertrag...
- mit Übertragsvorhersagelogik (Lookahead-Carry)

$$\begin{aligned} C_0 &= G_0 \vee P_0^* C_{-1} \\ C_1 &= G_1 \vee P_1^* C_0 = G_1 \vee P_1^* G_0 \vee P_1^* P_0^* C_{-1} \\ &\dots \\ C_n &= \left(C_{-1} \bigwedge_{j=0}^{n} P_j^* \right) \vee \bigvee_{i=0}^{n} G_i \left(\bigwedge_{j=i+1}^{n} P_j^* \right) \end{aligned}$$

Vorteil: vorhersagbare Durchlaufzeit; Nachteil: aufwendig;

Anmerkung: Schaltsymbol für Addierer nach DIN 40900:



3.1.4 Komparatoren

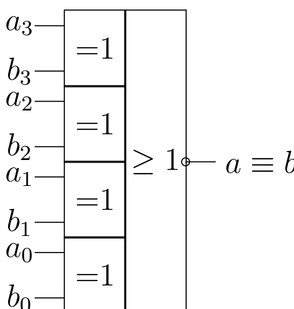
Gleichheitskomparator

Definition 3.17 (Gleichheitskomparator)
Ein *Gleichheitskomparator* prüft, ob zwei Dualzahlen gleich sind oder nicht.

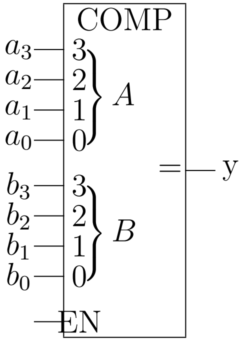
Erklärung 3.18 (Aufbau eines Gleichheitskomparators)
Zwei Zahlen sind gleich, wenn jede Stelle gleich ist. Gleichheit zweier Ein-Bit-Zahlen ist durch eine negiertere Antivalenz überprüfbar.

$$a_i b_i \vee \overline{a_i}\ \overline{b_i} = \overline{a_i \not\leftrightarrow b_i}$$

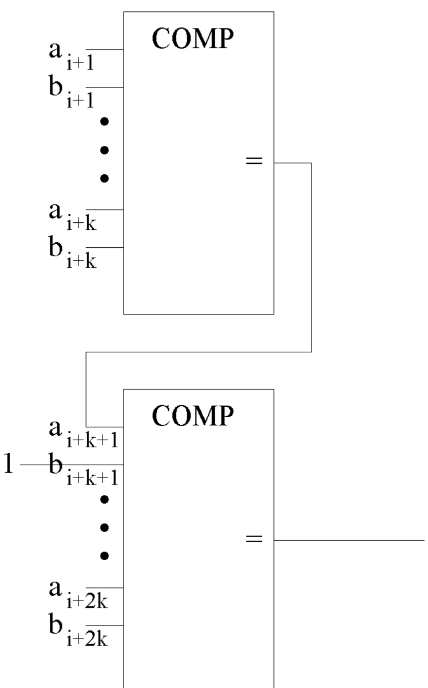
Resultierende Schaltung:



Anmerkung: Schaltsymbol für Komparator nach DIN 40900:



Anmerkung: Eine Kaskadierung von Gleichheitskomparatoren passiert durch Verbinden eines Komparatorausgangs mit dem Aktivierungseingang des folgenden Komparators oder durch UND-Verknüpfung.



Komparator mit Größenvergleich

Definition 3.19 (Komparator mit Größenvergleich)
Ein *Komparator mit Größenvergleich* vergleicht zwei Dualzahlen a und b und liefert an drei Ausgängen, ob $a < b$, $a = b$ oder $a > b$ ist.

Erklärung 3.20 (Aufbau eines Größenvergleichskomparators)
Wahrheitstafel des Komparators:

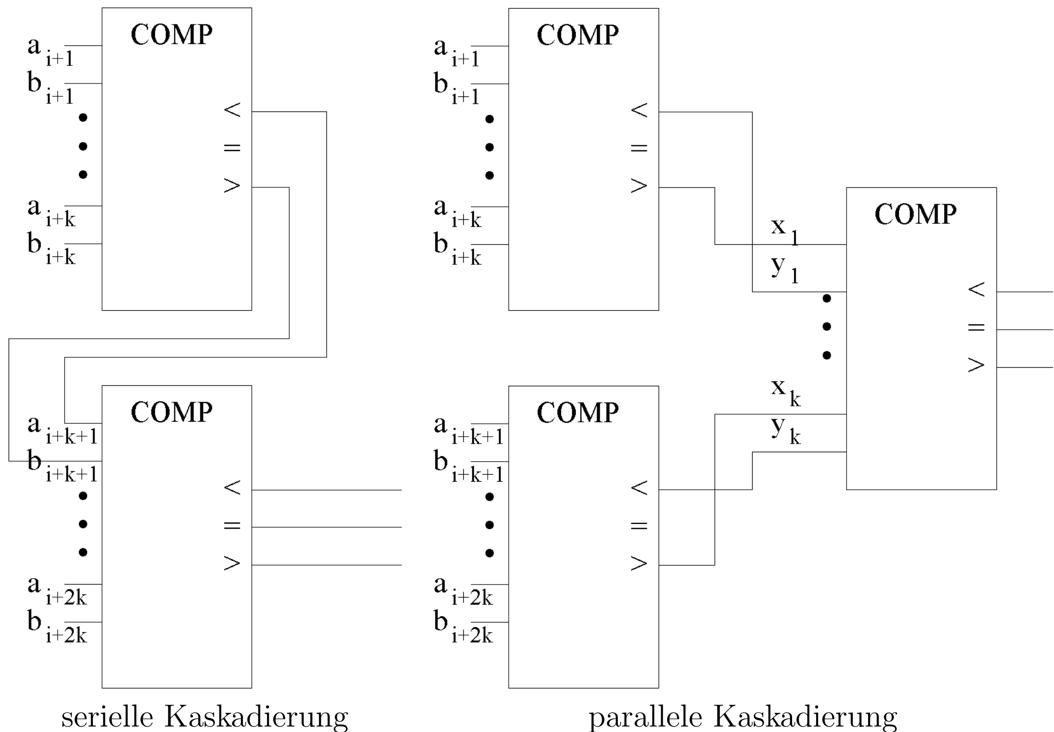
a_i	b_i	$y_< := a_i < b_i$	$y_= := a_i = b_i$	$y_> := a_i > b_i$
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

Resultierende Schaltfunktionen:

$$\begin{aligned} y_< &= \overline{a_i}\, b_i \\ y_= &= \overline{a_i \not\leftrightarrow b_i} \\ y_> &= a_i\, \overline{b_i} \end{aligned}$$

`zweistufigen Komparator vorführen`

Anmerkung: Eine Kaskadierung von Größenvergleichskomparatoren passiert durch Verbinden der beiden Ungleich-Ausgänge mit zwei Eingangsstellen eines anderen Komparators:



serielle Kaskadierung parallele Kaskadierung

Wenn ein Komparator gleiche Eingangsbelegung für beide Eingangswerte hat, so sind der Größer- und Kleinerausgang jeweils 0. Ansonsten wird die Kombination (1,0) bzw. (0,1) als Kodierung des Resultats an die andere Stufe weitergegeben.

`Leitungen kreuzen bei serieller Kaskadierung`

3.2 Elementare Schaltwerke

Definition 3.21 (Schaltwerk)
Ein Schaltwerk ist eine digitale Schaltung zum Verarbeiten von Schaltvariablen, bei der der Wert der Ausgangsgrößen neben den aktuellen Werten der Eingangsgrößen auch von deren früheren Werten zu endlich vielen vorhergehenden Zeitpunkten abhängt.

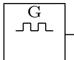
3.2.1 Digitale Oszillatoren

Definition 3.22 (Digitaler Oszillator)
Ein digitaler Oszillator ist eine Schaltung, die an ihrem Ausgang eine periodisches binäres Signal erzeugt.

Erklärung 3.23 (Typen von Oszillatoren)

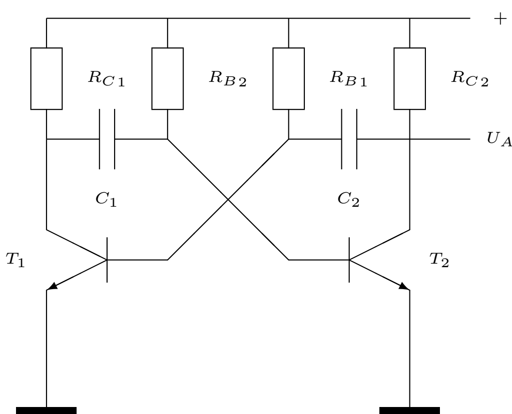
- Transistorschaltung (astabiler Multivibrator)
- rückgekoppelter Schwellwertschalter
- rückgekoppelte Negationsglieder
- Quarzschaltungen

Anmerkung: Schaltsymbol nach DIN:



Oszillator mittels Transistorschaltung

Erklärung 3.24 (Aufbau und Funktionsweise)

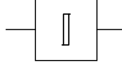


Funktionsweise: Sei T2 gesperrt und T1 leitend. Dann wird C_1 über R_{B2} aufgeladen. Gleichzeitig wird C_2 über R_{B1} und R_C entladen (auf Spannung 0). Sobald die Spannung an der Basis von T_2 hoch genug ist, beginnt dieser zu leiten. Über C_2 wird die Basis von T_1 mit stärker negativem Potential verbunden und sperrt stärker. Diese Sperren erhöht das positive Potential

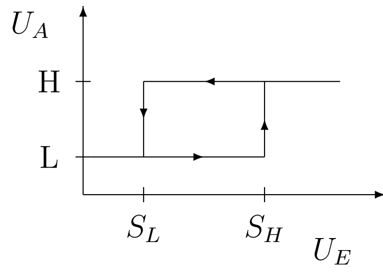
an der Basis von T_2: die Schaltung kippt. Eine analoge Überlegung zeigt, daß dieser Zustand ebenfalls nach einiger Zeit umkippt.
Periodendauer$\approx 2R_BC\ln 2$

Oszillator mittels Schmitt-Trigger

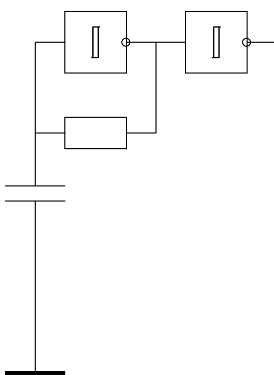
Definition 3.25 (Schmitt-Trigger)
Ein Schmitt-Trigger ist ein 1-Bit-Analog-Digitalwandler mit zwei Schaltschwellen S_H und S_L. Ist das Eingangssignal $\geq S_H$, so liegt der Ausgang auf H-Pegel, ist das Eingangssignal $\leq S_L$, dann auf L-Pegel. Zwischen beiden Schwellen verharrt die Schaltung auf dem bisherigen Zustand. Diesen Effekt nennt man Hysterese ("Zurückbleiben").
Schaltsymbol:



Erklärung 3.26 (Schaltverhalten des Schmitt-Triggers)
Die Eingangs-Ausgangsfunktion des Schmitt-Triggers ist von der Durchlaufrichtung abhängig. Sie ist im folgenden Diagramm durch Pfeile gekennzeichnet:



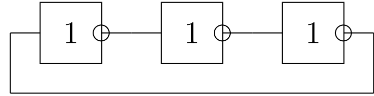
Erklärung 3.27 (Aufbau und Funktionsweise des Oszillators)



Funktionsweise: Der Kondensator sei anfangs entladen. Dann liegt am Ausgang des ersten Schmitt-Triggers H-Pegel. Der Kondensator wird über R bis auf S_H aufgeladen und der Schmitt-Trigger kippt. Durch den L-Pegel am Ausgang des Schmitt-Triggers wird der Kondensator bis auf S_L entladen und der S.T. kippt wiederum usw.
Periodendauer$\approx RC$

Oszillator mittels rückgekoppelter Inverter

Erklärung 3.28 (Aufbau und Funktionsweise)



Funktionsweise: Eine ungerade Anzahl n von Invertern wird in Reihe geschaltet und der Ausgang des letzten mit dem Eingang des ersten verbunden. Eine Umschaltung am Eingang wird erst nach n Gatterlaufzeiten am Ausgang wirksam.
Periodendauer: $2nt_{\text{Gatter}}$

Anmerkung: Anwendung: Messung der Gatterlaufzeit

Quarzoszillatoren

Anmerkung: Basis der Quarzoszillatoren sind Schwingquarze mit Eigenfrequenz >100kHz. Diese wird häufig durch digitale Frequenzteiler auf den benötigten Wert gebracht.

3.2.2 Monostabile Kippstufen (Monoflops)

Definition 3.29 (Monostabile Kippstufe)
Eine *monostabile Kippstufe (Monoflop)* reagiert auf einen Impuls am Eingang mit einem Impuls definierter Länge t_D am Ausgang (t_D=Haltezeit). Das Monoflop hat nur eine stabile Lage am Ausgang, in die es eine definierte Zeit nach Ende der Eingangsimpulse zurückkehrt.
Man unterscheidet

retriggerbare Monoflops: jede Taktflanke am Eingang bewirkt daß der Ausgang des Monoflops t_D im 1-Zustand verbleibt;

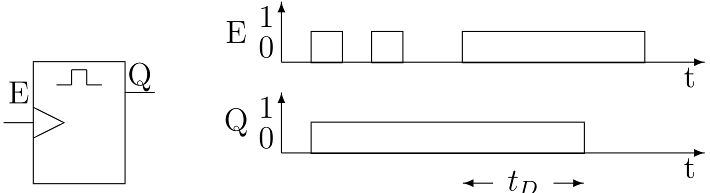
nicht retriggerbare Monoflops: eine Taktflanke am Eingang während der Ausgang des Monoflops im 1-Zustand ist, verlängert nicht die Haltezeit;

Anmerkung: Anwendungen von Monoflops sind:

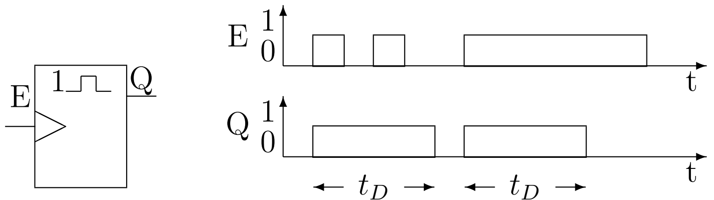
- Zeitverzögerungen
- Erzeugung von Impulsen definierter Dauer
- dynamische Speicherung von Signalen

Ihre Bedeutung in der Digitaltechnik im Abnehmen aufgrund stärkeren Einsatzes taktgesteuerter Systeme.

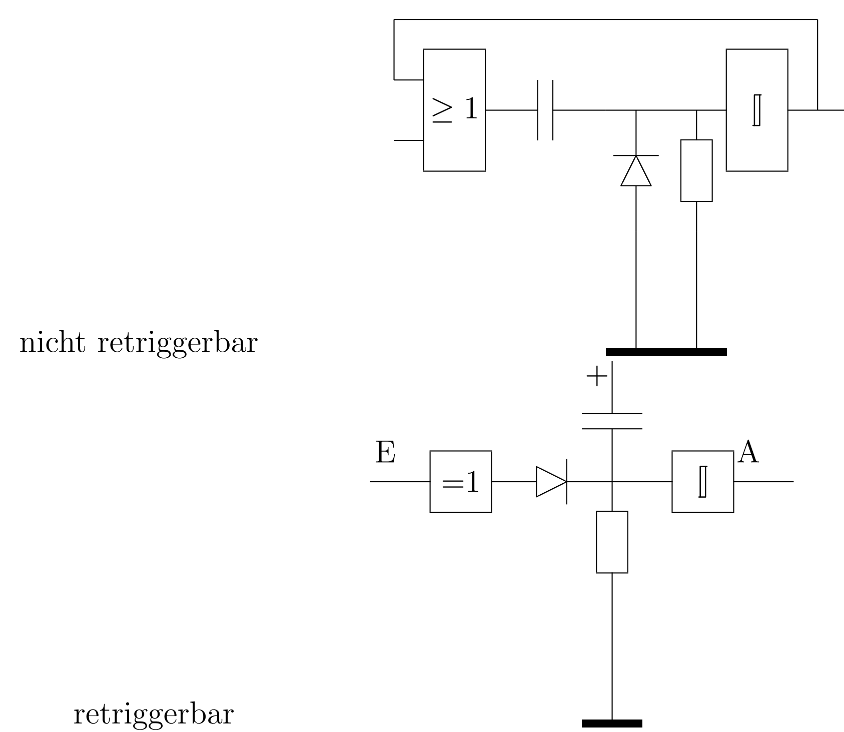
Erklärung 3.30 (Retriggerbares Monoflop)
Schaltsymbol und Impulsdiagramm



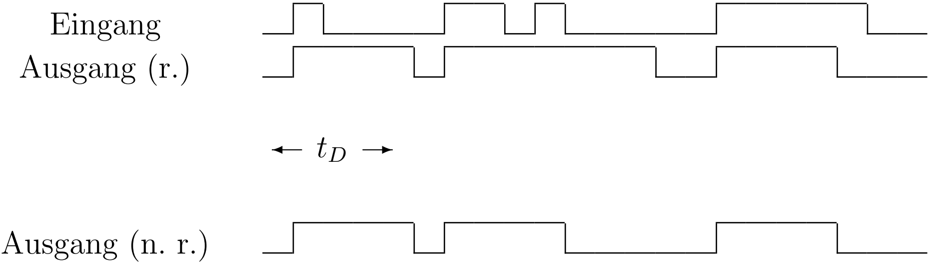
Erklärung 3.31 (Nicht retriggerbares Monoflop)
Schaltsymbol und Impulsdiagramm



Beispiel 3.32 (technische Realisierung von Monoflops)



Anmerkung: Die Reaktion auf Impulse am Eingang sieht bei einem retriggerbaren bzw. nicht retriggerbaren Monoflop wie folgt aus:



Anmerkung: Mit einem retriggerbaren Monoflop kann man beispielsweise feststellen, ob eine Pulsfolge unterbrochen ist.

3.2.3 Bistabile Kippstufen (Flipflops)

Definition 3.33 (E)
in *Flipflop* hat am Ausgang zwei stabile Zustände; es kann die Information 0 oder 1 speichern. Über entsprechende Eingänge kann ein Flipflop gesetzt oder rückgesetzt werden.
Sofern die Ausgänge auf die Eingänge rückgekoppelt sind, werden sie mit einem Zeitindex versehen (z.B. Q_m ist der Wert von Q zu t_m).

Anmerkung: Arten und Ansteuerung von Flipflops

Art	ungetaktet	einzustands-gesteuert	einflanken-gesteuert	zweizustands-gesteuert	zweiflanken-gesteuert
RS-Flipflop	X	X	X	X	X
D-Flipflop		X	X		
T-Flipflop			X		
JK-Flipflop		X	X	X	X

RS-Flipflop

Definition 3.34 (RS-Flipflop)
Ein *RS-Flipflop* hat einen Setzeingang S, einen Rücksetzeingang R und einen Ausgang Q und dessen Negation $\overline{Q}$.

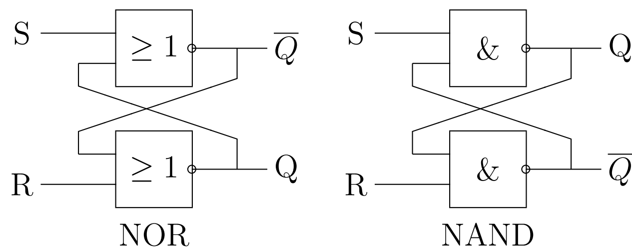
- Wenn S=1 und R=0 ist, wird Q auf 1 gesetzt.
- Wenn S=0 und R=1 ist, wird Q auf 0 gesetzt.
- Wenn R und S 0 sind, behält Q seinen alten Wert.
- Wenn R und S 1 sind, ist die Ausgangsbelegung undefiniert.

Erklärung 3.35 (Konstruktion eines RS-Flipflops)
Bei Speicherschaltungen wird in die Wahrheitstabelle oder ins KV-Diagramm der alte Wert des Ausgangs einbezogen.

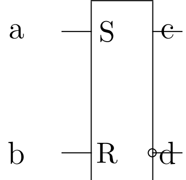
Das KV-Diagramm für Q_{n+1} sieht wie folgt aus:

	$\overline{S}$	S	S	$\overline{S}$
$\overline{R}$	0	1	1	1
R	0	*	*	0
	$\overline{Q_n}$	$\overline{Q_n}$	Q_n	Q_n

Resultierende Schaltfunktion: $Q_{n+1} = S \vee \overline{R}\, Q_n$
NOR-Version: $\overline{Q_{n+1}} = \overline{S \vee \overline{R \vee \overline{Q_n}}}$
NAND-Version: $Q_{n+1} = \overline{\overline{S} \wedge \overline{\overline{R} \vee Q_n}}$



Anmerkung: Schaltsymbol nach DIN:



Anmerkung: DIN-Erläuterung zu R und S: Das Symbol für Setzen ist "S", das für Rücksetzen "R". Alle mit "m" gekennzeichneten Ausgänge

- nehmen unabhängig von R den 1-Zustand an, wenn S mit m markiert und 1 ist,
- nehmen unabhängig von S den 0-Zustand an, wenn R mit m markiert und 1 ist.

RS-Flipflop mit NAND-Schaltungen

a — S1 1 — c
b — R2 2 — d

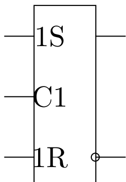
RS-Flipflop mit NOR-Schaltungen

a — S1 2 — c
b — R2 1 — d

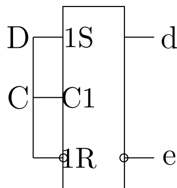
Einzustandsgesteuerte Flipflops

Definition 3.36 (Einzustandsgesteuertes Flipflop)
Ein *einzustandsgesteuertes Flipflop* übernimmt Daten nur zu Zeiträumen, in denen ein Steuersignal 1 ist.

Anmerkung: Schaltsymbol nach DIN (C=Steuerung(Control)):



Anmerkung: D-Flipflop zur Übernahme von 1bit-Information mit Steuersignal:



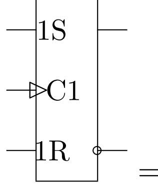
Anmerkung: Einzustandsgesteuerte Flipflops können nicht seriell gekoppelt werden (z.B. für Schieberegister), da Information durchläuft. $\leadsto$Trennung von Eingang und Ausgang durch interne Zwischenspeicherung (Master-Slave-Prinzip).

Anmerkung: Ähnliches Problem: Information an Eingängen wird nicht zu einem bestimmten Zeitpunkt, sondern in einem Zeitraum abgefragt.

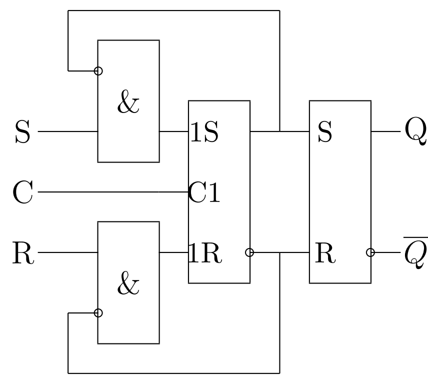
Einflankengesteuerte Flipflops

Definition 3.37 (Einflankengesteuertes Flipflop)
Ein *einflankengesteuertes Flipflop* übernimmt Daten nur zu dem Zeitpunkt, an dem ein Zustandswechsel des Steuersignal in eine vorgegebene Richtung stattfindet. Bei Übernahme beim Wechsel 0 nach 1 heißt das Flipflop positiv flankengesteuert, ansonsten negativ flankengesteuert.

Erklärung 3.38 (Aufbau eines einflankengesteuerten Flipflop)



=



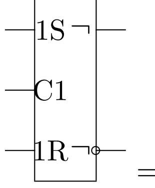
Erklärung: Nach Durchschalten des ersten Flipflops werden die Eingänge über die Und-Gatter gesperrt. Die Information wird dann an das zweite Flipflop weitergegeben.

Anmerkung: Damit obige Schaltung funktioniert, müssen die Informationen eine Gatterlaufzeit vor Übergang anliegen ("Setzzeit") und nach Übergang ebenfalls circa eine Gatterlaufzeit ("Haltezeit").

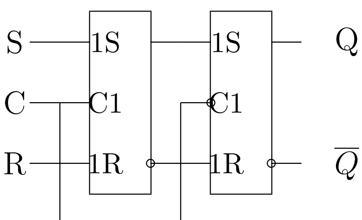
Zweizustandsgesteuerte Flipflops

Definition 3.39 (Zweizustandsgesteuertes Flipflop)
Ein *zweizustandsgesteuertes Flipflop* übernimmt Daten während des Zeitraums, in dem das Steuersignal 1 ist. Die Weitergabe an die Ausgänge erfolgt erst, wenn das Steuersignal 0 ist ("verzögerte (retardierte) Ausgänge").

Erklärung 3.40 (Aufbau eines zweizustandsgesteuerten Flipflops)



=

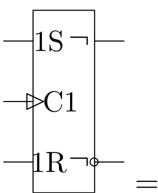


Das Retardierungszeichen ¬ besagt, daß ein Ausgang erst dann seinen Wert wechselt, wenn der verursachende Eingang wieder seinen ursprünglichen Wert angenommen hat (in unserem Fall der Steuereingang).

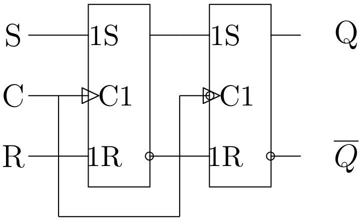
Zweiflankengesteuerte Flipflops

Definition 3.41 (Zweiflankengesteuertes Flipflop)
Ein *zweiflankengesteuertes Flipflop* übernimmt Daten nur zu dem Zeitpunkt, an dem ein Zustandswechsel des Steuersignal in eine vorgegebene Richtung stattfindet. Die Weitergabe an die Ausgänge erfolgt erst, wenn der umgekehrte Zustandswechsel des Steuersignals stattfindet.

Erklärung 3.42 (Aufbau eines zweiflankengesteuerten Flipflops)

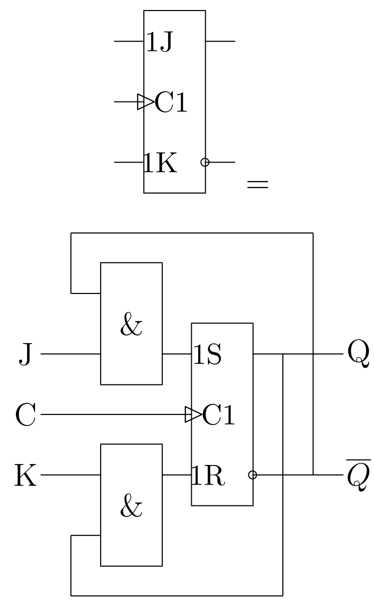


=



Spezielle Flipflopvarianten

Definition 3.43 (JK-Flipflop)
Ein *JK-Flipflop* ist ein flankengesteuertes oder zweizustandsgesteuertes RS-Flipflop, bei dem der R-Eingang mit der Konjunktion vom neuen Eingang K und Ausgang Q und der S-Eingang mit der Konjunktion vom neuen Eingang J und Ausgang $\overline{Q}$ verbunden ist.



Anmerkung: Wahrheitstafeln von JK-Flipflop und RS-Flipflop:

RS-Flipflop			JK-Flipflop		
S	R	Q_n	J	K	Q_n
0	0	Q_{n-1}	0	0	Q_{n-1}
0	1	0	0	1	0
1	0	1	1	0	1
1	1	????	1	1	$\overline{Q_{n-1}}$

Das JK-Flipflop wechselt bei der 1-1-Eingangsbelegung in jedem Takt seinen Zustand.

Erklärung 3.44 (Allgemeine Schaltfunktion des JK-Flipflops)
Die Schaltfunktion des Ausgangs Q_m in Abhängigkeit von J, K und Q_{m-1} ergibt sich aus folgender Wahrheitstabelle:

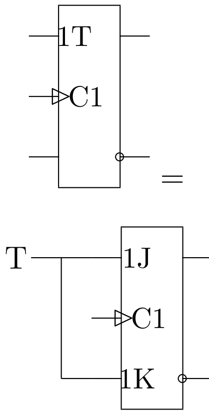
J	K	Q_{m-1}	Q_m
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Das resultierende KV-Diagramm für Q_m sieht wie folgt aus:

	$\overline{J}$	J		$\overline{J}$
$\overline{K}$	0	1	1	1
K	0	1	0	0
	$\overline{Q_{m-1}}$		Q_{m-1}	

Schaltfunktion: $Q_m = J\,\overline{Q_{m-1}} \vee \overline{K}\,Q_{m-1}$

Definition 3.45 (T-Flipflop)
Ein *T-Flipflop* (T steht für "toggle") ist ein JK-Flipflop, bei dem der J-Eingang und der K-Eingang mit dem neuen Eingang T verbunden sind.



Wahrheitstafel des T-Flipflops:

T	Q_n
0	Q_{n-1}
1	$\overline{Q_{n-1}}$

3.2.4 Digitale Zähler

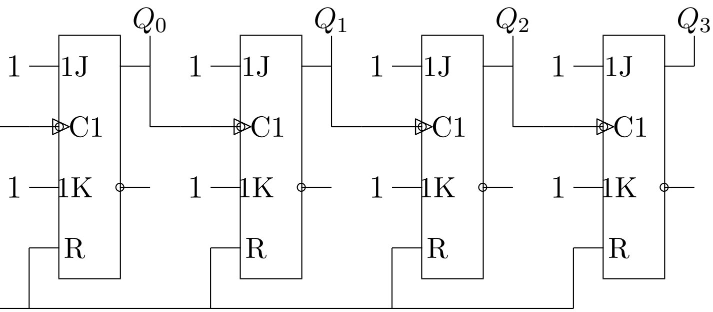
Definition 3.46 (Zähler)
Zähler sind Schaltwerke, die Taktimpulse zählen. Man unterscheidet Zähler nach

- Zählrichtung (vorwärts oder rückwärts),
- Zahlensystem (dual oder dekadisch) und
- Arbeitsweise (asynchron oder synchron).

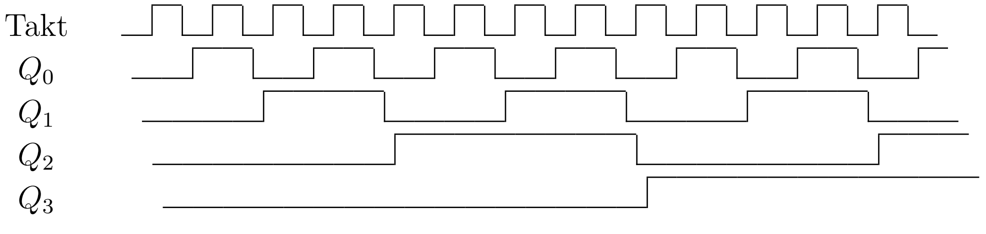
Asynchrone Zähler

Definition 3.47 (Asynchrone Zähler)
Bei einem *asynchronen Zähler* sind nicht alle Takteingänge der beteiligten Flipflops an den zentralen Takt angeschlossen.

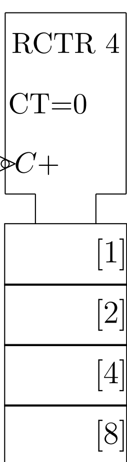
Erklärung 3.48 (Aufbau eines asynchronen Vorwärtsdualzählers)



Bei jedem Wechsel eines Flipflops von 1 nach 0 wird der nachfolgende Flipflop getriggert. Aufgrund der seriellen Kopplung der Zählstufen addieren sich die Verzögerungszeiten, d.h. die Stufen schalten nicht gleichzeitig.
Das Impulsdiagramm sieht wie folgt aus:

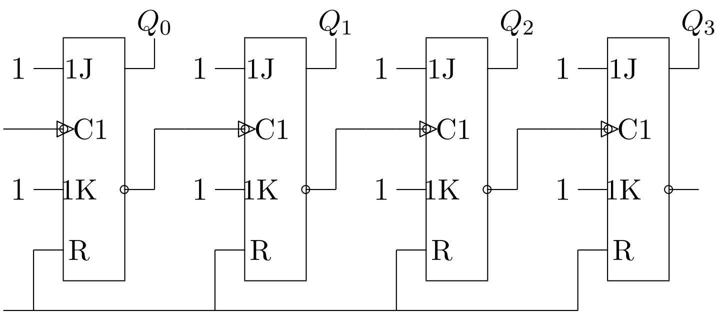


Anmerkung: Schaltsymbol nach DIN:

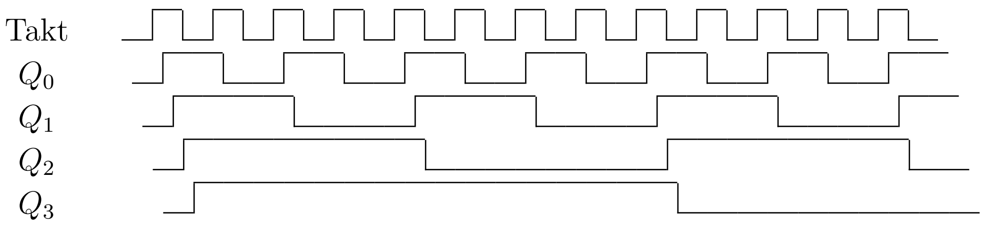


RCTR im Blockbeschreibungsfeld steht für "ripple counter" (asynchroner Zähler). Die Zahl hinter der Kennung gibt die Stelligkeit an. Die Beschriftung des Rücksetzeingang gibt an, auf welchen Wert der Inhalt (CT) gesetzt wird. $C+$ als Beschriftung am Takteingang heißt, daß der Zähler hochzählt. Die Zahlen in eckigen Klammern geben die Wertigkeit der Ausgänge an.

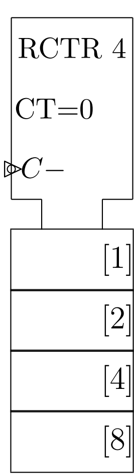
Erklärung 3.49 (Aufbau eines asynchronen Rückwärtsdualzählers)



Bei jedem Wechsel eines Flipflops von 0 nach 1 wird der nachfolgende Flipflop getriggert.
Das Impulsdiagramm sieht wie folgt aus:



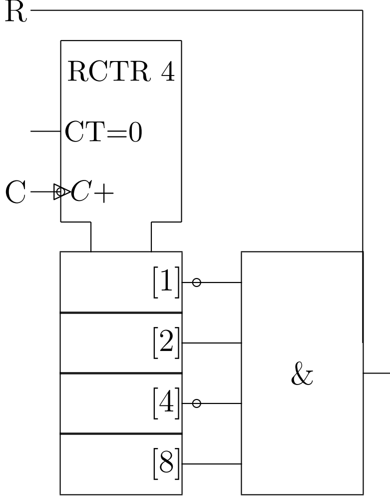
Anmerkung: Schaltsymbol nach DIN:



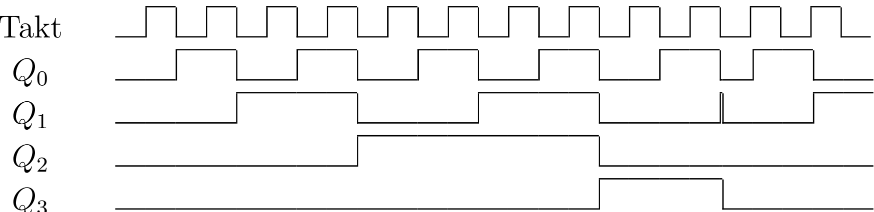
$C-$ als Beschriftung am Takteingang heißt, daß der Zähler herunterzählt.

Definition 3.50 (Asynchroner Modulo-m-Zähler)
Ein *asynchroner Modulo-m-Zähler* ist ein asynchroner Zähler, der nur die Werte 0 bis $m-1$ annimmt.

Erklärung 3.51 (Aufbau eines asynchronen Modulo-m-Zählers)
Er entsteht dadurch, daß die Ausgänge eines asynchronen Dualzähler (mit Stufenzahl $n > \log_2 m$) über ein Schaltnetz auf den Rücksetzeingang aller Stufen geführt wird. Das Schaltnetz stellt beim Erreichen des Żahlerstands m den Zähler auf 0.



Das Impulsdiagramm sieht wie folgt aus:



Durch die Gatterlaufzeit der Rückstelllogik tritt am Ausgang kurz der Wert m (in unserem Fall $m = 10$) auf.

Anmerkung: Probleme des asynchronen Modulo-m-Zählers

- wie bei allen asynchronen Zähler sind die Wert am Ausgang nach unterschiedlichen Verzögerungszeiten stabil

- kurzzeitig erscheint die unerwünschte Kombination am Ausgang

Synchrone Zähler

Definition 3.52 (Synchrone Zähler)
Bei einem *synchronen Zähler* sind alle Takteingänge der beteiligten Flipflops an den zentralen Takt angeschlossen. Die korrekte Weiterschaltung erfolgt durch Beschaltung der Vorbereitungseingänge J und K oder der Setz- und Rücksetzeingänge S und R.

Anmerkung: Aufgrund dieser Konstruktion haben alle Zählstufen dieselbe Verzögerungszeit.

Erklärung 3.53 (Aufbau eines synchronen Vorwärtsdualzählers)
Idee: Die neuen Werte von $Q_a, \ldots, Q_d$ hängen von deren vorhergehenden Werten ab, d.h. man konstruiert ein Schaltnetz, daß Eingänge der Zählstufen beschaltet.

Wert	Q_d	Q_c	Q_b	Q_a	Q'_d	Q'_c	Q'_b	Q'_a
0	0	0	0	0	0	0	0	1
1	0	0	0	1	0	0	1	0
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	1	0	0
4	0	1	0	0	0	1	0	1
5	0	1	0	1	0	1	1	0
6	0	1	1	0	0	1	1	1
7	0	1	1	1	1	0	0	0
8	1	0	0	0	1	0	0	1
9	1	0	0	1	1	0	1	0
10	1	0	1	0	1	0	1	1
11	1	0	1	1	1	1	0	0
12	1	1	0	0	1	1	0	1
13	1	1	0	1	1	1	1	0
14	1	1	1	0	1	1	1	1
15	1	1	1	1	0	0	0	0

Das KV-Diagramm für Q'_a sieht wie folgt aus:

	$\overline{Q_a}$	Q_a		$\overline{Q_a}$	
$\overline{Q_b}$	1	0	0	1	$\overline{Q_d}$
Q_b	1	0	0	1	$\overline{Q_d}$
Q_b	1	0	0	1	Q_d
$\overline{Q_b}$	1	0	0	1	Q_d
	$\overline{Q_c}$		Q_c		

$$Q'_a = \overline{Q_a}$$

Das KV-Diagramm für Q'_b sieht wie folgt aus:

	$\overline{Q_a}$	Q_a		$\overline{Q_a}$	
$\overline{Q_b}$	0	1	1	0	$\overline{Q_d}$
Q_b	1	0	0	1	$\overline{Q_d}$
Q_b	1	0	0	1	Q_d
$\overline{Q_b}$	0	1	1	0	Q_d
	$\overline{Q_c}$		Q_c		

$$Q'_b = Q_a \overline{Q_b} \vee \overline{Q_a} Q_b$$

Das KV-Diagramm für Q'_c sieht wie folgt aus:

	$\overline{Q_a}$	Q_a	Q_a	$\overline{Q_a}$	
$\overline{Q_b}$	0	0	1	1	$\overline{Q_d}$
Q_b	0	1	0	1	$\overline{Q_d}$
Q_b	0	1	0	1	Q_d
$\overline{Q_b}$	0	0	1	1	Q_d
	$\overline{Q_c}$	$\overline{Q_c}$	Q_c	Q_c	

$$Q'_c = Q_a Q_b \overline{Q_c} \vee \overline{Q_a} Q_c \vee \overline{Q_b} Q_c$$

Das KV-Diagramm für Q'_d sieht wie folgt aus:

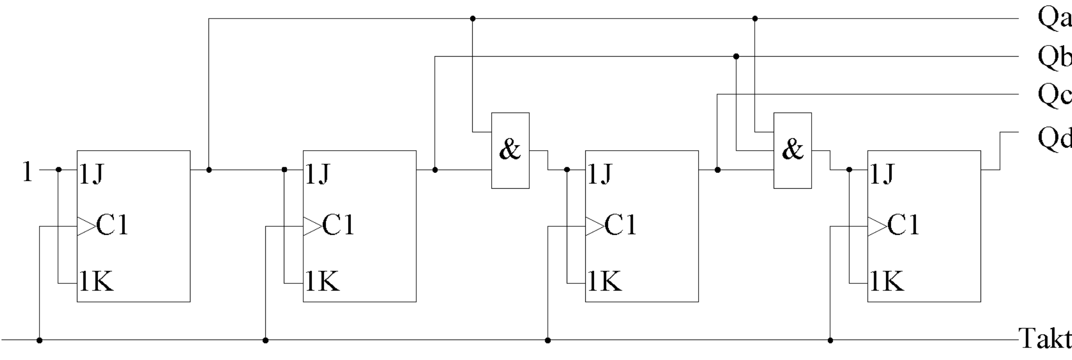
	$\overline{Q_a}$	Q_a	Q_a	$\overline{Q_a}$	
$\overline{Q_b}$	0	0	0	0	$\overline{Q_d}$
Q_b	0	0	1	0	$\overline{Q_d}$
Q_b	1	1	0	1	Q_d
$\overline{Q_b}$	1	1	1	1	Q_d
	$\overline{Q_c}$	$\overline{Q_c}$	Q_c	Q_c	

$$Q'_d = Q_a Q_b Q_c \overline{Q_d} \vee \overline{Q_a} Q_d \vee \overline{Q_b} Q_d \vee \overline{Q_c} Q_d$$

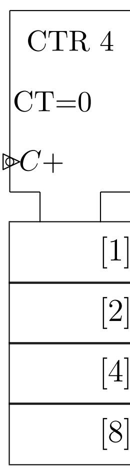
Koeffizientenvergleich mit Gleichung $Q' = J\,\overline{Q} \vee \overline{K}\,Q$ liefert:

$$\begin{aligned}
Q'_a = 1\,\overline{Q_a} \vee \overline{1}\,Q_a &\quad\Rightarrow\quad J_a = 1, K_a = 1 \\
Q'_b = Q_a\,\overline{Q_b} \vee \overline{Q_a}\,Q_b &\quad\Rightarrow\quad J_b = Q_a, K_b = Q_a \\
Q'_c = Q_aQ_b\,\overline{Q_c} \vee \overline{Q_aQ_b}\,Q_c &\quad\Rightarrow\quad J_c = Q_aQ_b, K_c = Q_aQ_b \\
Q'_d = Q_aQ_bQ_c\,\overline{Q_d} \vee \overline{Q_aQ_bQ_c}\,Q_d &\quad\Rightarrow\quad J_d = Q_aQ_bQ_c, K_d = Q_aQ_bQ_c
\end{aligned}$$

Resultierende Schaltung:



Anmerkung: DIN-Symbol für synchronen Zähler:



Erklärung 3.54 (Aufbau eines synchronen Rückwärtsdualzählers von 10 bis 5)
Idee: Die neuen Werte von $Q_a, \ldots, Q_d$ hängen von deren vorhergehenden Werten ab, d.h. man konstruiert ein Schaltnetz, daß Eingänge der Zählstufen beschaltet.

Wert	Q_d	Q_c	Q_b	Q_a	Q'_d	Q'_c	Q'_b	Q'_a
0	0	0	0	0	*	*	*	*
				...				
4	0	1	0	0	*	*	*	*
5	0	1	0	1	1	0	1	0
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	1	0
8	1	0	0	0	0	1	1	1
9	1	0	0	1	1	0	0	0
10	1	0	1	0	1	0	0	1
11	1	0	1	1	*	*	*	*
				...				
15	1	1	1	1	*	*	*	*

Wichtig: Damit Koeffizientenvergleich später möglich ist, dürfen die rückgeführten Ausgänge im KV-Diagramm nicht wegoptimiert werden!

Das KV-Diagramm für Q'_a sieht wie folgt aus:

	$\overline{Q_a}$	Q_a		$\overline{Q_a}$	
$\overline{Q_b}$	*	*	0	*	$\overline{Q_d}$
Q_b	*	*	0	1	$\overline{Q_d}$
Q_b	1	*	*	*	Q_d
$\overline{Q_b}$	1	0	*	*	Q_d
	$\overline{Q_c}$		Q_c		

$$Q_a' = \overline{Q_a}$$

Das KV-Diagramm für Q_b' sieht wie folgt aus:

	$\overline{Q_a}$	Q_a		$\overline{Q_a}$	
$\overline{Q_b}$	*	*	1	*	$\overline{Q_d}$
Q_b	*	*	1	0	$\overline{Q_d}$
Q_b	0	*	*	*	Q_d
$\overline{Q_b}$	1	0	*	*	Q_d
	$\overline{Q_c}$		Q_c		

$$Q_b' = \overline{Q_a}\;\overline{Q_b} \vee \overline{Q_d}\;\overline{Q_b} \vee Q_a Q_b$$

Das KV-Diagramm für Q_c' sieht wie folgt aus:

	$\overline{Q_a}$	Q_a		$\overline{Q_a}$	
$\overline{Q_b}$	*	*	0	*	$\overline{Q_d}$
Q_b	*	*	1	1	$\overline{Q_d}$
Q_b	0	*	*	*	Q_d
$\overline{Q_b}$	1	*	*	*	Q_d
	$\overline{Q_c}$		Q_c		

$$Q_c' = \overline{Q_a}\;\overline{Q_b}\;\overline{Q_c} \vee Q_b Q_c$$

Das KV-Diagramm für Q_d' sieht wie folgt aus:

	$\overline{Q_a}$	Q_a		$\overline{Q_a}$	
$\overline{Q_b}$	*	*	1	*	$\overline{Q_d}$
Q_b	*	*	0	0	$\overline{Q_d}$
Q_b	1	*	*	*	Q_d
$\overline{Q_b}$	0	1	*	*	Q_d
	$\overline{Q_c}$		Q_c		

$$Q'_d = \overline{Q_b}\,\overline{Q_d} \vee Q_bQ_d \vee Q_aQ_d$$

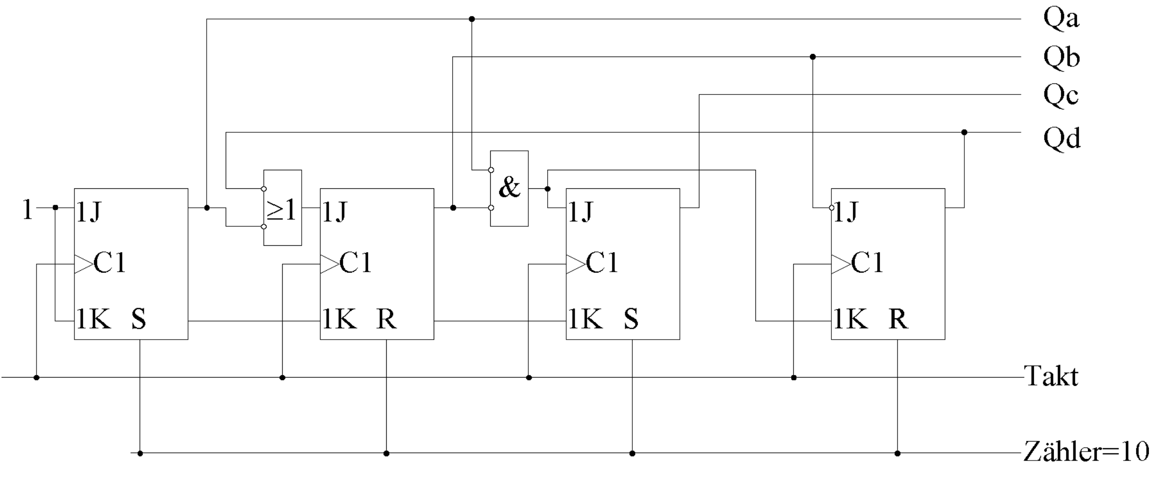
Koeffizientenvergleich mit Gleichung $Q' = J\,\overline{Q} \vee \overline{K}\,Q$ liefert:

$$\begin{aligned}
Q'_a &= 1\,\overline{Q_a} \vee \overline{1}\,Q_a &\Rightarrow\quad & J_a = 1, K_a = 1\\
Q'_b &= (\,\overline{Q_a} \vee \overline{Q_d}\,)\,\overline{Q_b} \vee Q_aQ_b &\Rightarrow\quad & J_b = \overline{Q_a} \vee \overline{Q_d}\,, K_b = \overline{Q_a}\\
Q'_c &= \overline{Q_a}\,\overline{Q_b}\,\overline{Q_c} \vee Q_bQ_c &\Rightarrow\quad & J_c = \overline{Q_a}\,\overline{Q_b}\,, K_c = \overline{Q_b}\\
Q'_d &= \overline{Q_b}\,\overline{Q_d} \vee \overline{\overline{Q_a}\,\overline{Q_b}}\,Q_d &\Rightarrow\quad & J_d = \overline{Q_b}\,, K_d = \overline{Q_a}\,\overline{Q_b}
\end{aligned}$$

Probe:

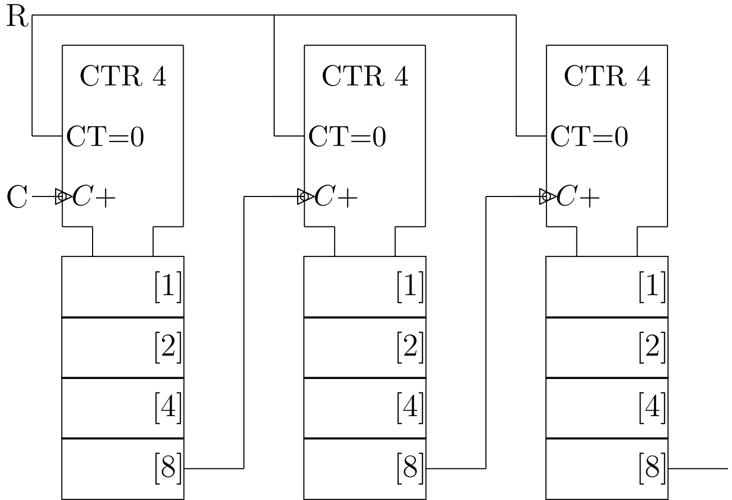
Zahl	Q_d	Q_c	Q_b	Q_a	J_d	K_d	J_c	K_c	J_b	K_b	J_a	K_a	Q'_d	Q'_c	Q'_b	Q'_a
5	0	1	0	1	1	0	0	1	1	0	1	1	1	0	1	0
6	0	1	1	0	0	0	0	0	1	1	1	1	0	1	0	1
7	0	1	1	1	0	0	0	0	1	0	1	1	0	1	1	0
8	1	0	0	0	1	1	1	1	1	1	1	1	0	1	1	1
9	1	0	0	1	1	0	0	1	0	0	1	1	1	0	0	0
10	1	0	1	0	0	0	0	0	1	1	1	1	1	0	0	1

Folgende Schaltung (mit Logik zur Einstellung auf 10 über die Setz- und Rücksetzeingänge der Flipflops) realisiert den Zähler:



Kaskadierung von Zählern

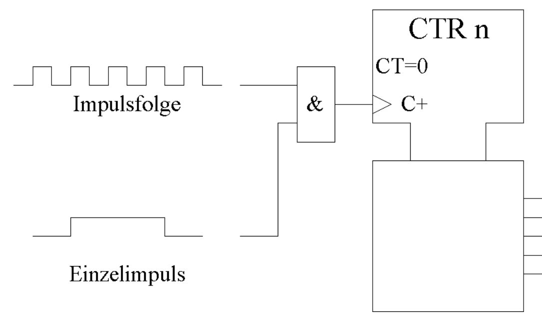
Anmerkung: Zähler können dadurch asynchron kaskadiert werden, indem man den höchstwertigen Ausgang einer Zählstufe mit dem Takteingang der nächsten Stufe verbindet.



In der Praxis wird bei synchronem Vorwärtszähler eine Kaskadierung dadurch erreicht, daß die Konjunktion sämtlicher Stellenausgänge als eigener Ausgang herausgeführt wird. Allerdings müssen dazu die Vorbereitungseingänge der ersten Zählstufe des folgenden Zählers zugänglich sein.

Anmerkung: Anwendungen von Zählern:

- Stückzählung
- Frequenzteiler (durch jede ganze Zahl ist möglich)
- Frequenz- oder Zeitmessung; dabei wird entweder eine Impulsfolge unbekannter Frequenz mit einem Impuls bekannter Länge (z.B. 1s) UND-verknüpft oder ein Impuls unbekannter Länge durch eine Impulsfolge mit fester Frequenz gemessen.



3.2.5 Schieberegister

Definition 3.55 (Schieberegister)
Schieberegister sind Speicherschaltungen mit kettenförmig angeordneten getakteten Flipflops (Speicherzellen). Die Information in jeder Speicherzelle wird durch einen Taktimpuls auf der gemeinsamen Taktleitung in eine benachbarte Speicherzelle verschoben.
Man unterscheidet Schieberegister nach:

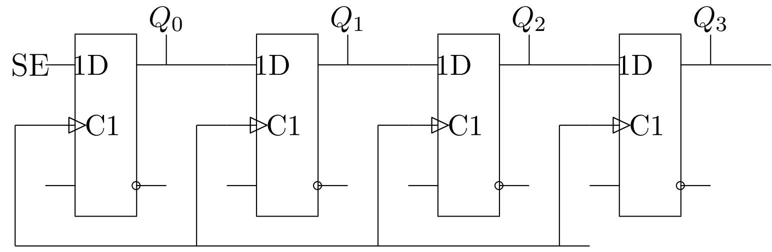
Schieberichtung: links/rechts

Art der Eingabe: seriell oder parallel

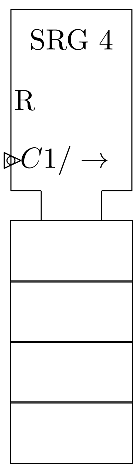
Art der Ausgabe: seriell oder parallel

Zellenzahl

Beispiel 3.56 (Rechtsschieberegister mit serieller Eingabe und Ausgabe)



Anmerkung: DIN-Symbol für Schieberegister:

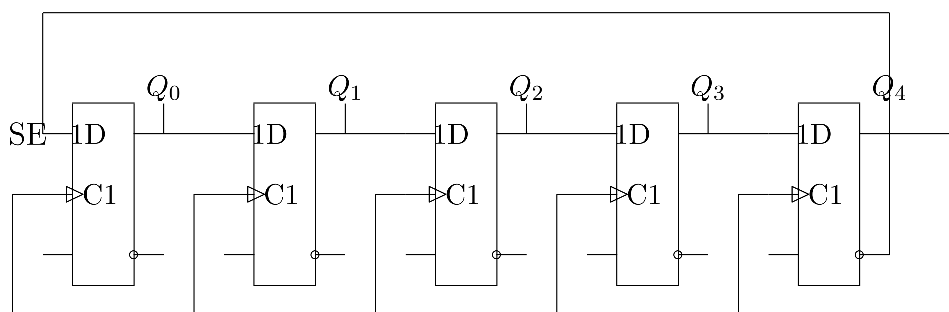


Schieberegisteranwendungen

Anmerkung: Schieberegister haben u.a. folgende Anwendungen:

- serielle Datenübertragung
- Rechenoperationen (Multiplikation/Division mit 2^k)
- Zähler über Rückkopplungen
- Pseudozufallszahlengeneratoren

Beispiel 3.57 (Johnson-Ringzähler)
Für einen 1-aus-10-Zähler braucht man bei einer Simpelimplementierung 10 Flipflops (Ringschieberegister mit einem gesetzten Bit).
Der Johnsonzähler braucht 5 und Zusatzlogik.



Zählfolge:

Zahl	Q'_5	Q'_4	Q'_3	Q'_2	Q'_1
0	0	0	0	0	0
1	0	0	0	0	1
2	0	0	0	1	1
3	0	0	1	1	1
4	0	1	1	1	1
5	1	1	1	1	1
6	1	1	1	1	0
7	1	1	1	0	0
8	1	1	0	0	0
9	1	0	0	0	0

Dekodierung der Zahlen jeweils durch Konjunktion mit zwei Eingängen (z.B. $7 = \overline{Q_2}\, Q_3$).

Kapitel 4

Entwurf von Schaltungen mit endlichen Automaten

4.1 Rekapitulation der Automatentheorie

4.1.1 Allgemeines

Definition 4.1 (Endlicher Automat mit Ausgabe)
Ein *endlicher Automat mit Ausgabe* ist charakterisiert durch:

Σ_I, Σ_O: Ein- und Ausgabezeichenmengen;

Z: Menge von *Zuständen*;

$Z_{\mathcal{F}}$: Menge von *Endzuständen* (finale Zustände);

$z_0 \in Z$: Startzustand des Automaten;

δ: Zustandsübergangsfunktion; $\delta \in [Z \times \Sigma_I \rightarrow Z]$;

λ: Ausgabefunktion; $\lambda \in [Z \times \Sigma_I \rightarrow \Sigma_O]$;

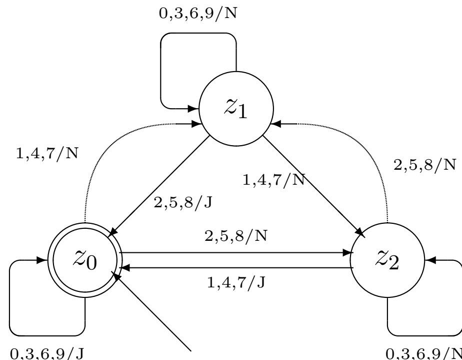
δ und λ werden oft in der *Zustandsübergangstabelle* zusammengefaßt.

Definition 4.2 (Zustandsdiagramm)
Ein endlicher Automat mit Ausgabe kann über ein *Zustandsdiagramm* dargestellt werden. Ein Zustandsdiagramm ist ein gerichteter Graph, bei dem jeder Knoten einem Zustand und jede Kante einem Zustandübergang entspricht. Eine Kante zwischen z_a und z_b ist erstens markiert mit den Eingaben $x_1, \ldots, x_k \in \Sigma_I$, für die $\delta(z_a, x_i) = z_b$ ist und zweitens nach einem

Schrägstrich mit der Ausgabe $\lambda(z_a, x_i) \in \Sigma_O$. Der Startzustand ist mit einem eingehenden Pfeil markiert, die Endzustände mit Doppelkreisen.

Anmerkung: Häufig werden Zustände im Kontext von Schaltungen mit Primtermen und Zustandsübergänge mit Disjunktionen von Primtermen markiert.

Beispiel 4.3 (endlicher Automat)
Es soll ein einfacher Automat für die Bestimmung der Teilbarkeit durch 3 einer Zeichenkette aus Dezimalziffern erstellt werden. Ausgabe des Automaten soll eine Buchstabe sein $\in \{J, N\}$ sein, der besagt, ob die bisher gelesene Zahl durch 3 teilbar ist;
Realisierung: Die Teilbarkeit durch 3 ist gegeben, wenn die Quersumme bei Teilung durch 3 den Rest 0 ergibt. Der Automat merkt sich in seinem Zustand den bisherigen Rest; bei Einlesen eines weiteren Zeichens wird der Zustand entsprechend geändert.



Formale Beschreibung des Automaten:

$$\Sigma_I = \{0,1,2,3,4,5,6,7,8,9\}$$
$$\Sigma_O = \{J,N\}$$
$$Z = \{z_0, z_1, z_2\}$$
$$\text{Startzustand} = z_0$$

δ, λ =

$z \in Z$	$x \in \Sigma_I$	$\delta(z,x)$	$\lambda(z,x)$
z_0	0,3,6,9	z_0	J
z_0	1,4,7	z_1	N
z_0	2,5,8	z_2	N
z_1	0,3,6,9	z_1	N
z_1	1,4,7	z_2	N
z_1	2,5,8	z_0	J
z_2	0,3,6,9	z_2	N
z_2	1,4,7	z_0	J
z_2	2,5,8	z_1	N

Beispielrechnungen für 385 und 17962:

Ausgabe	Zustand	Eingabe
	z_0	$\underline{3}85$
J	z_0	$\underline{8}5$
JN	z_2	$\underline{5}$
JNN	z_1	
	z_0	$\underline{1}7962$
N	z_1	$\underline{7}962$
NN	z_2	$\underline{9}62$
NNN	z_2	$\underline{6}2$
NNNN	z_2	$\underline{2}$
NNNNN	z_1	

Definition 4.4 (Moore- und Mealy-Automat)
Automaten, bei denen die Ausgabe nur vom vorhergehenden Zustand abhängt, heißen *Moore-Automaten* (d.h. $\forall x, y \in \Sigma_I, z \in Z : \lambda(z,x) = \lambda(z,y)$). Automaten ohne diese Eigenschaft heißen *Mealy-Automaten.*
Alternative Definition: ... bei denen die Ausgabe nur vom Ziel-Zustand abhängt...

Mealy-Automat

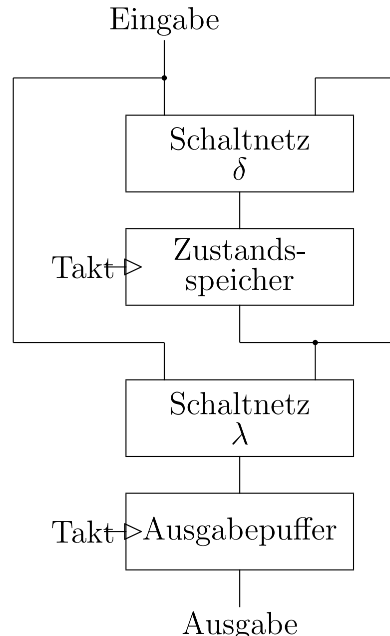
Eingabe
δ
Zustands-
speicher
Takt
λ
Ausgabe

Moore-Automat

Eingabe
δ
Zustands-
speicher
Takt
λ
Ausgabe

Die flankengesteuerten D-Flipflops vor dem Schaltnetz für λ sorgen dafür, daß die Ausgabe nicht vom aktuellen Zustand, sondern vom Zustand *vor dem letzten Zustandsübergang* abhängt, wie es bei endlichen Automaten notwendig ist. In der Praxis gibt es meist zwei Techniken, den Entwurf zu gestalten:

1. Man speichert nicht den letzten Zustand mit D-Flipflops vor dem Schaltnetz für λ, sondern puffert die Ausgabe des Schaltnetzes λ mit einem D-Flipflop:



Vorteil: Änderungen der Eingabe wirken sich erst bei der nächsten Taktung aus.

2. Man baut das Schaltnetz λ so, daß die Ausgabe von Eingabe und *aktuellem* Zustand abhängt (in unserer Zeichnung wären der Ausgang des Zustandsspeichers und der Eingang von λ direkt verbunden). Dann werden keine weiteren Speicher neben dem Zustandsspeicher benötigt. Nachteil: Änderungen am Eingang schlagen auf die Ausgabe durch und es wird strenggenommen kein endlicher Automat implementiert (da die Ausgabe vom aktuellen Zustand abhängt!).

Anmerkung: Der Automat des Beispiels ist ein Mealy-Automat.

4.1.2 Zustandsminimierung

Anmerkung: Für die Realisierung ist es sinnvoll, einen Automat mit der minimalen Zahl von Zuständen zu finden.

Definition 4.5 (Unterscheidbarkeit von Zuständen bzgl. Zustandsmenge)
Zwei Zustände $z_1, z_2 \in Z$ sind unterscheidbar bezüglich einer Zustandsmenge $M \subseteq Z$, wenn

a) $z_1 \in Z_{\mathcal{F}}$ und $z_2 \notin Z_{\mathcal{F}}$ oder

b) es gibt ein $x \in \Sigma_I$ mit $\delta(z_1, x) \in M$ und $\delta(z_2, x) \notin M$ oder

c) es gibt ein $x \in \Sigma_I$ mit $\lambda(z_1, x) \neq \lambda(z_2, x)$.

Definition 4.6 (Partition einer Menge)
Eine *Partition P einer Menge M* ist eine Menge mit folgenden Eigenschaften:

1. Die Elemente von P sind Teilmengen von M, d.h. $p \in P \Longrightarrow p \subseteq M$.
2. Alle Elemente von P ergeben zusammen die Menge M, d.h. $\bigcup_{p \in P} p = M$.
3. Zwei Elemente von P haben keine gemeinsamen Elemente, d.h. $p_i, p_j \in P, p_i \neq p_j \Longrightarrow p_i \cap p_j = \emptyset$.

Eine Partition definiert eine Äquivalenzrelation auf den Elementen von M.

Erklärung 4.7 (Bestimmung des minimalen Automaten)
Der minimale Automat wird wie folgt bestimmt: Es wird eine Partition von den Zuständen bestimmt. Anfangs hat die Partition als Elemente die Menge der Endzustände und die Menge der Nichtendzustände. Danach werden

Elemente der Partition aufgeteilt, sofern sich in ihnen Zustände befinden, die bezüglich eines anderen Partitionselementes unterscheidbar sind. Dies passiert solange, bis keine Aufteilung mehr möglich ist. Die Elemente der Partition sind die neuen Zustände des Automaten. Die Übergänge ergeben sich aus den Übergängen zwischen den Zuständen.

```
P : Partition von Zustand;   M, M1, MA, MB : Menge von Zustand;
z1, z2 : Zustand;   x : Eingabezeichen;   aufgeteilt : Boolean;
BEGINN
    P:={{ersterZustand,...,letzterZustand}};
    ≪Teile P nach Endzuständen und Nichtendzuständen auf≫
    WIEDERHOLE
        aufgeteilt:=FALSCH;
        FüR x:=erstesEingabezeichen BIS letztesEingabezeichen
            FüR z1:=ersterZustand BIS letzterZustand
                M:=diejenige Menge aus P mit Element z1;  M1:={};
                FüR z2:=erstesElement(M) BIS letztesElement(M)
                    MA:=diejenige Menge aus P mit Element δ(z1,x);
                    MB:=diejenige Menge aus P mit Element δ(z2,x);
                    WENN MA≠MB ODER λ(z1,x) ≠ λ(z2,x) DANN
                        aufgeteilt:=WAHR;  M1:=M1∪{z2};
                    ENDE-WENN
                ENDE-FüR
                P:=P-{M};  M:=M-M1;  P:=P∪{M,M1};
            ENDE-FüR
        ENDE-FüR
    BIS aufgeteilt=FALSCH;
ENDE
```

Beispiel 4.8 (Zustandsminimierung)
Es soll ein Automat für die Teilbarkeit durch drei gebaut werden. Als Entwurf liegt ein Automat mit 5 Zuständen $z_a, \ldots, z_e$, bei dem bei jedem Übergang nach z_a und z_e 'J' ausgegeben wird (ansonsten 'N').

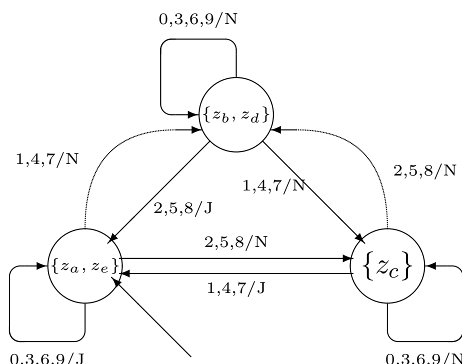
Z/Σ_I	0	1	2	3	4	5	6	7	8	9
z_a	z_e/J	z_b/N	z_c/N	z_e/J	z_d/N	z_c/N	z_e/J	z_b/N	z_c/N	z_a/J
z_b	z_b/N	z_c/N	z_e/J	z_b/N	z_c/N	z_a/J	z_b/N	z_c/N	z_e/J	z_b/N
z_c	z_c/N	z_a/J	z_b/N	z_c/N	z_e/J	z_b/N	z_c/N	z_a/J	z_b/N	z_c/N
z_d	z_b/N	z_c/N	z_a/J	z_b/N	z_c/N	z_e/J	z_b/N	z_c/N	z_e/J	z_d/N
z_e	z_a/J	z_d/N	z_c/N	z_e/J	z_d/N	z_c/N	z_a/J	z_b/N	z_c/N	z_e/J

Die Minimierung dieses Automaten läuft wie folgt ab:

P=$\{\{z_a, z_b, z_c, z_d, z_e\}\}$, d.h. alle Zustände sind gleichwertig
x='0'; Z1=z_a;
  M=$\{z_a, z_b, z_c, z_d, z_e\}$
  Aufteilung aufgrund unterschiedlicher Ausgabe für Eingabe '0':
  M1=$\{z_b, z_c, z_d\}$, M=$\{z_a, z_e\}$
  ansonsten kein weitere Aufteilung möglich
P=$\{\{z_a, z_e\}, \{z_b, z_c, z_d\}\}$
x='0'; Z1=z_b;
  M=$\{z_b, z_c, z_d\}$
  Aufteilung aufgrund unterschiedlicher Ziele für Eingabe '0':
  M1=$\{z_c\}$
  M=$\{z_b, z_d\}$
  ansonsten kein weitere Aufteilung möglich
P=$\{\{z_a, z_e\}, \{z_b, z_d\}, \{z_c\}\}$
keine weitere Aufteilung möglich, d.h. fertig

Das heißt, daß die Zustände z_a und z_e sowie z_b und z_d nicht unterschieden werden können. Der neue Automat hat drei Zustände $ZP := \{z_a, z_e\}$, $ZQ := \{z_b, z_d\}$ und $ZR := \{z_c\}$ sowie folgende Zustandsübergangstabelle:

Startzustand/Eingabe	0,3,6,9	1,4,7	2,5,8
ZP	ZP/J	ZQ/N	ZR/N
ZQ	ZQ/N	ZR/N	ZP/J
ZR	ZR/N	ZP/J	ZQ/N



Als vereinfachter Automat ergibt sich also nach Umbenennung der aus unserem ersten Beispiel.

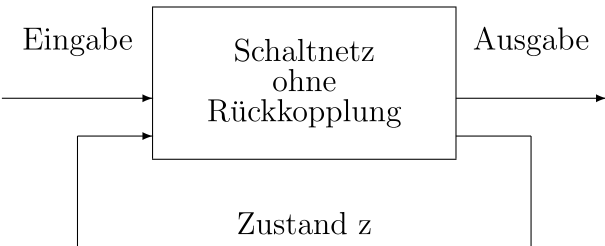
4.2 Analyse und Synthese von Schaltwerken mit Automaten

4.2.1 Analyse

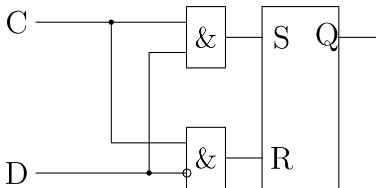
Bei der Analyse ist eine Schaltung vorgegeben und es soll die Funktion dieser Schaltung in Form eines endlichen Automaten beschrieben werden.

Analyse asynchroner Schaltwerke

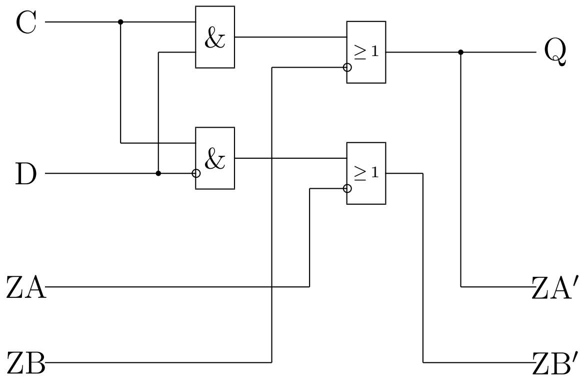
Anmerkung: Bei asynchronen Schaltwerken wirken Rückkopplungen in der Schaltung unmittelbar. Diese Rückkopplungen können so betrachtet werden, daß sie einen Zustand definieren. Jede Änderung des Zustands wirkt sich wiederum auf den Zustand aus. Eine asynchrone Schaltung ist nur dann stabil, wenn für Eingabe e und aktuellen Zustand ein Nachfolgezustand z entsteht, für den $\delta(z, e) = z$ gilt. Dann kann nur eine Änderung der Eingänge eine Änderung des Zustands und der Ausgänge bewirken.



Erklärung 4.9 (Analyse eines zustandsgesteuerten D-Flipflops)
Ein zustandsgesteuertes D-Flipflop kann wie folgt aussehen:



Wenn man das Flipflop durch ODER-Gatter mit negierten Eingängen ersetzt und umzeichnet ergibt sich als das Schaltnetz ohne Rückkopplungen folgendes:



Als Schaltfunktionen ergibt sich:

$$\begin{aligned} Y = ZA' &= \overline{ZA} \vee (C \wedge \overline{D}) \\ ZB' &= \overline{ZB} \vee (C \wedge D) \end{aligned}$$

Man kann die Eingabe- und Zustandsvariablen wie folgt auf Eingabezeichen und Zustände abbilden:

C	D	Zeichen
0	0	00
0	1	01
1	0	10
1	1	11

ZA	ZB	Zustand
0	0	z_{00}
0	1	z_{01}
1	0	z_{10}
1	1	z_{11}

Dann ergibt sich als Zustandsübergangstabelle:

Zustand	00	01	10	11
z_{00}	$z_{11}?/1$	$z_{11}?/1$	$z_{11}?/1$	$z_{11}?/1$
z_{01}	$z_{01}/0$	$z_{01}/0$	$z_{01}/0$	$z_{11}/1$
z_{10}	$z_{10}/1$	$z_{10}/1$	$z_{11}/1$	$z_{10}/1$
z_{11}	$z_{00}?/0$	$z_{00}?/0$	$z_{01}/0$	$z_{10}/1$

Die Zustände z_{00} und z_{11} sind beide instabil (nach obigem Kriterium). Bei idealer Schaltung ist eine Oszillation zwischen diesen Zuständen möglich; in der Praxis fällt die Schaltung in einen Vorzugszustand, d.h. die Übergänge mit Fragezeichen können anders aussehen.
Auffällig ist, daß beim Umschalten zwischen den beiden Speicherzuständen z_{01} und z_{10} und umgekehrt der kurzzeitig der instabile Zustand z_{11} durchlaufen wird (transienter Zustand).

Anmerkung: In komplexeren Systemen treten bei asynchronen Schaltwerken die eben beschriebenen Probleme (instabile bzw. transiente Zustände) häufig auf. Dort sind daher eher synchrone Schaltwerke gebräuchlich.

Analyse synchroner Schaltwerke

Anmerkung: In synchronen Schaltwerken erfolgt die Übernahme des Zustands in den Zustandsspeicher taktgesteuert. Wenn die Taktung langsam genug erfolgt, kann das Schaltnetz den Nachfolgezustand korrekt bestimmen. Transiente Zustände werden von den Ausgängen durch einen getakteten Ausgangspuffer abgeschottet.

Erklärung 4.10 (Analyse eines Schaltung mit JK-Flipflops)
Folgende Schaltung sei gegeben:

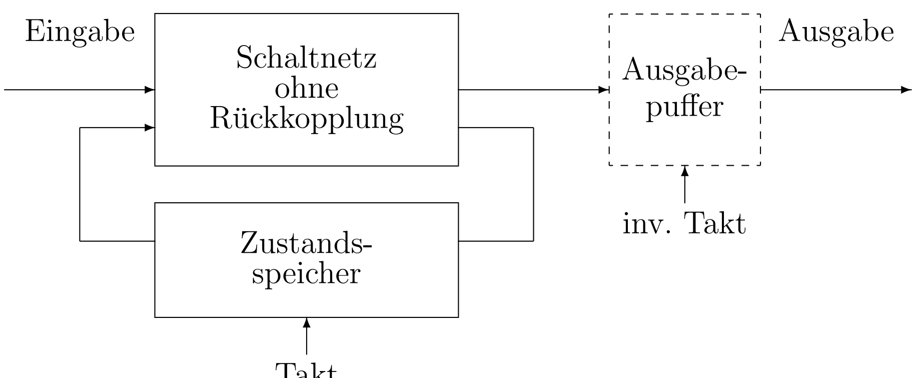
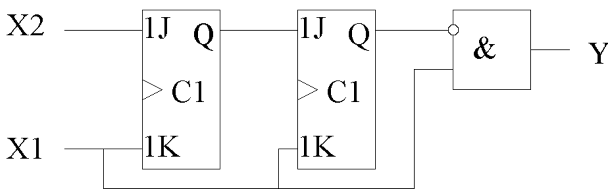


Abbildung 4.1: Schaltwerk mit asynchroner Ausgabe bzw. synchroner Ausgabe (gestrichelt)



Die Gleichungen für die Vorbereitungseingänge der Flipflops und den Ausgang lauten:

$$J2 = X2, K2 = X1, J1 = Q2, K1 = X1, Y = \overline{Q1}\, X1$$

Man kann die Eingabe- und Zustandsvariablen wie folgt auf Eingabezeichen und Zustände abbilden:

X1	X2	Zeichen
0	0	00
0	1	01
1	0	10
1	1	11

Q1	Q2	Zustand
0	0	z_{00}
0	1	z_{01}
1	0	z_{10}
1	1	z_{11}

Dann ergibt sich als Zustandsübergangstabelle:

Zustand	00	01	10	11
z_{00}	$z_{00}/0$	$z_{01}/0$	$z_{00}/1$	$z_{01}/1$
z_{01}	$z_{11}/0$	$z_{11}/0$	$z_{10}/1$	$z_{10}/1$
z_{10}	$z_{10}/0$	$z_{11}/0$	$z_{00}/0$	$z_{01}/0$
z_{11}	$z_{11}/0$	$z_{11}/0$	$z_{00}/0$	$z_{00}/0$

Synthese synchroner Schaltwerke

Erklärung 4.11 (Vorgehen zur Synthese synchroner Schaltwerke)
Bei der Synthese eines Schaltwerks ist ein endlicher Automat gegeben und es wird ein synchrones Schaltwerk gesucht, das diesen Automaten realisiert. Man geht dazu wie folgt vor:

1. Die Zahl der Zustände des Automaten wird minimiert.
2. Sämtliche Eingabezeichen, Ausgabezeichen und Zustände werden binär kodiert.
3. Auf der Basis der Zustandsübergangstabelle werden die Funktionen für sämtliche Zustandsvariablen und Ausgangsvariablen (jeweils in Abhängigkeit von den vorhergehenden Zustandsvariablen und den Eingängen) bestimmt. Bei einer Realisierung mit Flipflops werden statt Funktionen für Zustandsvariablen Funktionen für die Vorbereitungseingänge bestimmt.

Anmerkung: Wie bereits oben erwähnt, gibt es bei synchronen Schaltwerken mehrere Techniken um eine korrekte Ausgabe zu gewährleisten:

1. Pufferung der Ausgabe synchron mit der Übernahme des neuen Zustands in den Zustandsspeicher oder
2. Entwurf des Ausgabeschaltnetzes so, daß die Ausgabe vom aktuellen Zustand und der Eingabe abhängt.

Wir werden beide Techniken diskutieren.

Beispiel 4.12 (Realisierung eines endlichen Automaten)
Wir realisieren unser Standardbeispiel des Automaten, der die Teilbarkeit durch drei prüft. Um das Schaltnetz nicht zu kompliziert zu machen, lassen wir als Eingabezeichen nur 0, 1 und 2 zu. Die Zustandsübergangstabelle ist also wie folgt:

	Zustand		
Eingabezeichen	z_0	z_1	z_2
0	z_0/J	z_1/N	z_2/N
1	z_1/N	z_2/N	z_0/J
2	z_2/N	z_0/J	z_1/N

Wir gehen schrittweise wie oben beschrieben vor:

1. Eine Minimierung des Automaten liefert denselben Automaten.
2. Wir kodieren die Eingabezeichen, Ausgabezeichen und Zustände wie folgt:

Eingabez.	X2	X1
0	0	0
1	0	1
2	1	0

Ausgabez.	Y
N	0
J	1

Zstd.	Z2	Z1
z_0	0	0
z_1	0	1
z_2	1	0

Der Zustand $(Z1 = 1, Z2 = 1)$ und das Zeichen $(X1 = 1, X2 = 1)$ sind nicht zulässig und werden bei den Übergängen nicht berücksichtigt.

3. Die Übergangstabelle sieht nun wie folgt aus:

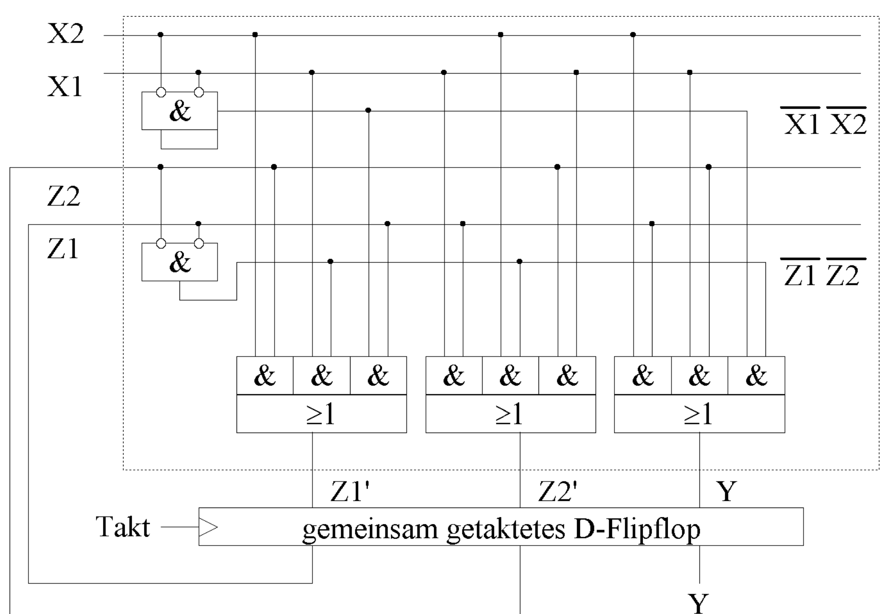
Eingabe		Zustand		Folgezustand		Ausgabe
X2	X1	Z2	Z1	Z2′	Z1′	Y
0	0	0	0	0	0	1
0	0	0	1	0	1	0
0	0	1	0	1	0	0
0	0	1	1	*	*	*
0	1	0	0	0	1	0
0	1	0	1	1	0	0
0	1	1	0	0	0	1
0	1	1	1	*	*	*
1	0	0	0	1	0	0
1	0	0	1	0	0	1
1	0	1	0	0	1	0
1	0	1	1	*	*	*
1	1	0	0	*	*	*
1	1	0	1	*	*	*
1	1	1	0	*	*	*
1	1	1	1	*	*	*

Wir haben zwei mögliche Realisierungsvarianten: Flankengetriggerte D-Flipflops zur Pufferung von Zustand und Ausgabe oder ein Schaltwerk aus JK-Flipflops.

(a) Zur Pufferung der Zustandsvariablen und des Ausgangs benötigen wir drei D-Flipflop mit gemeinsamem Takteingang. Aus der obigen Tabelle ergeben sich folgende Schaltfunktionen:

$$\begin{aligned} Z2' &= X2\,\overline{Z2}\,\overline{Z1} \vee \overline{X2}\,\overline{X1}\,Z2 \vee X1Z1 \\ Z1' &= X1\,\overline{Z2}\,\overline{Z1} \vee \overline{X2}\,\overline{X1}\,Z1 \vee X2Z2 \\ Y &= X2Z1 \vee X1Z2 \vee \overline{X2}\,\overline{X1}\,\overline{Z2}\,\overline{Z1} \end{aligned}$$

Die Schaltung sieht dann wie folgt aus (das Schaltnetz ist gestrichelt):



Natürlich lassen sich die Schaltnetze für λ und δzusammenfassen und beispielsweise durch ein ROM oder PLD mit vier Eingängen und drei Ausgängen realisieren. An externer Beschaltung sind dann nur noch die D-Flipflops nötig.

(b) Wenn man (z.B. für Zähler) JK-Flipflops verwenden will, muß man die Beschaltung der Vorbereitungseingänge aus der Übergangstabelle bestimmen.

Man kann so vorgehen wie in Kapitel 3.2.4 durch Koeffizientenvergleich oder pragmatischer. Die Überlegung ist wie folgt:
In einer JK-Flipflop-Stufe können bei einer Taktung vier Übergänge des Stufenausgangs vorkommen: 0 bleibt bestehen, $0 \rightarrow 1$, $1 \rightarrow 0$ und 1 bleibt bestehen. Für jeden dieser Übergänge müssen die Vorbereitungseingänge J und K folgendermaßen beschaltet sein:

erwünschter Übergang am Stufenausgang	J	K
$0 \rightarrow 0$	0	*
$0 \rightarrow 1$	1	*
$1 \rightarrow 0$	*	1
$1 \rightarrow 1$	*	0

Aufgrund der zwei Zustandsvariablen werden zwei JK-Flipflop-Stufen benötigt. Aus der Zustandsübergangstabelle ergibt sich folgende Tabelle für die Vorbereitungseingänge:

Eingabe		Zustand		Folgezustand		Stufe 2			Stufe 1		
X2	X1	Z2	Z1	Z2′	Z1′	Übergang 2	J2	K2	übergang 1	J1	K1
0	0	0	0	0	0	$0 \rightarrow 0$	0	*	$0 \rightarrow 0$	0	*
0	0	0	1	0	1	$0 \rightarrow 0$	0	*	$1 \rightarrow 1$	*	0
0	0	1	0	1	0	$1 \rightarrow 1$	*	0	$0 \rightarrow 0$	0	*
0	0	1	1	*	*	*	*	*	*	*	*
0	1	0	0	0	1	$0 \rightarrow 0$	0	*	$0 \rightarrow 1$	1	*
0	1	0	1	1	0	$0 \rightarrow 1$	1	*	$1 \rightarrow 0$	*	1
0	1	1	0	0	0	$1 \rightarrow 0$	*	1	$0 \rightarrow 0$	0	*
0	1	1	1	*	*	*	*	*	*	*	*
1	0	0	0	1	0	$0 \rightarrow 1$	1	*	$0 \rightarrow 0$	0	*
1	0	0	1	0	0	$0 \rightarrow 0$	0	*	$1 \rightarrow 0$	*	1
1	0	1	0	0	1	$1 \rightarrow 0$	*	1	$0 \rightarrow 1$	1	*
1	0	1	1	*	*	*	*	*	*	*	*
1	1	0	0	*	*	*	*	*	*	*	*
1	1	0	1	*	*	*	*	*	*	*	*
1	1	1	0	*	*	*	*	*	*	*	*
1	1	1	1	*	*	*	*	*	*	*	*

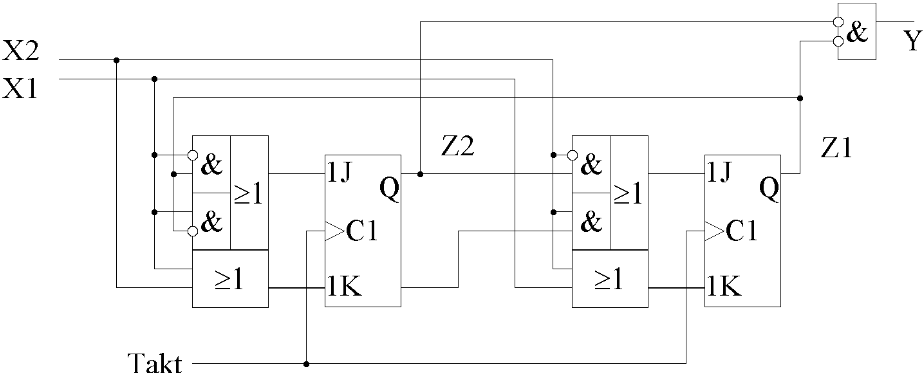
Aus dieser Tabelle ergeben sich folgende Gleichungen für J1, K1, J2 und K2:

$$\begin{aligned} J2 &= X1Z1 \vee X2\,\overline{Z1} \\ K2 &= X2 \vee X1 \\ J1 &= X2Z2 \vee X1\,\overline{Z2} \\ K1 &= X2 \vee X1 \end{aligned}$$

Wenn man Y in Abhängigkeit des *aktuellen* Zustands betrachtet (also von Z2′ und Z1′), so ergibt sich

$$Y = \overline{Z2'}\;\overline{Z1'}$$

(Dies ist die Implementierungsvariante von oben bei der die Ausgänge vom aktuellen Zustand der Schaltung abhängen.) Insgesamt liefert das folgende Schaltung:


X2
X1
&
&
≥1
≥1
1J
Q
C1
1K
Z2
&
&
≥1
≥1
1J
Q
C1
1K
Z1
&
Y
Takt

Kapitel 5

Technische Realisierung digitaler Schaltungen

Anmerkung: Folgende Trends bei der Realisierung digitaler Schaltungen sind auszumachen:

- Schaltungen praktisch nur noch in Form integrierter Schaltungen
- Höchstintegration mit > 500 000 Transistoren
- hohe Integrationsdichten in MOS-Technologie

5.1 Grundlagen

5.1.1 Kenngrößen digitaler integrierter Schaltungen

Für die Beurteilung digitaler Schaltungen sind folgende Parameter interessant (ohne Priorisierung):

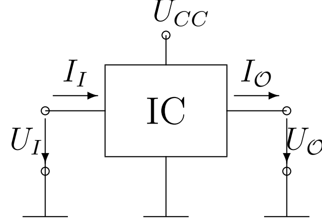
- Versorgungsspannungen
- Logikpegel
- Störspannungsabstand (statisch und dynamisch)
- Eingangs- und Ausgangslastfaktor für Gatter
- Verlustleistung pro Gatter

- Gatterlaufzeit
- Verzögerungs-Leistungsprodukt
- Grenzfrequenz
- Betriebstemperaturbereich

Anmerkung: Im folgenden verwendete Symbole für technische Größen:

U_{CC}	Versorgungsspannung
U_I	Eingangsspannung (Input Voltage)
U_{IH}	Eingangsspannung bei H-Pegel
U_{IL}	Eingangsspannung bei L-Pegel
$(U_{IH})_{min}$	minimale Eingangsspannung bei H-Pegel
$(U_{IL})_{max}$	maximale Eingangsspannung bei L-Pegel
$U_{\mathcal{O}}$	Ausgangsspannung (Output Voltage)
$U_{\mathcal{O}H}$	Ausgangsspannung bei H-Pegel
$U_{\mathcal{O}L}$	Ausgangsspannung bei L-Pegel
$(U_{\mathcal{O}H})_{min}$	minimale Ausgangsspannung bei H-Pegel
$(U_{\mathcal{O}L})_{max}$	maximale Ausgangsspannung bei L-Pegel
I_I	Eingangsstrom (Input Current)
I_{IH}	Eingangsstrom bei H-Pegel
$(I_{IH})_N$	Eingangsstrom des Einheitsgatters bei H-Pegel
I_{IL}	Eingangsstrom bei L-Pegel
$(I_{IL})_N$	Eingangsstrom des Einheitsgatters bei L-Pegel
$I_{\mathcal{O}}$	Ausgangsstrom (Output Current)
$I_{\mathcal{O}H}$	Ausgangsstrom bei H-Pegel
$I_{\mathcal{O}L}$	Ausgangsstrom bei L-Pegel
$(I_{\mathcal{O}H})_{min}$	minimal zulässiger Ausgangsstrom bei H-Pegel
$(I_{\mathcal{O}L})_{max}$	maximal zulässiger Ausgangsstrom bei L-Pegel
t_{pd}	Schaltzeit (propagation delay)

Anmerkung: Zugrundeliegende Vorstellung:



Lastfaktoren

Definition 5.1 (Lastfaktor)
Die Belastung, die ein Ausgang durch einen nachfolgenden Eingang der gleichen Schaltkreisfamilie erfährt, wird durch *Lastfaktoren* beschrieben. Die Belastung wird auf ein Einheitsgatter normiert.

Definition 5.2 (Eingangslastfaktor)
Der *Eingangslastfaktor* eines Eingangs gibt an, um welchen Faktor die Stromaufnahme größer ist als beim Einheitsgatter derselben Schaltkreisfamilie.
Formal:

$$\begin{aligned}(\mathrm{Fan-In})_H &:= |I_{IH}/(I_{IH})_N| \\ (\mathrm{Fan-In})_L &:= |I_{IL}/(I_{IL})_N| \\ (\mathrm{Fan-In}) &:= \max\{(\mathrm{Fan-In})_H, (\mathrm{Fan-In})_L\}\end{aligned}$$

Definition 5.3 (Ausgangslastfaktor)
Der *Ausgangslastfaktor* eines Ausgangs gibt an, mit wievielen Eingängen von Einheitsgattern derselben Schaltkreisfamilie dieser Ausgang belastet werden darf.
Formal:

$$\begin{aligned}(\mathrm{Fan-Out})_H &:= |I_{\mathcal{O}H}/(I_{\mathcal{O}H})_N| \\ (\mathrm{Fan-Out})_L &:= |I_{\mathcal{O}L}/(I_L)_N| \\ (\mathrm{Fan-Out}) &:= \min\{(\mathrm{Fan-Out})_H, (\mathrm{Fan-Out})_L\}\end{aligned}$$

Anmerkung: Die physikalische Grundlage für diese Festlegungen ist das Kirchhoffsche Gesetz.

Anmerkung: Bei Kombination verschiedener Schaltkreisfamilien können in der Regel keine normierten Lastfaktoren verwendet werden. Hier werden die Eingangsströme addiert und mit den maximalen Ausgangsströmen verglichen.

Störspannungsabstand

Definition 5.4 (Störspannungsabstand)
Der *Störspannungsabstand* ist die Spannung, um die ein Ausgang variieren darf, ohne daß ein angeschlossener Eingang derselben Logikfamilie in den verbotenen Pegelbereich gelangt.

Man unterscheidet den

den statischen Störspannungsabstand (bei Störimpulsen, die länger als die Gatterlaufzeit sind) und

den dynamischen Störspannungsabstand (bei extrem kurzen Störimpulsen).

Definition 5.5 (statischer Störspannungsabstand)
Der *statische Störspannungsabstand im schlechtesten Fall* ist definiert als:

$$\begin{aligned} (U_S)_H &:= (U_{\mathcal{O}H})_{min} - (U_{IH})_{min} \\ (U_S)_L &:= (U_{IL})_{max} - (U_{\mathcal{O}L})_{max} \\ U_S &:= \min\{(U_S)_H, (U_S)_L\} \end{aligned}$$

Der *typische statische Störspannungsabstand* ist definiert als:

$$\begin{aligned} (U_{S_{typ}})_H &:= (U_{\mathcal{O}H})_{typ} - U_{TH} \\ (U_{S_{typ}})_L &:= U_{TH} - (U_{\mathcal{O}L})_{typ} \\ U_{S_{typ}} &:= \min\left\{(U_{S_{typ}})_H, (U_{S_{typ}})_L\right\} \end{aligned}$$

Dabei sind ist $(U_{\mathcal{O}H})_{typ}$ die typische Ausgangsspannung bei H-Pegel, $(U_{\mathcal{O}L})_{typ}$ die typische Ausgangsspannung bei L-Pegel und U_{TH} die mittlere Schwellwertspannung (threshold) bei Pegelwechsel zwischen L- und H-Pegel.

$U_{\mathcal{O}}$ Ausgang | Eingang U_I
U_{CC} | U_{CC}
$U_{\mathcal{O}Hmin}$ H-Pegel | H-Pegel
$(U_S)_H$
U_{IHmin}
U_{ILmax}
$(U_S)_L$
$U_{\mathcal{O}Lmax}$ L-Pegel | L-Pegel

Beispiel 5.6 (Werte für Standard TTL-Schaltkreise)

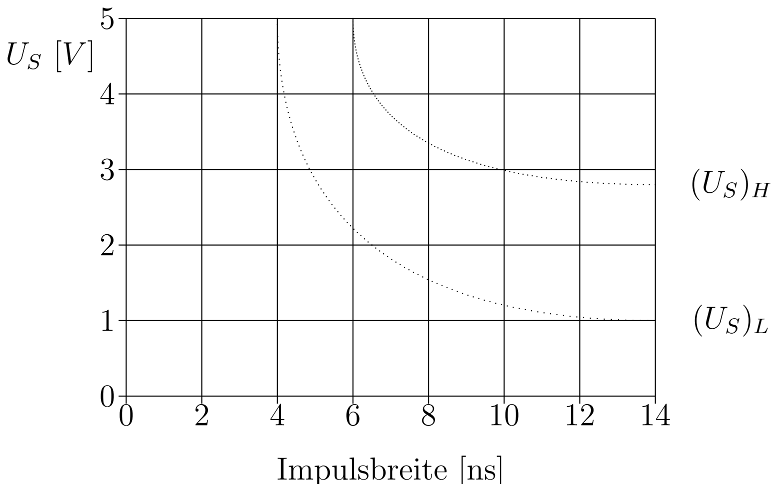
$$\begin{aligned} \text{Worst-Case-Werte} &: (U_{\mathcal{O}H})_{min} = 2,4V;\ (U_{IH})_{min} = 2,0V \\ \text{— " —} &: (U_{\mathcal{O}L})_{max} = 0,4V;\ (U_{IL})_{max} = 0,8V \\ \text{Betriebswerte} &: (U_{\mathcal{O}H})_{typ} = 3,9V;\ (U_{\mathcal{O}L})_{typ} = 0,25V;\ U_{TH} = 1,26V \end{aligned}$$

Die Werte der letzten Zeile sind temperaturabhängig.

Definition 5.7 (dynamischer Störspannungsabstand)
Der *dynamische Störspannungsabstand* kennzeichnet das Verhalten eines digitalen Schaltkreises gegenüber Störimpulsen, deren Impulsbreite kleiner als die Gatterlaufzeit ist. Er ist definiert als die Amplitude eines Impulses mit vorgegebener Breite, die noch keine Änderung des Ausgangspegels hervorruft.

Anmerkung:

- Der dynamische Störspannungsabstand nähert sich asymptotisch ∞ für sehr kurze Impulse und nähert sich asymptotisch dem statischen Störabstand bei Impulsen, deren Breite die Gatterlaufzeit ist.
- Die Werte sind temperaturabhängig.



Schaltzeiten

Definition 5.8 (Anstiegs- und Abfallzeit, Schaltzeit)
Die Anstiegs- und Abfallzeiten eines Gatters sind die Zeiten, die ein Ausgang bei einem wesentlichen steilflankigen Pegelübergang am Eingang benötigt, um einen Pegelübergang von 10% bis 90% des Pegelhubes (oder umgekehrt) durchzuführen.
Die *Schaltzeit* eines Gatters ist die Zeit, die es bei einem wesentlichen Pegelübergang am Eingang benötigt, um am Ausgang einen Übergang durchzuführen. Da normalerweise die Schaltflanken nicht senkrecht sind, wird die

Zeit zwischen je nach Schaltkreisreihe unterschiedlich definierten Standardpegeln (z.B. U_{TH}) gemessen. Diese Schaltzeit wird als t_{pd} bezeichnet (propagation delay).

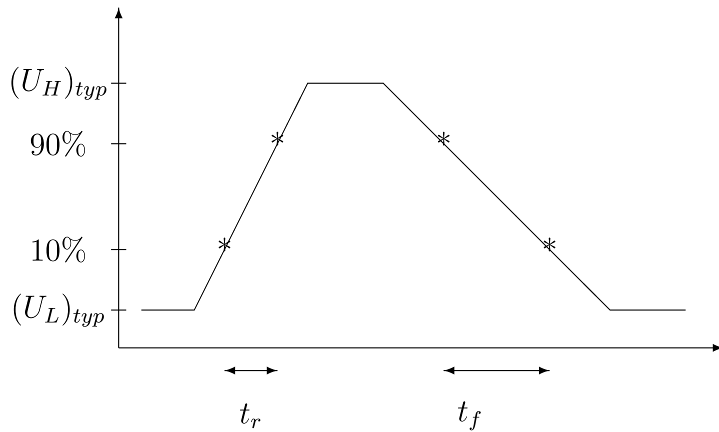


Abbildung 5.1: Anstiegs- und Abfallzeit

Anmerkung: Bei unterschiedlichen Schaltzeiten für Übergänge von L nach H ($(t_{pd})_{LH}$) und umgekehrt ($(t_{pd})_{LH}$) wird der Mittelwert beider Werte genommen.

Anmerkung: Die Schaltzeit hängt neben internen Charakteristika eines Gatters von Ausgangslasten und Beschaltung unbenutzter Eingänge ab.

5.1.2 Grundlagen zu Halbleitern

Allgemeines

Definition 5.9 (Halbleiter)
Halbleiter sind Kristalle aus chemischen Elementen wie Silizium, Germanium, Selen, Tellur oder Verbindungen wie Galliumarsenid oder Zinkselenid. Sie liegen im spezifischen Widerstand zwischen Isolatoren ($10^{10}\Omega cm$) und Metall ($10^{-3}\Omega cm$).

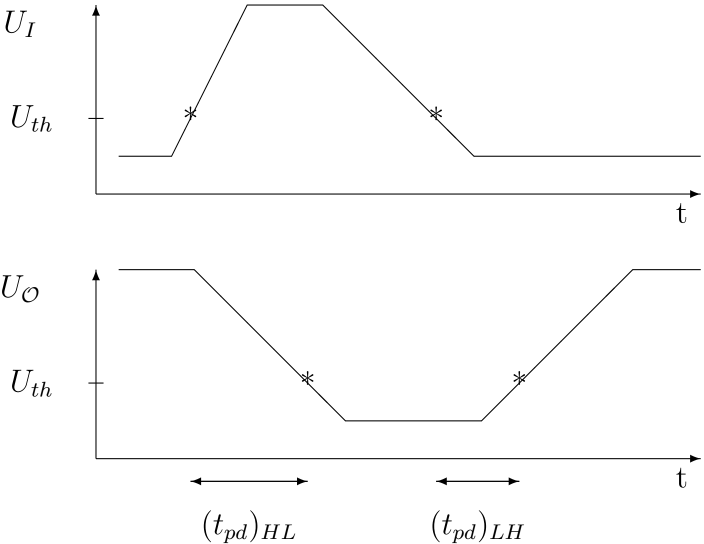


Abbildung 5.2: Schaltzeit

Anmerkung: Metalle sind nicht gut geeignet für elektronische Steuerung, da sie eine hohe Dichte von freien Elektronen besitzen (pro Atom ein freies Elektron) $\Longrightarrow$ um Leitfähigkeit entscheidend zu verringern, müssen viele Elektronen "abgesaugt" werden. Bei Halbleitern gibt es bei entsprechender physikalischer Vorbehandlung erheblich weniger freie Ladungsträger (Orientierung: nur jedes 10000ste Atom besitzt einen freien Ladungsträger).

Anmerkung: Aus der Atomphysik ist bekannt, daß Elektronen nur in bestimmten Energiebereichen existieren können, sogenannte *Bänder*. Bei Halbleitern ist das energetisch niedrigste Band fast völlig besetzt (*Valenzband*), das nächsthöhere praktisch nicht (*Leitungsband*).
Grund: Kristallbindung

Definition 5.10 (Löcher)
Freie Plätze im Valenzband werden als *Löcher* bezeichnet. Sie verhalten sich wie positive Ladungsträger im Leitungsband.
Ein Loch repräsentiert eigentlich alle Elektronen des Valenzbands bis auf ein einziges.

Beispiel 5.11 (Gedankenexperiment zu Löchern)
Bei Anlegen von Spannung wandern Löcher entgegengesetzt zu Elektronen.

Definition 5.12 (n-leitend, p-leitend)
Halbleiter, die vorwiegend Löcher enthalten heißen *p-leitend*, solche mit vorwiegend Elektronen heißen *n-leitend*. Man nennt die vorwiegend vorkommenden Ladungsträger *Majoritätsladungsträger*, die anderen Minoritätsladungsträger.

Beispiel 5.13 (Ermittlung der Ladungsträger)
Man kann feststellen, ob die ein Strom durch Löcher oder durch Elektronen verursacht wird, indem man den Hall-Effekt ausnutzt.

Definition 5.14 (Dotierung)
Einen Prozeß, bei dem durch systematische Verunreinigung Löcher oder freie Elektronen geschaffen werden, heißt *Dotierung*. In Standardhalbleitern (z.B. Silizium) gibt es vier Valenzelektronen, benachbarte Elemente sind Phosphor (5 Elektronen) und Bor (3 Elektronen). Phosphor gibt gerne Elektronen ab, Bor gibt Löcher ab.

Erklärung 5.15 (Kristallstruktur)
Alle Atome versuchen sich so anzuordnen, daß sie sämtliche Plätze ihrer äußersten belegten Schale füllen. $\Longrightarrow$Tetraederstruktur

Anmerkung: Durch Verunreinigungen entstehen zusätzliche "Elektronenfallen" oder "Löcherfallen" durch weitere zusätzliche Energieniveaus.
Da der Dotierungsgrad in der Größenordnung von 1:10000 liegt, muß der Reinheitsgrad erheblich größer sein $\Longrightarrow$Reinheitsfanatismus

Definition 5.16 (Erzeugung von freien Ladungsträgern, Rekombination)
Durch externe Energie (beispielsweise Licht) kann ein Elektron in das Leitungsband gehoben werden. Dabei entsteht auch ein Loch im Valenzband. $\Longrightarrow$Leitfähigkeitserhöhung
Umgekehrt kann ein Elektron aus dem Leitungsband in ein Loch fallen (unter Aussendung eines Lichtquants). Dann spricht man von *Rekombination*.

Halbleiterdioden

Erklärung 5.17 (Raumladungszone)
Bringt man einen p-dotierten und einen n-dotierten Halbleiter zusammen (*pn-Übergang*), diffundieren die freien Ladungsträger in den jeweils anderen Teil. Dadurch ergibt sich eine Potentialbarriere (sogenannte *Raumladungszone*), die weiteres Eindringen von Ladungsträgern verhindert. Die Raumladungszone ist Ursache für eine Spannung im pn-Übergang (*Diffusionsspannung*).

Durch Rekombination wird die Konzentration freier Ladungsträger in der Raumladungszone verringert ($\Longrightarrow$ *Verarmungsgebiet*). Die Teile außerhalb der Raumladungszone heißen *Bahngebiete*.

Erklärung 5.18 (Gleichrichtungswirkung eines pn-Übergangs)
Wenn man an den pn-Übergang eine Spannung anlegt, passiert folgendes:

- *bei höherem Potential am p-Halbleiter* werden die Majoritätsladungsträger zur Raumladungszone getrieben und überwinden sie, wenn die Spannung größer ist als die Diffusionsspannung. In dem anderen Bahngebiet sind sie Minoritätsladungsträger und rekombinieren mit den dortigen Majoritätsladungsträgern. Dies passiert oft. Die verschwundenen Majoritätsladungsträger werden im Bahngebiet über die Spannungsquelle ersetzt.
- *bei niedrigerem Potential am p-Halbleiter* werden die Majoritätsladungsträger von der Raumladungszone abgezogen. Ein Strom kommt nur dadurch zustande, daß im Verarmungsgebiet Ladungsträger (thermisch) generiert werden. Dies passiert selten (im Vergleich zur Rekombination).

Daher fließt im ersten Fall ein großer Rekombinationsstrom, im zweiten Fall ein kleiner Generationsstrom. Der pn-Übergang wirkt also als Gleichrichter (mit Durchlaß- und Sperrichtung).

Bipolarer Transistor

Definition 5.19 (Bipolarer Transistor)
Ordnet man zwischen zwei gleichartig stark dotierten Teilen ein gegenteilig schwach dotierten und dünnen Teil an, so erhält man einen *bipolaren Transistor*. Es gibt also npn- und pnp-Transistoren.

Erklärung 5.20 (Funktion eines npn-Transistors)
Wir bezeichnen die Schichten von Bild XXX (fehlt noch) von oben nach unten mit Kollektor, Basis und Emitter. Zwischen Kollektor und Emitter sei eine Spannung angelegt mit positiven Pol am Kollektor. An der Basis liegt ein Potential zwischen dem von Kollektor und Emitter. Die Basis-Emitter-Diode ist in Durchlaßrichtung, die Basis-Kollektor-Diode in Sperrichtung betrieben.

- Elektronen wandern vom Emitter zur Basis, rekombinieren dort aber nur selten, weil die Basis sehr dünn ist und wenig Löcher hat. Vielmehr

wandern sie weiter in die Schicht des Kollektors und von dort zum Pluspol der Spannungsquelle.
Aus der Basis fließen nur wenig Elektronen ab (im wesentlichen Ersatz für die bei Rekombination verbrauchten Löcher).

- Löcher aus der Basis fließen zum Emitter und rekombinieren dort. Dies führt jedoch nur zu einem kleinen Strom, da die Basis schwach dotiert ist. Daher ist der Gesamtstrom in die Basis klein.

Wird die Basis-Emitterspannung U_{BE} erhöht, werden erheblich mehr Elektronen aus dem Emitter in den Kollektor beschleunigt, aber der Basisstrom erhöht sich nur wenig. Beim Sinken von U_{BE} fließen weniger Elektronen zum Kollektor und ein geringerer Basisstrom. Ist U_{BE} kleiner als die Diffusionsspannung, kommt der Elektronenstrom zum Erliegen.
Betrachtet man den Basisstrom als Eingangsgröße und den Kollektorstrom als Ausgangsgröße, so sind sie (in gewissen Bereichen) proportional.

Anmerkung: Der Unterschied zwischen Kollektor und Emitter ergibt sich lediglich aus den anliegenden Spannungen; auch wenn man Emitter und Kollektor vertauscht, funktioniert der Transistor (*Inversbetrieb*).

Feldeffekttransistor

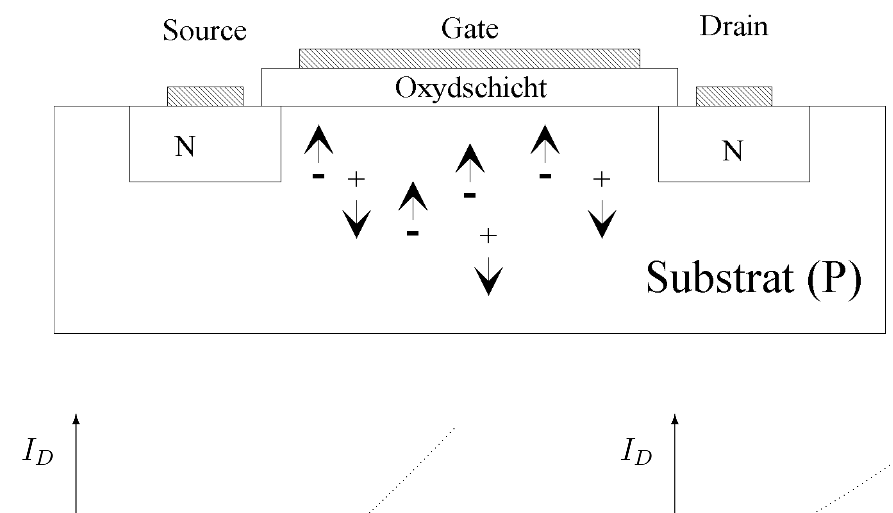
Definition 5.21 (Feldeffekttransistor)
Ein *Feldeffekttransistor* beruht auf der Idee, durch ein elektrisches Feld die Leitfähigkeit eines Widerstands in diesem Feld zu beeinflußen.
Ein FET hat drei Elektroden: Gate, Source und Drain. Das Gate steuert durch eine Spannung zur Source den Strom zwischen Source und Drain.

`Feldeffekt uralt; aber Eindringtiefe indirekt proportional zu Leitfähigkeit des Widerstands`

Erklärung 5.22 (Funktion eines Feldeffekttransistors)
Auf einen dotierten Halbleiter (in unserem Beispiel n-dotiert) wird — isoliert durch eine Zwischenschicht — eine Elektrode angebracht. Wird diese Elektrode positiv geladen, so werden durch Influenz Elektronen in den Halbleiter gezogen ($\Longrightarrow$Verringerung des Widerstands im Halbleiter, da mehr Ladungsträger vorhanden sind).
Der Feldeffekttransistor benötigt nur einen geringen Steuerstrom, da nur die Elektrode zu laden oder zu entladen ist.

Anmerkung: Erstes Konzept eines Feldeffekttransistors stammt von 1928 (Lillienfeld); die Realisierung war erst in den 60er-Jahren möglich.

Erklärung 5.23 (Mögliche Realisierung als n-Kanal-MOSFET)
Bei einem MOSFET (Metal Oxide Semiconductor) sind Source und Drain n-dotiert, der Bereich dazwischen — das Substrat — p-dotiert, das Gate ist aus Metall auf einer isolierenden Siliziumdioxidschicht (daher der Name).
Der Transistor ist "selbstsperrend", weil sich im Kanal die Ladungsträger erst bilden, wenn am Gate eine positive Spannung anliegt, d.h. der Transistor sperrt, wenn $U_{GS} = 0$ ist (Synonym: Anreicherungstyp, Gegensatz: selbstleitend, Verarmungstyp). Wenn $U_{GS} = 0$ ist, dann kann zwischen Source und Drain kein Strom fließen, weil zwei pn-Übergänge dazwischenliegen. Wenn $U_{GS} \gg 0$ ist, werden durch das positive Potential Elektronen aus dem Substrat angezogen, die zunächst mit den Löchern im p-Substrat rekombinieren, danach aber als Ladungsträger dienen können.



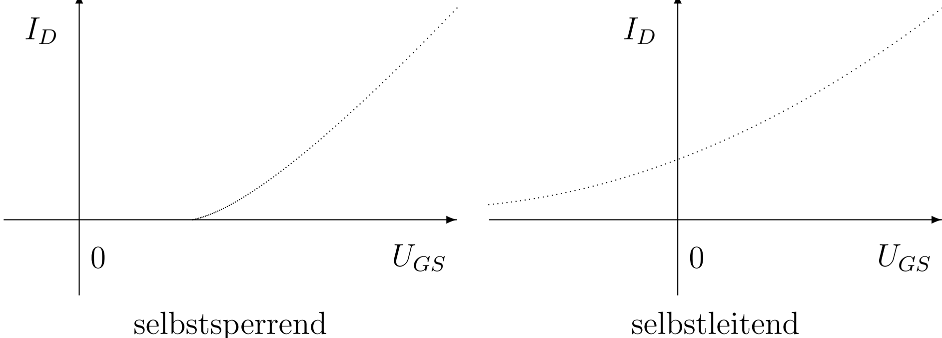


Abbildung 5.3: $U_{GS}-I_D$-Kennlinie eines selbstleitenden und selbstsperrenden n-Kanal-MOSFETs

5.1.3 Schaltkreistechnologien

Übersicht über Schaltkreistechnologien:

IS-Technologien				
Bipolarschaltungen		MOS-Schaltungen		
Übersteuerungstechnik	nichtübersteuerte Technik	statische Schaltungstechnik		dynamische Schaltungstechnik
TTL,I^2L	ECL,STTL	PMOS,NMOS	CMOS	

Transistor-Transistor-Logik

TTL-Logik TTL ist die wichtigste Gruppe bei den bipolaren Schaltkreisen; sie hat die größte Produktpalette.
TTL-Unterfamilien:

Familie	$t_D[ns]$	$P_V[mW]$	Bezeichnung	Jahr
Low-Power	33	1	74LXX	1963
Standard	10	10	74XX	1963
High-Speed	6	22	74LXX	1963
Schottky	3	19	74SXX	1969
Low-Power-Schottky	9	2	74LSXX	1971
Advanced-LS	4	1	74ASXX	1980
Advanced-Schottky	1,7	8,5	74ASXX	1982

Schaltungen in Standard-TTL `R1=4k, R2=1,6k, R3=130, R4=1k`

Erklärung 5.24 (Funktion der Standard-TTL-Schaltung)
Wenn X1 oder X2 auf L-Pegel liegen, wird eine Basis-Emitterdiode von T1 leitend und schaltet L-Pegel an Basis von T2. Damit sperrt T4 und T3 leitet (wegen Strom von R2). Über R3, T3 und D1 liegt der Ausgang Y an H-Pegel. Wenn X1 und X2 auf H-Pegel liegen, arbeitet T1 im Inversbetrieb (d.h. der Emitter fungiert als Kollektor und umgekehrt). An der Basis von T2 liegt H-Pegel. Damit leitet T4 und T3 sperrt (der Emitter wird durch D1 auf höheren Pegel gebracht). Der Ausgang Y liegt über T4 auf L-Pegel.

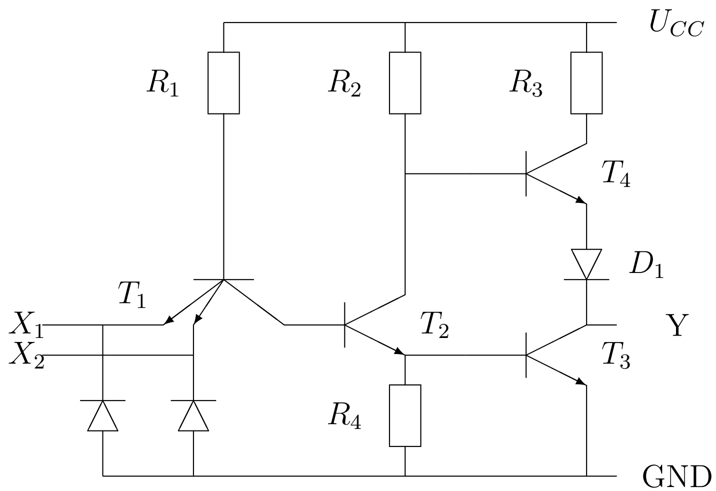


Abbildung 5.4: NAND-Gatter in Standard-TTL

Anmerkung: Diese Gegentaktendstufe heißt *Totem-Pole-Endstufe*. Wesentlicher Vorteil (gegenüber einfachem Ausgangstransistor) ist, daß durch die Transistoren schnelle Umladungen bei beiden Pegelübergängen passieren. Problem: bei Umschaltung zwischen L und H am Ausgang kurzfristiger Kurzschluß (Behebung durch Stützkondensatoren)

TTL-Schaltungen mit spezieller Ausgangsstufe

Anmerkung: Zwei digitale Ausgänge dürfen im allgemeinen nicht miteinander verbunden werden. In Bussystemen sind Schaltungen nötig, die parallel geschaltet werden können.

Three-State-Ausgang

Definition 5.25 (Three-State-Ausgang)
Digitale Bausteine mit *Three-State-Ausgang* können ausgangsseitig hochohmig geschaltet werden gegenüber beiden Logikpegeln. Derartige Ausgänge haben einen dritten Zustand, in dem sie passiv sind. Diese Passivierung erfolgt in der Regel über einen Steuereingang.

Erklärung 5.26 (Funktion eines Three-State-Ausgangs)
Liegt der Steuereingang EN auf H-Pegel, so arbeitet die Schaltung — wie oben beschrieben — als NAND-Gatter. Wenn EN auf L-Pegel liegt, so ist via

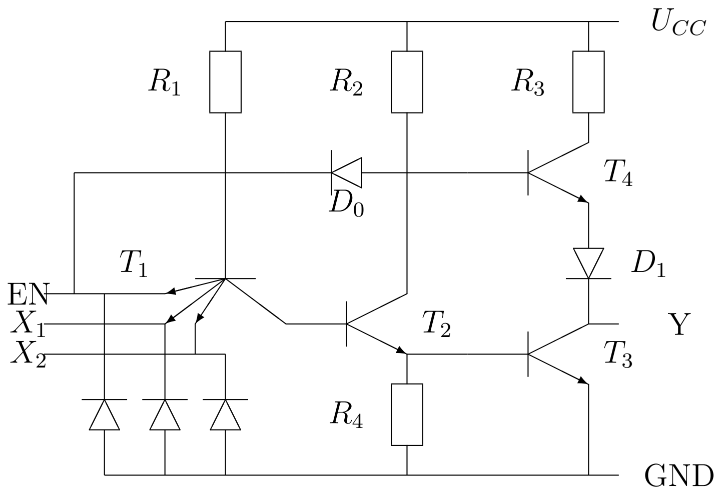


Abbildung 5.5: NAND-Gatter mit Three-State-Ausgang

T1 der Transistor T2 und damit T4 gesperrt, über D0 und R2 auch T3. Der Ausgang Y liegt hochohmig auf undefiniertem Pegel.

Gatter mit offenem Kollektorausgang

Definition 5.27 (offener Kollektorausgang)
Ein *Gatter mit offenem Kollektorausgang* hat keine Gegentaktendstufe, sondern lediglich einen einfachen Transistor. Der Ausgang ist mit dem Emitter oder Kollektor verbunden, nicht aber mit L- oder H-Pegel. Mehrere derartige Ausgänge können parallel geschaltet werden und werden mit einem gemeinsamen Widerstand mit L- oder H-Pegel verbunden (wired-AND bzw. wired-OR).

Schaltungen in Schottky-TTL

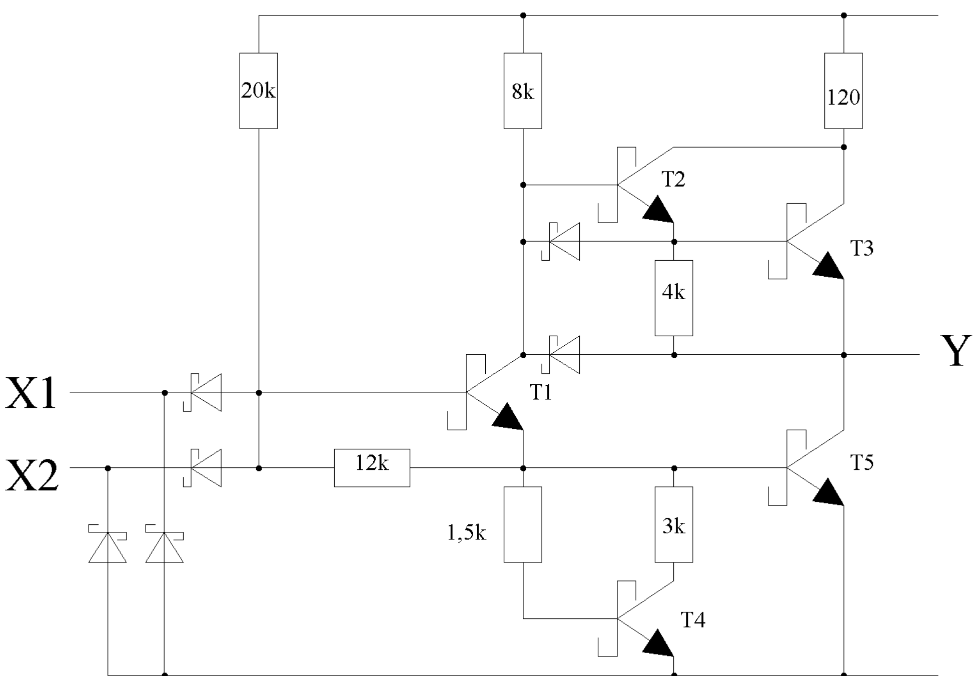
Definition 5.28 (Schottky-Diode, Schottky-Transistor)
Eine Schottky-Diode ist eine Diode mit einem Metall-Halbleiter-Übergang. Sie haben eine geringe Durchlaßspannung und eine geringe Sperrerholzeit. Ein Schottky-Transistor entsteht, indem zwischen Basis und Kollektor eines Transistors eine Schottkydiode geschaltet wird.

Erklärung 5.29 (Wirkung der Schottkydiode im Schottkytransistor)
Vorteil: Durch die Schottkydiode wird verhindert, daß die Basisspannung wesentlich höher wird als die Kollektorspannung (die Diode klammert den Kollektor an die Basis). Damit kann der Transistor nicht in die Sättigung

geraten (d.h. $U_{BC} \geq 0$) und seine Schaltzeit sinkt.
Nachteil: Weil Schottky-Transistoren nicht übersteuert werden, ist ihre Kollektor-Emitterspannung im durchgeschalteten Zustand größer als normal $\Longrightarrow$ geringerer Störabstand

Erklärung 5.30 (Wirkung der Low-Power-Schottky-Schaltung)
Die Funktion ist analog zur Standard-TTL-Schaltung. Besonderheiten sind:

- X1 und X2 sind mittels Dioden auf die Basis von T1 geschaltet (die Wirkung ist aber analog zum Multiemittertransistor).
- T4 wirkt als Emitterwiderstand von T1.
- T2 und T3 sind kaskadiert. Damit sinkt der Ausgangswiderstand, d.h. die Umladezeiten für Lastkapazitäten fallen.
- Die Widerstände sind größer als bei Standard-TTL. Damit ist die Leistungsaufnahme circa 80% geringer.



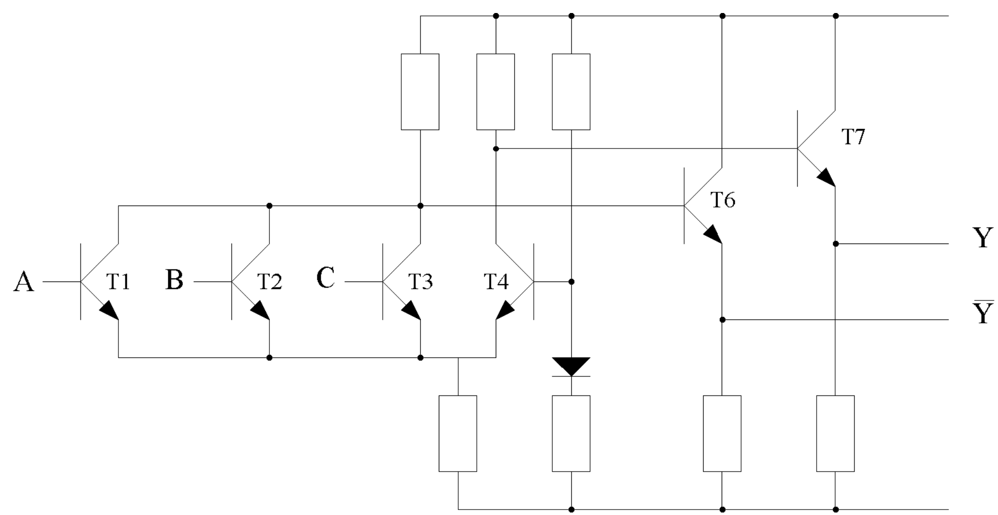
ECL-Schaltungen

Anmerkung: ECL (Emitter Coupled Logic) hat folgende Eigenschaften:

- hohe Schaltgeschwindigkeit
- geringere Integrationsdichte als MOS oder TTL
- geringer Störspannungsabstand
- große Verlustleistung

Typischer Einsatz der ECL-Technik ist z.B. in Zentraleinheiten von Großrechnern.

Beispiel 5.31 (Funktion eines ECL-NOR-Gatters)
Analog zur Schottky-Technik wird bei ECL vermieden, daß Transistoren in die Sättigung geraten.
An der Basis von T4 liegt die konstante Hilfsspannung U_H an.
Wenn die Spannung an den Basen von T1, T2 und T3 kleiner als U_H ist, dann sperren sie und T4 leitet. Damit liegt die Basis von T7 auf L-Pegel (über den leitenden T4) und die Basis von T6 auf H-Pegel (über $(R_B)_6$. Y ist auf L-Pegel und $\overline{Y}$ auf H-Pegel.
Wenn die Spannung an einer Basis von T1, T2 oder T4 etwas größer wird als U_H, dann leitet dieser Transistor und T3 sperrt. Die Ausgänge wechseln ihre Pegel symmetrisch.
Die Schaltung wirkt als Differenzverstärker. Für $U_H = -1,175V$ sind die L- und H-Pegel -1,6V und -0,75V.



Anmerkung: Auffällige Eigenschaft der Schaltung ist, daß eine Erhöhung der Basisspannung eine Erhöhung des Basisstroms bewirkt, damit auch eine

Erhöhung des Emitterstroms (aufgrund der Stromverstärkung des Transistors) und damit der Emitterspannung. Somit kann die Spannung U_{BE} und auch U_{CB} nicht schnell wachsen.

statische MOS-Schaltungen

Anmerkung: Kernstück der MOS-Technik ist der MOS-Feldeffekttransistor (Metal Oxide Semiconductor).

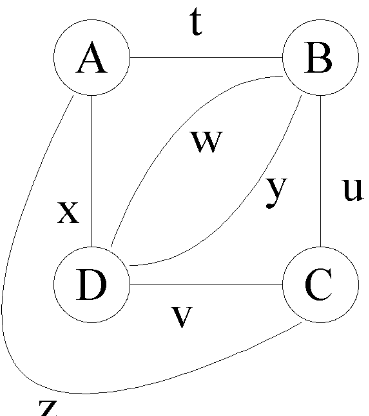
Transmissionsnetze

Anmerkung: Wichtig für die Erstellung von MOS-Schaltungen sind sogenannte Transmissionsnetze. Hier bilden Feldeffekttransistoren (als ideale spannungsgesteuerte Schalter) Netze, die durch Eingangsgrößen gesteuert werden.

Definition 5.32 (Ungerichteter Graph mit Mehrfachkanten)
Ein *ungerichteter Graph mit Mehrfachkanten* ist ein Tupel $G = (V, E, I)$ mit einer *Knotenmenge* $V = \{v_1, \ldots, v_n\}$, einer *Kantenmenge* $E = \{e_1, \ldots, e_m\}$ und einer Abbildung von E in zweielementige Teilmengen von V, d.h. $I : E \to \{x \in \wp(V), |x| = 2\}$ (die *Inzidenzfunktion*).

Beispiel 5.33 (Beispiel eines Graphen mit Mehrfachkanten)

$$\begin{aligned} V &= \{A, B, C, D\} \\ E &= \{t, u, v, w, x, y, z\} \\ I &= \{(t, \{A, B\}), (u, \{B, C\}), (v, \{C, D\}), (w, \{D, B\}), \\ &\quad (x, \{D, A\}), (y, \{B, D\}), (z, \{A, C\})\} \end{aligned}$$



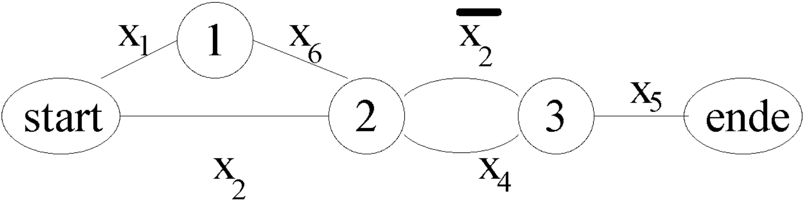
Definition 5.34 (Transmissionsnetze)
Ein *Transmissionsnetz* ist eine Anordnungen von Schaltern zwischen zwei definierten *Terminalknoten.* Jeder Schalter ist markiert mit einer Variablen oder ihrer Negation; dabei dürfen Markierungen auch mehrfach vorkommen. Der Schalter ist genau dann geschlossen, wenn die entsprechende Variable mit 1 belegt ist (bzw. mit 0 bei einer negierten Variablen).

Definition 5.35 (Transmissionsnetz, mathematisch)
Ein Transmissionsnetz $T = (G, start, ende, k)$ enthält folgende Elemente:

- ein ungerichteter Graph $G = (V, E, I)$ mit Mehrfachkanten;
- zwei ausgezeichnete Terminalknoten $start, ende \in V$;
- eine Kantenbeschriftungsfunktion B, die jeder Kante aus E einen aussagenlogischen Term zuordnet.

Definition 5.36 (Durchlaßfunktion eines Transmissionsnetzes)
Die *Durchlaßfunktion eines Transmissionsnetzes* ist bestimmt durch die Belegungen, bei der die Terminalknoten miteinander verbunden sind. Es werden die Terme an Kanten eines Pfades konjunktiv verknüpft und diese Ergebnisse für alle möglichen Pfade disjunktiv verknüpft.

Beispiel 5.37 (Beispiel eines Transmissionsnetzes)
$T = ((x_1 \wedge x_6) \vee x_2) \wedge (\, \overline{x_2} \vee x_4) \wedge x_5$



Anmerkung: Transmissionsnetze ist ein Begriff der sogenannten *Schalterebene* (*Switching-Ebene*) beim Schaltkreisentwurf. Die Hierarchie der Entwurfsebenen sieht wie folgt aus:

1. Ebene der Funktionsbausteine (Rechenwerk, Schieberegister,. . .)
2. Logikebene (mit elementaren Gattern)
3. Schalterebene (mit pegelgesteuerten Schaltern)
4. Schaltkreisebene (mit Transistoren als Schalter)
5. topologische Ebene (prinzipielle Auslegung auf Trägermaterial)
6. geometrische Ebene (exakte physikalische Auslegung auf Trägermaterial)

Seriell-parallel-Transmissionsnetze

Definition 5.38 (Seriell-parallel-Transmissionsnetz)
Ein *seriell-parallel-Transmissionsnetz* kann in zwei seriell-parallel-Transmissionsnetze zerlegt werden, die bezüglich der Terminalknoten in Reihe oder parallel geschaltet sind. Schalter sind atomare seriell-parallel-Transmissionsnetze.

Anmerkung: Obiges Beispiel ist ein seriell-parallel-Transmissionsnetz.

Algorithmus 5.39 (Analyse von sp-Transmissionsnetzen)
Für ein Transmissionsnetz T wird die Durchlaßfunktion berechnet, indem man es in ein gleichwertiges Netz T' umformt mit nur noch einer Kante zwischen den Terminalknoten (durch sukzessive Verschmelzung von Kanten und deren Beschriftungen). Die Beschriftung der verbleibenden Kante ist die Durchlaßfunktion:

1. Wenn für $e_i, e_j \in E$ gilt, daß $I(e_i) = I(e_j)$, dann streicht man e_j aus E. Außerdem faßt man die Einträge (e_i, ρ) und (e_j, κ) in B zum Eintrag $(e_i, \rho \vee \kappa)$ zusammen.
2. Wenn für $e_i, e_j \in E$ gilt, daß $|I(e_i) \cap I(e_j)| = 1$ und $|I(e_i) \cup I(e_j)| = 3$ und $I(e_i) \cap I(e_j) \not\subseteq \{start, ende\}$ und $I(e_i) \cap I(e_j) \not\subseteq I(e_k)$: dann streicht man e_j aus E und setzt $I(e_i) := I(e_i) \uplus I(e_j)$. Außerdem faßt man die Einträge (e_i, ρ) und (e_j, κ) in B zum Eintrag $(e_i, \rho \wedge \kappa)$ zusammen.

3. Sofern sich irgendwelche Änderungen an I ergeben haben, wiederhole Schritte 1 bis 3.
4. Ist mehr als eine Markierung übriggeblieben, dann war T kein SP-Transmissionsnetz. Ansonsten gibt die Markierung die Schaltfunktion.

Anmerkung: Schritt 1 faßt parallele Pfade zusammen; daraus ergibt sich eine ODER-Verknüpfung der Pfadmarkierungen. Schritt 2 faßt hintereinanderliegende Pfade zusammen; daraus ergibt sich eine ODER-Verknüpfung der Pfadmarkierungen. Dies ist aber nur erlaubt, wenn der Verbindungsknoten kein Terminalknoten ist, danach kein Zyklus entsteht und wenn der Verbindungsknoten nicht noch auf weiteren Kanten liegt.

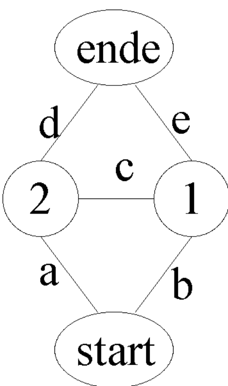
Algorithmus 5.40 (Synthese von sp-Transmissionsnetzen)
Für einen aussagenlogischen Ausdruck A wird ein SP-Transmissionsnetz T gesucht, das die Durchlaßfunktion A hat und an den Kanten nur mit Aussagenvariablen oder deren Negation beschriftet ist.

- Wenn A eine Aussagenvariable v bzw. deren Negation $\overline{v}$ ist, ist $T = ((\{start, ende\}, \{e_1\}, \{(e_1, \{start, ende\})\}), start, ende, \{(e_1,'' v'')\})$ bzw. $T = ((\{start, ende\}, \{e_1\}, \{(e_1, \{start, ende\})\}), start, ende, \{(e_1,'' \overline{v}'')\})$.
- Wenn A negiert ist, aber nicht nur aus einer Aussagevariablen besteht, wendet man die De-Morgansche Regel an und bildet das SP-Transmissionsnetz für den resultierenden Ausdruck.
- Wenn A von der Form $B \vee C$ ist, bildet man die SP-Transmissionsnetze T_1 für B und T_2 für C. Alle Knoten außer *start* und *ende* und alle Kanten werden in T_2 so konsistent umbenannt, daß sie disjunkt zu denen aus T_1 sind. Dann ist $T = ((V_1 \cup V_2, E_1 \cup E_2, I_1 \cup I_2), start, ende, B_1 \cup B_2)$.
- Wenn A von der Form $B \wedge C$ ist, bildet man die SP-Transmissionsnetze T_1 für B und T_2 für C. Alle Knoten außer *start* und *ende* und alle Kanten werden in T_2 so konsistent umbenannt, daß sie disjunkt zu denen aus T_1 sind. Es wird ein neuer Knotennamen vergeben und *ende* in T_1 bzw. *start* in T_2 konsistent auf diesen neuen Namen umbenannt. Dann ist $T = ((V_1 \cup V_2, E_1 \cup E_2, I_1 \cup I_2), start, ende, B_1 \cup B_2)$.

Erklärung 5.41 (Funktionsweise der SP-Transmissionsnetz-Synthese)
Die ersten beiden Fälle sind einfach. Für eine Disjunktion werden zwei Transmissionsnetze parallel geschaltet, für eine Konjunktion in Reihe. Man benennt die Knoten nur um, um Namensgleichheiten in Teilnetzen zu vermeiden.

Anmerkung: Man erhält das Transmissionsnetz der Negation eines Ausdrucks, indem man alle Serienschaltungen in Parallelschaltungen umwandelt und umgekehrt und die Beschriftungen der Kanten negiert (Anwendung von Shannontheorem).

Anmerkung: Es gibt Transmissionsnetze, die keine SP-Netze sind. Ihre Durchlaßfunktion ist die Disjunktion aller Pfade zwischen den Terminalknoten. Analyse der Durchlaßfunktion ist also algorithmisch einfach, die Synthese (z.B. wenn man die Zahl der Knoten des Netzes minimieren will) kompliziert (NP-vollständig?).

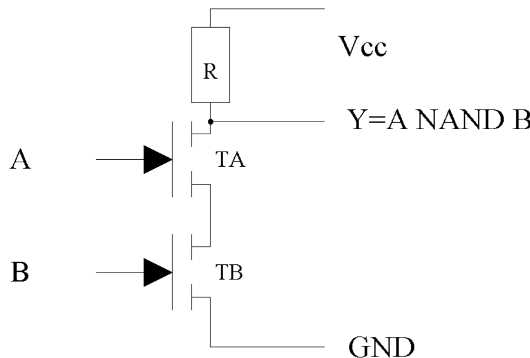


$$f(T) = a \wedge (d \vee c \wedge e) \vee b \wedge (e \vee c \wedge d)$$

NMOS, PMOS

Anmerkung: Die NMOS-Technik verwendet n-Kanäle zur Verbindung von Source und Drain, die PMOS-Technik p-Kanäle.
Inzwischen ist die PMOS-Technik veraltet, weil PMOS-Transistoren einen höheren Widerstand haben.

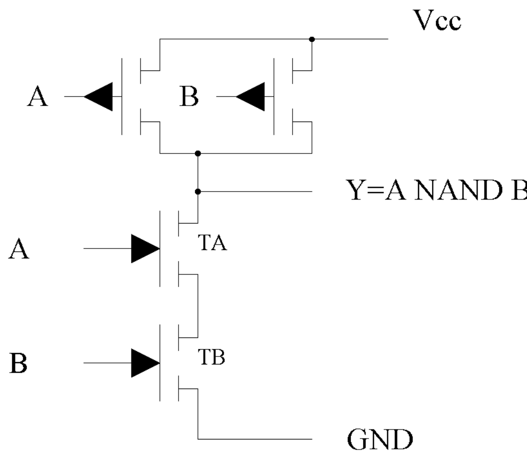
Beispiel 5.42 (NAND-Gatter in NMOS)
Liegt am Gate von TA und TB H-Pegel, dann leiten sowohl TA als auch TB und Y ist mit L-Pegel verbunden. Bei L-Pegel an mindestens einem Gate sperrt der entsprechende Transistor und Y ist über R mit H-Pegel verbunden. Statt eines Widerstands kann ein FET verwendet werden.



CMOS

Anmerkung: Die CMOS-Technik verwendet sowohl n- als auch p-FETs, die im Gegentaktbetrieb arbeiten.
CMOS ist eine Standardtechnologie, die schon stark TTL abgelöst hat.

Beispiel 5.43 (NAND-Gatter in CMOS)
Wenn an beiden Eingängen H-Pegel anliegt, dann leiten beide n-FETs und am Ausgang liegt L-Pegel, weil die p-FETs sperren.
Wenn an einem Eingang L-Pegel anliegt, dann leitet ein p-FET und mindestens ein n-FET sperrt, d.h. der Ausgang liegt auf H-Pegel.
Die p-FETs realisieren in negativer Logik dieselbe Schaltfunktion wie die n-FETs in positiver Logik.



Anmerkung: Charakteristische technische Merkmale der CMOS-Technik sind:

- Stromverbrauch nur beim Umschalten oder durch Leckströme $\Longrightarrow$ geringe Verlustleistung

- Signalhub schöpft Pegelbereich fast völlig aus $\Longrightarrow$ hohen Störabstand;

- aufgrund geringer Schwellenspannung (circa 1V) breiter Bereich für Versorgungsspannung (3V. . . 20V)

Anmerkung: Generelle Merkmale der MOS-Technik sind:

- geringer Bedarf an Chipfläche (10% von TTL), weil selbstisolierend

- einfache Herstellung

- extrem kleine statische Verlustleistung (1% von TTL, ab 3-5MHz größer)

- hohe Ein- und Ausgangswiderstände; damit: lange Schaltzeiten und Empfindlichkeit gegen statische Ladungen

- nur ein Grundelement

- niedrige Schaltgeschwindigkeit

Vergleich der Schaltungsfamilien

Parameter	Schaltkreisfamilie				
	TTL			CMOS	ECL
	Std.	ALS	AS	HCT	
U_{CC} $[V]$	5	5	5	5	-5,2
$(U_{OH})_{min}$ [V]	2,4	2,4	2,4	3,84	-1,025
$(U_{OL})_{max}$ [V]	0,5	0,5	0,5	0,33	-1,620
$(U_{IH})_{min}$ [V]	2,0	2,0	2,0	2,0	-1,165
$(U_{IL})_{max}$ [V]	0,8	0,8	0,8	0,8	-1,475
$(U_S)_H$ [V]	0,4	0,4	0,4	1,84	0,14
$(U_S)_L$ [V]	0,4	0,3	0,3	0,47	0,145
(I_{IH}) $[\mu A]$	40	20	20	1	350
(I_{IL}) [mA]	-1,6	-0,1	-0,1	-0,001	0,0005
(I_{OH}) $[\mu A]$	-400	-400	-2000	-4000	k.A.
(I_{OL}) [mA]	16	8	20	4	k.A.
Fan-Out	10	20	40	4000	10
P_V [mW]/Gatter	10	1	8,5	0,5/MHz	50
t_{pd} [ns]	10	4	1,5	8	0,75
f_{max} [MHz]	35	70	200	40	400

5.2 Digitale Halbleiterspeicher

Definition 5.44 (Speicher)
Ein digitaler Speicher ist eine Anordnung mehrerer Speicherzellen (mit jeweils einem Bit Kapazität), die so organisiert sind, daß von außen zu jeder Speicherzelle Zugriff besteht.

- Festwertspeicher; Read-Only-Memory (ROM)
- Schreib-/Lesespeicher; Read-Write-Memory (RWM) bzw. Random-Access-Memory (RAM)
- Speicher mit vorwiegendem Lesezugriff; Read-Mostly-Memory (RMM)

Definition 5.45 (Speicherkenngrößen)
Folgende Parameter charakterisieren Speicher:

- Adressierungsart: nach Ort (Adresse) oder nach Inhalt (assoziativ);

- Zugriffseinheit: bitorientiert, wenn Adressierung einzelner Speicherzellen erfolgt; bei Adressierung von Speicherzellengruppen wortorientiert;
- Kapazität (in Zahl der Speicherworte);
- Flüchtigkeit: *flüchtig*, wenn bei Ausfall der Betriebsspannung Information verloren geht, sonst nicht flüchtig;
- Zugriffszeit: Zeit zwischen Adressierung und Liefern der Daten;
- Zykluszeit: Minimalzeit zwischen zwei Zugriffen;
- Löschbarkeit;
- Zugriffsart: sequentiell oder wahlfrei;
- Zwang zur Auffrischung: dynamisch oder statisch;

5.2.1 Festwertspeicher

Definition 5.46 (Festwertspeicher (ROM))
Ein ROM ist ein Speicher, dessen Inhalt in der Regel einmalig festgelegt wird und der anschließend nicht mehr verändert werden kann.

Anmerkung: Man kann Festwertspeicher beliebig oft auslesen, sie sind nicht flüchtig und meistens billig.
Festwertspeicher werden verwendet als Programmspeicher, als Kodewandler, in Funktionsgeneratoren usw.

Struktur von Festwertspeichern

Anmerkung: Die Grundstruktur eines ROMs ist realisierbar durch zwei unterschiedliche Arten der Speicheradressierung:

- lineare Auswahl der Adresse
- Koinzidenzauswahl der Adresse

Erklärung 5.47 (Funktion eines ROMs mit linearer Adreßauswahl)
Es wird über den Adreßdekoder stets genau eine von N Wortleitungen auf H-Pegel gelegt. Jede der M Bitleitungen, die über eine Diode mit dieser Wortleitung verbunden sind, wird auf H-Pegel gezogen, der Rest über die Widerstände auf L-Pegel.

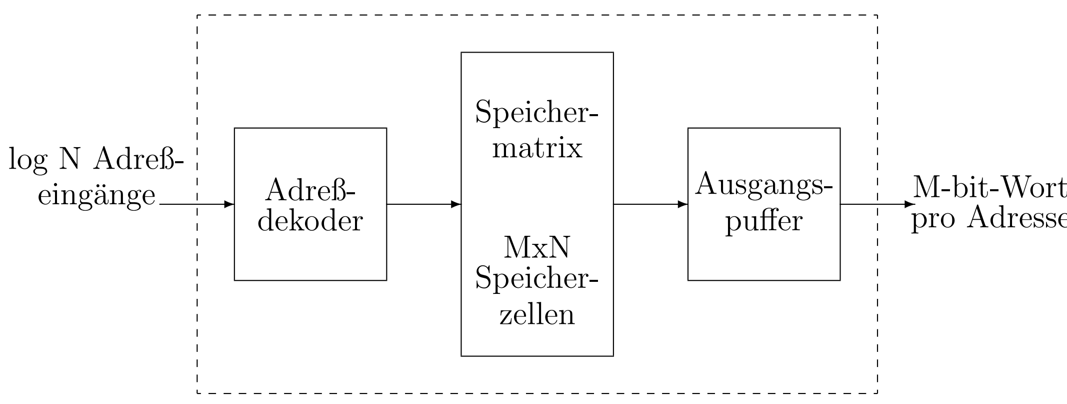
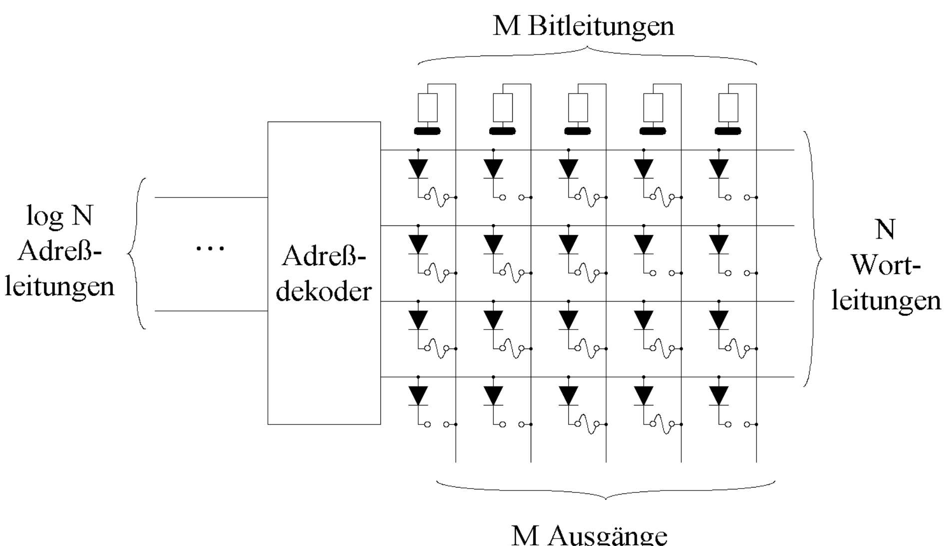
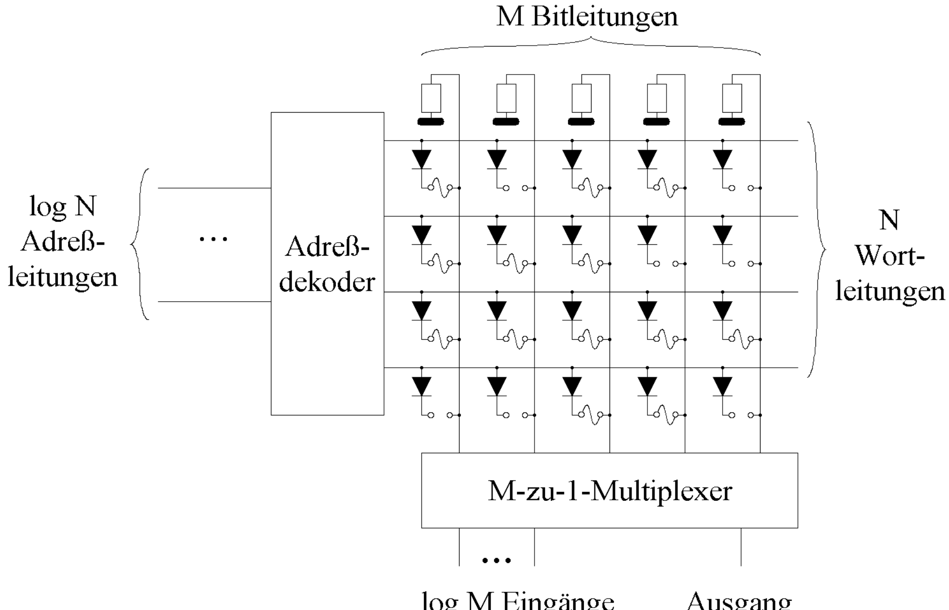


Abbildung 5.6: Grundsätzliche Struktur eines ROMs



Alternative Vorstellung eines ROMs: N UND-Gatter (für jeden Minterm eines) und M ODER-Gatter mit je N Eingängen. Der Ausgang jedes UND-Gatters ist mit einem Eingang jedes ODER-Gatters verbunden. Diese Verbindungen zwischen UND- und ODER-Gattern können wahlweise aufgetrennt werden.

Erklärung 5.48 (Funktion eines ROMs mit Koinzidenzadreßauswahl) Statt M Ausgängen werden $\log_2 M$ Eingänge verwendet, die als Adressen für einen M-zu-1-Multiplexer verwendet werden$\leadsto$ Reduktion der Chipanschlüsse.



Anmerkung: Da ROMs meist über Tri-State-Ausgänge verfügen, kann die Kaskadierung — wie gehabt — über zusätzliche Adreßdekodierung (wie z.B. bei Demultiplexern) erfolgen.

Implementierungen von Festwertspeichern

Anmerkung: Es gibt drei Arten von ROMs:

maskenprogrammierte ROMs: sie erhalten ihren Inhalt bei der Herstellung durch eine spezielle Maske (nur für hohe Stückzahlen rentabel);

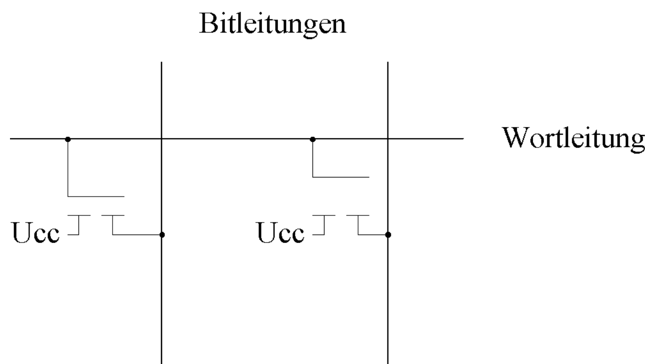
einmalprogrammierbare ROMs (PROMs): Programmierung erfolgt durch Zerstörung bestimmter Chipstrukturen mittels Spannungen;

wiederprogrammierbare ROMs: Löschung und Wiederprogrammierung entweder außerhalb des Geräts (mittels UV-Licht und Spannungen, EPROMs) oder nur durch Spannungen im Gerät.

Realisierung maskenprogrammierter ROMs

Anmerkung: Bei der technischen Realisierung mittels MOSFETs werden z.B. "angeschlossene Dioden" durch FETs mit dünner Oxidschicht realisiert,

"abgetrennte Dioden" durch FETs mit dicker Oxidschicht. In folgendem Bild kann der rechte Transistor niemals leiten, wenn die Wortleitung aktiviert wird, da der Feldeffekt zu schwach ist, um die Selbstsperrung zu überwinden.



Denkbar ist auch die Programmierung bei Aufbringung der Metallisierung.

Realisierung einmalprogrammierbarer ROMs (PROM)

Anmerkung: Die Programmierung von PROMs erfolgt meistens durch Durchbrennen von Sicherungen auf dem Chip. Diese Sicherungen liegen zwischen Wort- und Bitleitung in Reihe mit den Dioden bzw. Transistoren. Durch Anlegen eines kurzen Spannungsstoßes mit einer großen Programmierspannung wird die Verbindung aufgetrennt.
Problem: Das Durchbrennen kostet Zeit...

Realisierung wiederprogrammierbarer ROMs Die wiederprogrammierbaren ROMs werden insbesondere für Prototypen oder bei Kleinserien verwendet.

Anmerkung: Man unterscheidet wiederprogrammierbaren ROMs danach, wie Löschung und Neuprogrammierung erfolgt:

- Löschung außerhalb der Schaltung und nur global möglich (EPROMs);
- selektive Löschung innerhalb der Schaltung (EEPROMs);

Erklärung 5.49 (Funktionsweise eines EPROMs (erasable PROM))

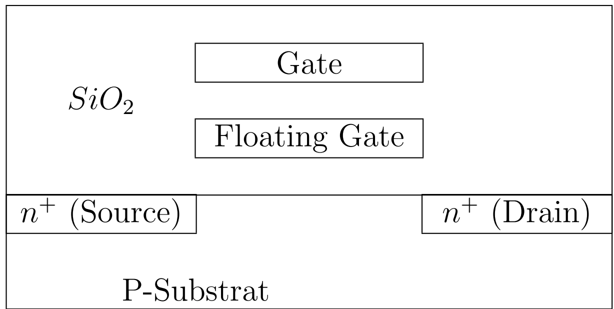


Abbildung 5.7: Struktur einer EPROM-Zelle

Der einzige Unterschied zu PROMs besteht in der Struktur der Verbindungstransistoren zwischen Wort- und Bitleitungen. Beim EPROM liegt zwischen Source und Gate im Oxid ein zusätzliches Gate ("floating gate"). Es trägt keine Ladung und beeinflußt die Funktion des Transistors nicht.
Wird aber einmalig eine Programmierspannung zwischen Source und Gate angelegt, werden negative Ladungen in dieses Gate injiziert. Damit steigt die Schwellspannung des FETs entscheidend an, weil durch die negative Ladung Elektronen aus dem n-Kanal zwischen Source und Drain "weggedrückt" werden.
Durch Bestrahlen des Siliziumdioxids mit UV-Licht wird es leitend und die Ladung der schwebenden Gates geht verloren.

Anmerkung: Anmerkungen zu EPROMs:

- Das schwebende Gate hält seine Ladung über mehrere Jahre (> 10).
- Die Löschung kann in der Regel nur außerhalb des Geräts erfolgen mit Spezialwerkzeugen (braucht > 10min!).
- Es sind nur circa 100 Reprogrammierungen möglich.
- Sonnenlicht und Zimmerlicht löschen ein EPROM in einer Woche bzw. 3 Jahren!

Erklärung 5.50 (Funktionsweise eines EEPROMs)
Die Programmierung auf 1-Potential erfolgt analog zum EPROM durch Anlegen einer positiven Spannung zwischen Gate und Drain. Dadurch wird das "schwebende Gate" mit einer negativen Ladung versehen.
Weil der Abstand zwischen Drain und schwebendem Gate sehr gering ist

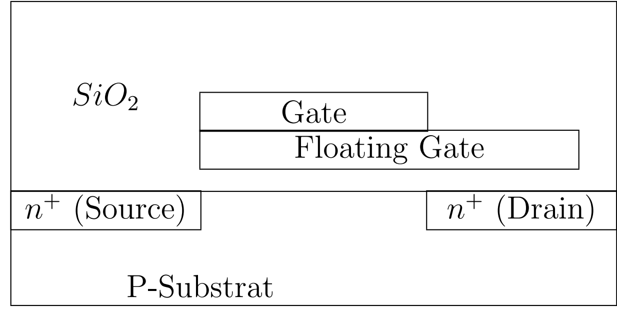


Abbildung 5.8: Struktur einer EEPROM-Zelle

(20nm), kann durch Umkehrung der Programmierspannung (Drain positiv, Gate an Masse) das schwebende Gate entladen werden.

Anmerkung: Kenndaten der EEPROMs:

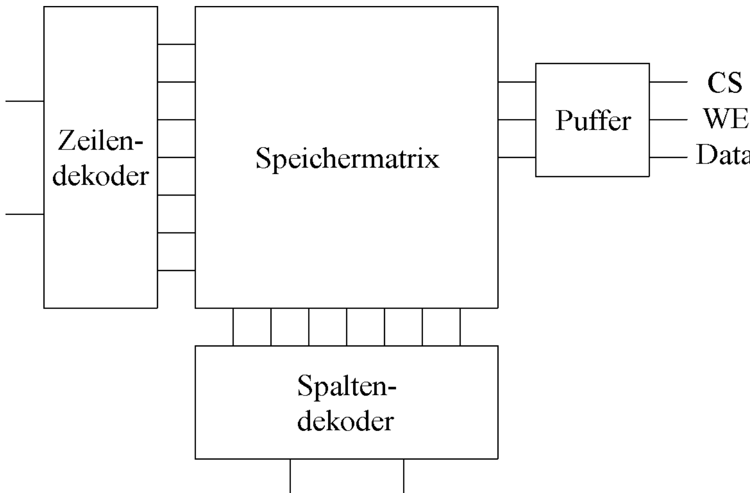
- Anzahl der Schreibzyklen≈10.000;
- Speicherdauer≈10a;
- Schreibzykluszeit: 10ms

Erklärung 5.51 (Nichtflüchtige RAMs (NOVRAMs))
Wegen der hohen Schreibzykluszeit können EEPROMs RAMs nicht ersetzen. $\leadsto$Man kombiniert RAMs mit EEPROMs; bei Abfall der Betriebsspannung wird in 10ms der Inhalt des RAMs ins EEPROM kopiert (non-volatile RAMs, NOVRAMs).

5.2.2 Schreib-/Lesespeicher (RAM)

Definition 5.52 (Schreib-/Lesespeicher (RAM))
Ein *Schreib-/Lesespeicher* ist ein Speicher mit direktem Zugriff auf jede Speicherzelle, bei dem Information beliebig oft eingeschrieben und ausgelesen werden kann. Die Schreib- und Lesezeiten liegen in der gleichen Größenordnung.

Erklärung 5.53 (Aufbau eines RAMs)
Bei einem RAM findet in der Regel eine Koinzidenzauswahl wie bei ROMs statt. Zusätzlich gibt es die Eingänge CS (für die Chipauswahl), WE (für die Richtung des Datentransfers) und die bidirektionale Data-Leitung.



Implementierungen von RAMs

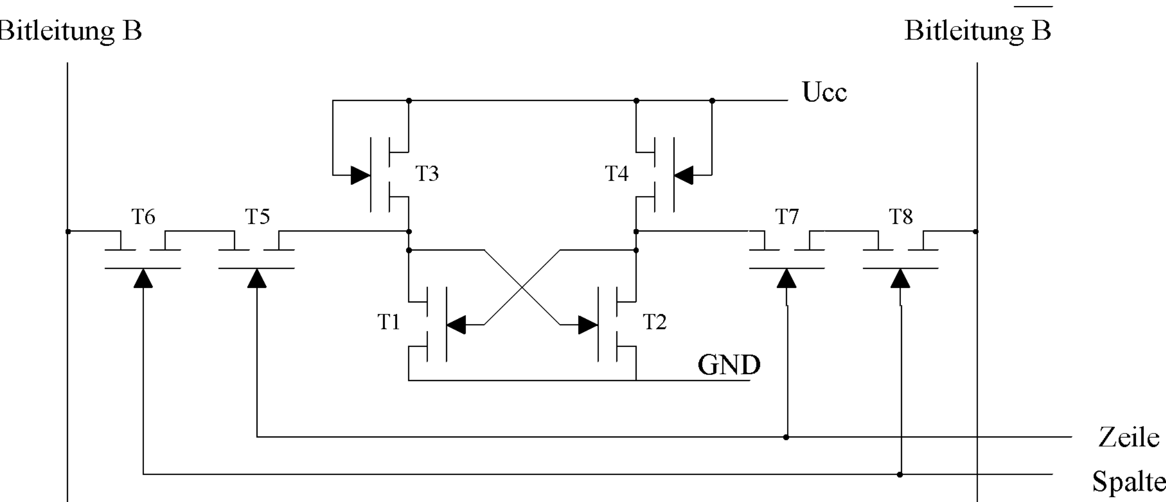
Anmerkung: Man unterscheidet — je nach Implementierung der Speicherzelle —

- statische RAMs und
- dynamische RAMs

Statische RAMs (SRAM)

Definition 5.54 (statisches RAM)
Ein *statisches RAM* ist ein digitaler Speicher mit Speicherzellen, die ohne Auffrischung des Speicherzelleninhalts auskommen.

Erklärung 5.55 (Funktionsweise eine SRAM-Speicherzelle)
T1, T2, T3 und T4 bilden eine bistabile Kippstufe. T3 und T4 wirken als Lastwiderstände. Wenn T1 leitet, dann sperrt T2 und umgekehrt. Sei z.B. T1 leitend, dann wird über dessen leitende Source-Drainstrecke das Gate von T2 auf niedrigen Pegel gelegt wird. Weil die Source-Drainstrecke von T2 hochohmig ist, wird das Gate von T1 auf hohen Pegel gelegt und dessen Durchschaltung stabilisiert.



T5, T6, T7 und T8 sind Transistoren, die bei Adressierung die Ausgänge der bistabilen Kippstufe mit der Bitleitungen B und $\overline{B}$ verbinden.
Lesen: Sei T2 leitend und T1 gesperrt. Wenn an Zeilen- und Spaltenleitung H-Pegel anliegt, dann schalten T5 bis T8 durch und an B liegt H-Pegel und an $\overline{B}$ L-Pegel.
Schreiben: An B liege H-Pegel und an $\overline{B}$ L-Pegel. Nach Auswahl der Zelle wird T2 leitend gemacht und T1 gesperrt. Dieser Zustand bleibt nach Wegfall der Auswahl erhalten.

Erklärung 5.56 (Schreib-/Lesezyklus eines statischen RAMs)
Das Schreiben und Lesen erfolgt in folgenden Schritten:

1. Die Adresse der Speicherzelle wird auf die Adreßleitungen gegeben.

2. Die Schreib-/Leseleitung (WE) wird in den entsprechenden Zustand geschaltet.

3. Der Speicherchip wird mittels Auswahlleitung (CS) aktiviert.

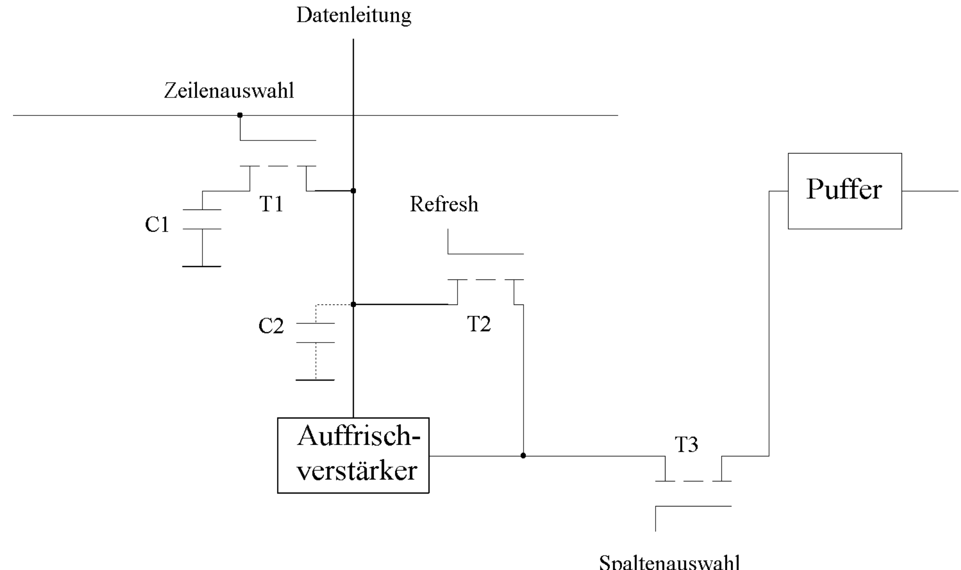
4. Lesen: Nach Ablauf der Lesezugriffszeit erscheint die Information.
Schreiben: Nach Ablauf der Schreibzugriffszeit darf die Information angelegt werden.

Abweichungen in der Abfolge sind denkbar.

Dynamische RAMs

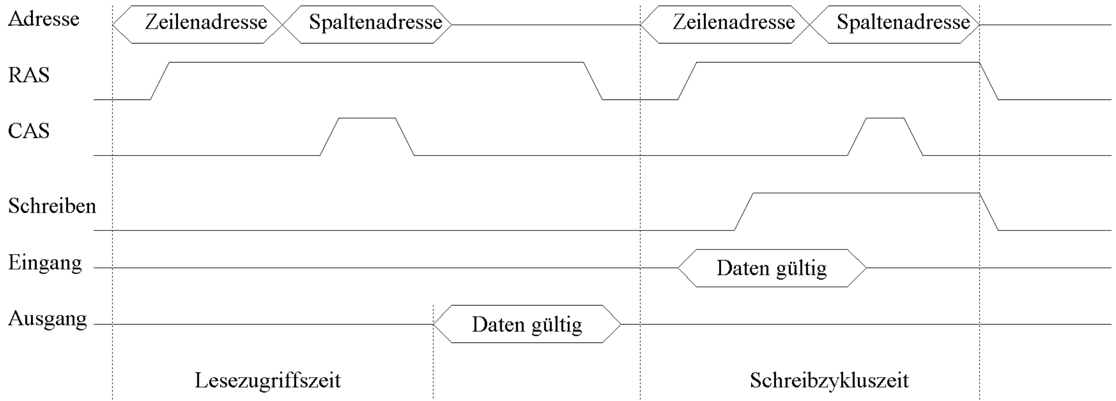
Definition 5.57 (dynamisches RAM)
Ein *dynamisches RAM* ist ein digitaler Speicher mit Speicherzellen aus einer inneren Kapazität eines Feldeffekttransistors. Ihr Inhalt muß aufgrund von Leckströmen regelmäßig aufgefrischt werden.

Erklärung 5.58 (Funktion einer dynamischen Speicherzelle)
Sobald die Zeilenauswahlleitung aktiviert, wird die Datenkapazität (≈ 0,1pF) über den Transistor T1 entladen in C2. C2 ist sehr groß, d.h. es tritt nur ein kurzer Ladestrom auf, der von einem Regenerierverstärker gepuffert wird. Durch T3 erfolgt die Spaltenauswahl; der Pegel der Zelle wird durch einen Leseverstärker nach außen gegeben.
Nach dem Lesen ist der Speicherinhalt zerstört. Eine Wiederherstellung erfolgt so: T3 wird gesperrt, der Senseverstärker hält das Signal eine gewisse Zeit und über T2 wird der Wert wiederum in C1 geschrieben.
Schreiben erfolgt durch Anlegen des Werts an die Datenleitung und Einschalten der Zeilen- und Spaltenauswahl.



Anmerkung: Bei DRAMs werden zur Reduktion der Anschlußzahl die Zeilen- und Spaltenadresse nacheinander an Adreßeingänge angelegt. Information über die Art der Adresse erfolgt durch die Steuerleitungen RAS (row adress strobe) und CAS (column adress strobe). Wenn RAS von 0 auf 1 wechselt, dann ist die an den Adreßleitungen anliegende Adresse eine Zeilenadresse, bei RAS=1 und positivem Flankenwechsel von CAS ist es eine Spaltenadresse. Bei negativen Flankenwechsel von CAS wird Information übernommen bzw. ausgelesen und die komplette Zeile aufgefrischt

Erklärung 5.59 (Schreib-/Lesezyklus eines dynamischen RAMs)
Es gibt Haltezeiten für die Zeilen und Spaltenadresse. Wenn die Schreibleitung auf 0 ist, kommt nach der Lesezugriffszeit der Datenwert an den Ausgang. Wenn die Schreibleitung von 0 auf 1 geht, wird der Datenwert am Eingang übernommen und es muß bis zum nächsten Zugriff die Schreibzykluszeit vergehen.
Nach der Zykluszeit steht das RAM für die nächste Operation zur Verfügung.



Genauer Ablauf:

1. Anlegen von RAS; wenn neue Zeile selektiert wurde, muß der Inhalt der Senseverstärker zurückgeschrieben werden t_{RP}=RAS-precharge-time (2 bis 3 Takte)

2. Selektion der Zeile aufgrund von RAS; Abwarten von t_{RCD}=RAS-to-CAS-delay (2 bis 3 Takte), bis Information in Pufferverstärker stabil

3. Anlegen von CAS; Abwarten von t_{CL}=CAS-latency (2 bis 3 Takte) bis Information am Ausgang verfügbar ist

$\Longrightarrow$Kennzeichnung der Chips mit Taktzeiten für t_{CL}, t_{RCD}, t_{RP}, z.B. PC133-233
$\Longrightarrow$gängige Speicherbandbreitenangaben (1 Zugriff/10ns x 8Byte/Zugriff = 800MB/s!) sind unrealistische best-case-Annahmen!!

Anmerkung: Die Speicherzellen müssen aufgefrischt werden, indem jede Wortleitung einmal innerhalb von $(T_{refr})_{max}$ (z.B. alle 64ms) angesprochen wird (durch einen Lesezyklus).

Erklärung 5.60 (Betrachtungen zum Refresh)
Betrachten wir einen externen Takt von 100MHz (d.h. $t_z = 10ns$). Vorgabe des Herstellers sei, daß jede Zelle nach $\approx 40ms$ aufgefrischt werden muß d.h. alle 4.000.000 Zykluszeiten.

Jetzt spielt die Struktur der Speichermatrix, spezieller die Anzahl der Zeilen eine große Rolle: Da zeilenweise aufgefrischt wird und der Speicherchip bei Auffrischung eine Zeit benötigt bis zum nächsten Auslesen (RAS Cycle Time $\approx$ 8 Takte), geht ein gewisser Prozentsatz an Zeit verloren. Bei einer 8MBit-Speichermatrix kann man z.B. 4096 Zeilen x 2048 Spalten oder auch 16384 Zeilen x 512 Spalten haben. Im ersten Fall gehen alle 40ms $4096 \cdot 80ns = 0,33ms$ verloren, d.h. circa 1%, im zweiten bereits 4%!. Außerdem muß im zweiten Fall der Refresh in 1/4 der Zeit erfolgen (in $40ms/16384 = 2,4\mu s$).

Man bezeichnet übrigens einen Chip der ersten Kategorie als einen mit 12/11-Mapping ($2^{12}x2^{11}$) und einen der zweiten als einen mit 14/9-Mapping. Der Chipsatz muß sich auf die unterschiedlichen Refreshanforderungen einstellen können. Manchmal klappt das nicht: wenn beispielsweise Speicher mit unterschiedlichem Mapping kombiniert werden, kann es zu Speicherfehlern wegen fehlerhaftem Refresh kommen.

Anmerkung: Refreshtechniken sind:

Standard Refresh: RAS und CAS wird aktiviert und Zeilenadresse angelegt; damit wird Zeile aufgefrischt;

Burst Refresh: Abschalten vor Ablauf von $(T_{refr})_{max}$; dann Auffrischen mehrerer (oder aller) Zeilen; bei einem Zugriff wird der Zugreifer nötigenfalls wartend gesetzt (wait states)

Hidden Refresh: Nachschalten eines Refreshzyklus nach jedem Zugriff, eventuell durch internen Zeilenadreßgenerator gesteuert (auto refresh)

Cycle Stealing: gleichmäßiges Aufteilen der Auffrischungszyklen auf $(T_{refr})_{max}$; eventuell Wait States;

Heute gängig: Refreshlogik auf dem RAM-Chip; eventuell Entzerrung der Refreshvorgänge durch Chipsatz (staggered refresh), weil bei der Auffrischung Stromspitzen auftreten

Erklärung 5.61 (Techniken zur Reduktion der Zugriffszeit)
Weil die Zugriffszeit der DRAMs viel zu hoch ist, werden Maßnahmen zur Verkürzung ergriffen:

Interleaving (Verkämmung von Speichern): benachbarte Adressen liegen in mehreren Speichermatrizen/-chips (sog. Bänken); dadurch kann eine Adresse in einer Bank bereits angelegt werden, bevor der Zugriff auf die vorhergehende abgeschlossen ist (beispielsweise bei RAMBUS-Zellen 32 Bänke)

Page-Mode: für Speicherzellen an derselben Wortleitung werden nur die Spaltenadressen angegeben, die Zeilenadresse wird vom Speicher festgehalten; daher ist nur noch t_{CL} pro Informationseinheit zu warten

Double-Data-Rate: Übertragung von Information bei steigender und fallender Taktflanke; Problem: aufgrund der endlichen Ausbreitungsgeschwindigkeit des Taktsignals (15cm/ns) und der Platzbeschränkung (Chipabstand≈10cm) gibt es bereits hohen Versatz der Taktinformation an unterschiedlichen Speicherchips

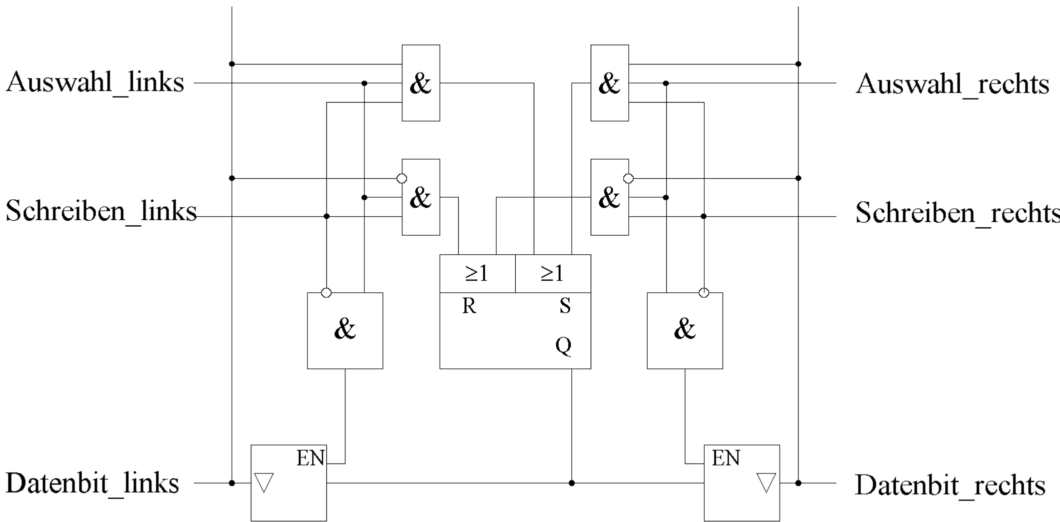
Anmerkung: Vor- und Nachteile von dynamischen RAMs gegenüber statischen:

- höhere Speicherdichte (+)
- geringerer Leistungsverbrauch (+)
- größere Zugriffszeit (-)

Dual-Port-RAMs und VRAMs

Definition 5.62 (Dual-Port-RAM)
Ein *Dual-Port-RAM* ist ein Speicher, der zwei fast unabhängige Adreß- und Dateneingänge für dieselbe Speichermatrix besitzt.

Erklärung 5.63 (Funktion einer Zelle eines Dual-Port-RAMs)
Die Ansteuerlogik ist in zweifacher Form vorhanden. Wenn eine Zelle auf der rechten Seite ausgewählt ist, dann ist die SELECT_R-Leitung auf 1. Wenn SCHREIBEN_R auf 1 liegt, dann wird die Information in DATENBIT_R in die Zelle übernommen. Wenn SCHREIBEN_R 0 ist, dann wird der Lesetreiber aktiviert und der Inhalt der Zelle auf DATENBIT_R gelegt. Analoge Überlegungen gelten für die andere Seite.
Probleme gibt es, wenn beide Ports auf dieselbe Adresse zugreifen wollen und davon mindestens einer schreibend. ⇝Schiedsrichterlogik (Arbitration), die einen Port blockiert; beispielsweise kann das derjenige sein, der später zugreifen wollte

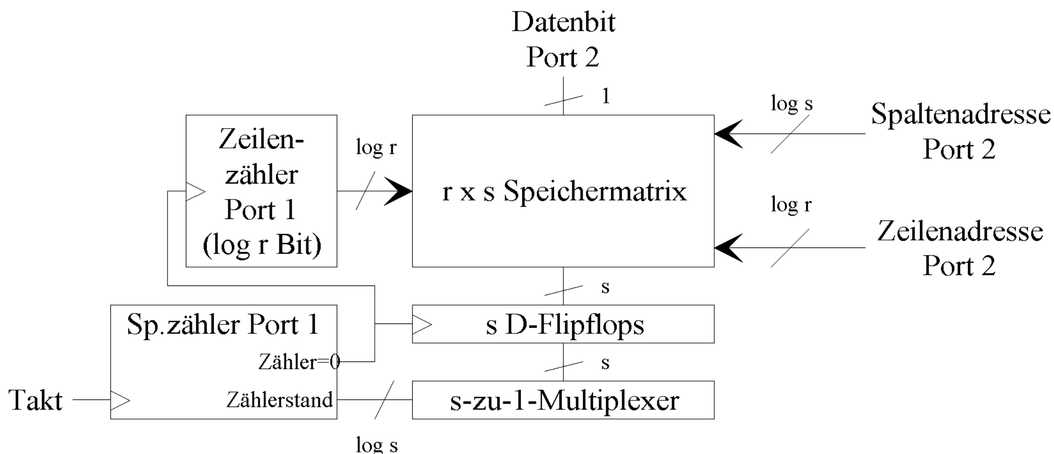


Anmerkung: Anwendungen finden Dual-Port-RAMs in Multiprozessorsystemem (d.h. auch bei intelligenter Peripherie).

Definition 5.64 (VRAM (Video RAM))
Ein *VRAM* ist ein Dual-Port-RAM, bei der ein Port seriell ausliest. Die Speichermatrix dient zur Ablage von Bildern, die punktweise ausgegeben werden.

Anmerkung: Überschlagsrechnung für Bildschirm: bei Auflösung von 1024x1024 Punkten und Bildwiederholfrequenz von 100Hz bleibt pro Punkt eine Auslesezeit von circa $1s/(100 \cdot 1024^2) \approx 10ns$.

Erklärung 5.65 (Technische Realisierung eines VRAMs)
$\rightsquigarrow$ am ersten Port (Videoport) wird eine Speicherzeile komplett in ein Schieberegister übernommen und dieses punktweise an den Ausgang geschrieben (dafür nötig: zwei Zähler); am zweiten Port kann die CPU Veränderungen vornehmen;



5.2.3 Assoziativspeicher

Definition 5.66 (Assoziativspeicher)
Bei einem Assoziativspeicher erfolgt die Adressierung einer Gruppe von Speicherzellen nach einem Teil ihres Inhalts. Der Inhaltsteil, nach dem adressiert wird, heiße *Schlüssel*, der andere *Wert*.

Erklärung 5.67 (Funktionsweise eines Assoziativspeichers)
Bei einem Assoziativspeicher ist die Speicherung der N Schlüssel und Werte getrennt. Jeder Schlüssel liegt in einem eigenen Speicherblock, der mit Zusatzlogik versehen ist.
Beim Lesen werden alle Schlüssel parallel mit dem Eingabeschlüssel verglichen. Die Resultate kommen auf einen 1-aus-N-Kodierer, der eine Adresse für ein nachfolgenden Speicher der Werte erzeugt.
Beim Schreiben wird — wenn der Eingabeschlüssel vorhanden ist — an dieselbe Adresse der neue Wert geschrieben.
Ansonsten wird die Schlüsselzelle beschrieben, auf die am längsten nicht zugegriffen wurde (LRU=least recently used). Dies wird z.B. mit einer Matrixlogik realisiert, die sich für jedes Schlüsselzellenpaar merkt, welche der beiden Zellen länger nicht benutzt wurde. Wenn eine Schlüsselzelle länger als alle anderen nicht benutzt wurde, wird sie beim nächsten Schreibzugriff beschrieben.

Erklärung 5.68 (Logik zur Realisierung des LRU-Mechanismus)
Zur Lösung nehmen wir eine Matrix von RS-Flipflops. Flipflop $\mathrm{FF}_{i,j}$ merkt sich, ob auf Zelle i zeitlich vor Zelle j das letzte Mal zugegriffen wurde ($Q_{i,j} = 1$, wenn Inhalt von Zelle i älter als Inhalt von Zelle j ist).
Wie man sich schnell überlegt, muß gelten: $Q_{i,j} = \overline{Q_{j,i}}$. Man kann sich also

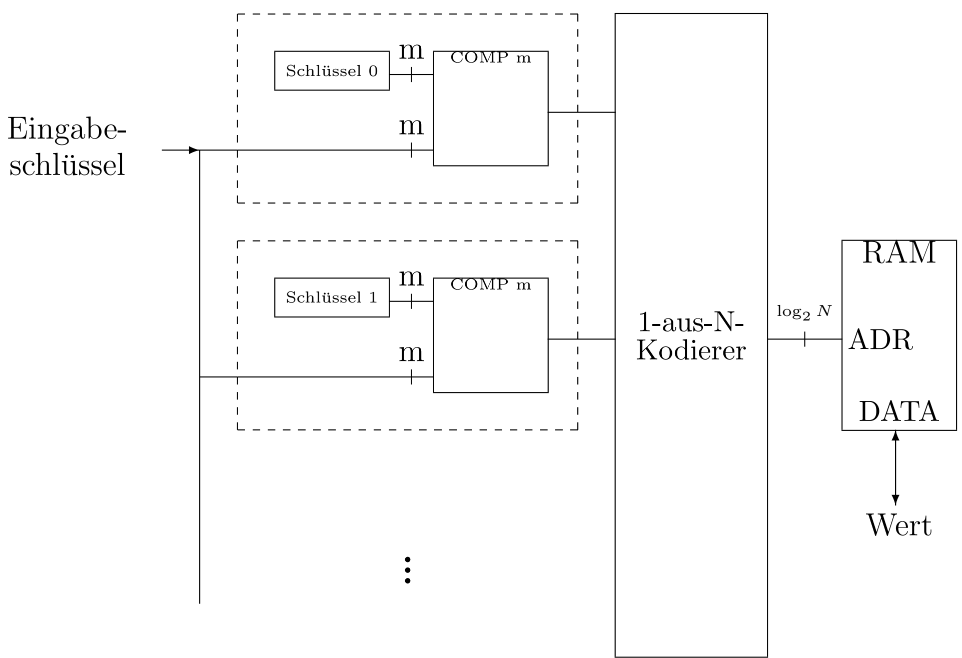


Abbildung 5.9: Lesen in einem Assoziativspeicher

die Hälfte der Flipflops sparen und auch die $\mathrm{FF}_{k,k}$.
Wenn auf Zelle i zugegriffen wird, dann müssen alle $\mathrm{FF}_{i,k}$ zurückgesetzt ($i < k$) und die $\mathrm{FF}_{k,i}$ gesetzt werden ($k < i$) weil der Inhalt von Zelle i der jüngste ist. Bei vier Zellen würde gelten (z_i stehe für den Zugriff auf Zelle i):

z_1 setzt $\mathrm{FF}_{1,2}$, $\mathrm{FF}_{1,3}$ und $\mathrm{FF}_{1,4}$ zurück.

z_2 setzt $\mathrm{FF}_{2,3}$ und $\mathrm{FF}_{2,4}$ zurück und setzt $\mathrm{FF}_{1,2}$.

z_3 setzt $\mathrm{FF}_{3,4}$ zurück und setzt $\mathrm{FF}_{1,3}$ und $\mathrm{FF}_{2,3}$.

z_4 setzt $\mathrm{FF}_{1,4}$, $\mathrm{FF}_{2,4}$ und $\mathrm{FF}_{3,4}$.

Bei der Ausgangslogik gilt: Eine Zelle i ist die älteste, wenn alle Flipflops $\mathrm{FF}_{i,k}$ (auch die wegoptimierten) auf 1 stehen. Für die wegoptimierten muß man prüfen, ob die Flipflops $\mathrm{FF}_{k,i}$ auf 0 stehen. Das heißt

$$A_i = (\bigwedge_{k<i} \overline{Q_{k,i}}) \wedge (\bigwedge_{i<k} Q_{i,k})$$

Anmerkung: Aufgrund der aufwendigen Logik (insbesondere zur Regelung des Schreibens) werden volle Assoziativspeicher mit großer Kapazität

selten realisiert.
Gängig sind Assoziativspeicher mit eingeschränkter Funktionalität z.B. als Cache in Speichersystemen. $\leadsto$: Schlüssel werden nicht beliebig im Speicher abgelegt, sondern nur in bestimmten Bereichen. Ein Teil des Schlüssels (z.B. die ersten 16Bits, bei einem 24Bit-Schlüssel) wählt eine Zelle aus. Dort ist ein kleiner Assoziativspeicher, in dem der Schlüssel abgelegt wird. Dieser Assoziativspeicher hat nur wenige Zellen; damit ist seine LRU-Logik einfach. Derartige Assoziativspeicher heißen *Mehrwegassoziativspeicher* (Beispiel: 4- oder 8-Weg-Assoziativspeicher).

5.2.4 Zusammenfassung Speicher

Anmerkung: Einige technischen Kenndaten von Speicherbausteinen:

Art	Kapazität	Technologie	P_v aktiv/statisch [mW]	Zugriffszeit [ns]
SRAM	1K x 4	ECL	2400/2400	10
SRAM	64K x 1	CMOS	600/100	45
DRAM	4M x 1	CMOS	400/20	100
VRAM	64K x 4	NMOS	1000/80	120
Masken-ROM	256K x 8	CMOS	250/0,5	250
PROM	4K x 4	ECL	750	15
PROM	128K x 8	CMOS	20/MHz	200
EPROM	128K x 8	CMOS	250/0,005	300
E^2PROM	128K x 8	CMOS	300/2	200

Anmerkung: Prinzipielles Problem: Kosten pro Bit sind bei Halbleiterspeichern hoch (0,3Pf bis 10mPf) im Vergleich zu anderen Medien (10μPf bis 100mPf bei Magnetband), dafür ist Zugriffszeit gering (10 bis 100ns) gegenüber anderen Medien (10ms bis 100s)
Rightarrow Zugriffszeitlücke und Preislücke.

Anmerkung: Als Trend ist Reduktion der Speicherarten auf vier Gruppen zu erwarten: dynamische RAMs, statische CMOS-RAMs, EEPROMs und PROMs

5.3 Anwenderspezifische ICs

Definition 5.69 (Anwenderspezifische ICs)
Trend: Schaltungen werden nicht aus einzelnen Standardschaltungen aufgebaut, sondern es wird eine kundenspezifische Schaltung (mit vielen Gatterfunktionen) auf wenigen Chips realisiert.
Man nennt derartige Schaltungen *anwenderspezifische ICs*.

Digitale integrierte Schaltkreise						
Standard-IC			ASIC			
Logikfamilien	Speicher	Mikroprozessor	Semicustom IC			Fullcustom IC
Gatter, Flipflops, Zähler	RAM, ROM		PLD	Gate Arrays	Standardzellen	

Abbildung 5.10: Einteilung der Schaltkreise

Definition 5.70 (Nomenklatur bei ASIC)

PLD (Programmable Logic Device): Eine Schaltung ist von ihrer prinzipiellen logischen Struktur vorgegeben, kann durch Programmierung angepaßt werden (z.B. EPROM).

Gate Arrays: Vordefinierte Zellen mit einfacher Logik können durch chipinterne Verdrahtung zur gewünschten Funktion verschaltet werden. Diese Zellen sind regelmäßig angeordnet (meist matrixartig).

Standardzellenschaltkreise: Aus einer Zellbibliothek können Standardschaltungen (für Gatter, Zähler, Flipflops usw.) nahezu beliebig auf Chip plaziert und verdrahtet werden.

Vollkundenschaltkreis (Fullcustom): Eine Schaltung wird bis auf Transistorniveau spezifiziert und vollständig gefertigt.

Anmerkung: Charakteristika der anwendungsspezifischen ICs:

Art	Gatterzahl	Entwicklungszeit	Entwicklungs-kosten [DM]	Stück-kosten [DM]	Stückzahlen
Fullcustom	500.000	Jahre	200.000	1–200	ab 50.000
Standard-zellen	100.000	Monate	80.000	20–100	ab 5.000
Gate Arrays	70.000	Wochen	50.000	50–100	ab 1.000
PLD	2.000	Tage	2.000	20–50	ab 1

5.3.1 Programmierbare Logikanordnungen (PLD)

Definition 5.71 (PLD)
Ein PLD besteht aus einer zellförmigen Anordnung von UND- und ODER-Gattern aufgebaut, die (programmierbar) die Terme einer beliebigen Schaltfunktion realisieren.

Erklärung 5.72 (Strichsymbolik im Kontext von PLDs)
In schematischen Darstellungen wird folgende Symbolik verwendet:

Die technische Realisierung einer derartigen Zelle erfolgt mit durchtrennbaren Verbindungen zwischen den Eingangsleitungen und Gattereingängen (mit Techniken analog zu PROM, EPROM oder EEPROM).

Erklärung 5.73 (Generelle Struktur von PLDs)
Die UND- und ODER-Matrizen sind je nach Typ des PLDs programmierbar oder fest verdrahtet.

	UND-Feld	ODER-Feld
PLE(PROM)	fest verdrahtet	programmierbar
PAL	programmierbar	fest verdrahtet
PLA	programmierbar	programmierbar

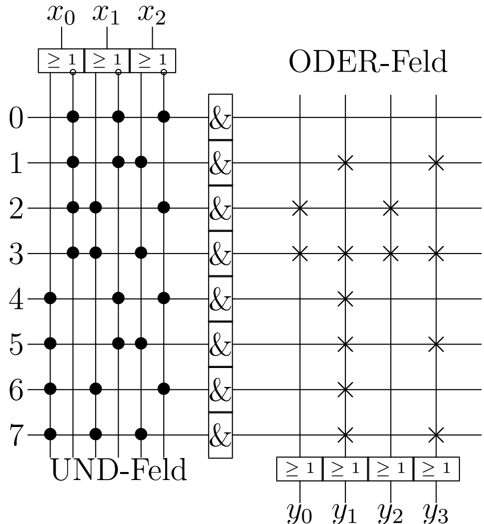
PLE steht für programmable logic element, PLA für programmable logic array und PAL für programmable and logic.

Vergleich zwischen PLE, PAL und PLA

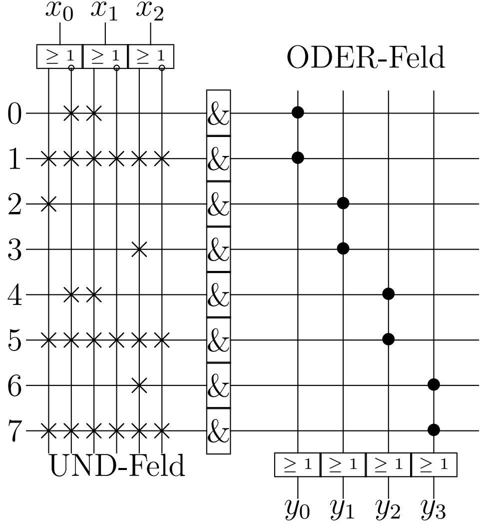
Beispiel 5.74 (Realisierung einer Schaltfunktion mit PLDs)
Eine Schaltfunktion von 3 Variablen sei zu realisieren:

x_0	x_1	x_2	y_0	y_1	y_2	y_3
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	1	0	1	0
0	1	1	1	1	1	1
1	0	0	0	1	0	0
1	0	1	0	1	0	1
1	1	0	0	1	0	0
1	1	1	0	1	0	1

Das heißt: $y_0 = \overline{x_0}\, x_1$, $y_1 = x_0 \vee x_2$, $y_2 = \overline{x_0}\, x1$ und $y_3 = x2$.
Im PLE ist die Programmierung des UND-Feldes festgelegt: es werden alle Minterme der Eingangsvariablen durchgespielt. Das ODER-Feld wird wie die Ergebnisspalte der Wahrheitstabelle programmiert. (Im folgenden stehen Punkte für vorgegebene Verbindungen, Kreuze für programmierte Verbindungen.)

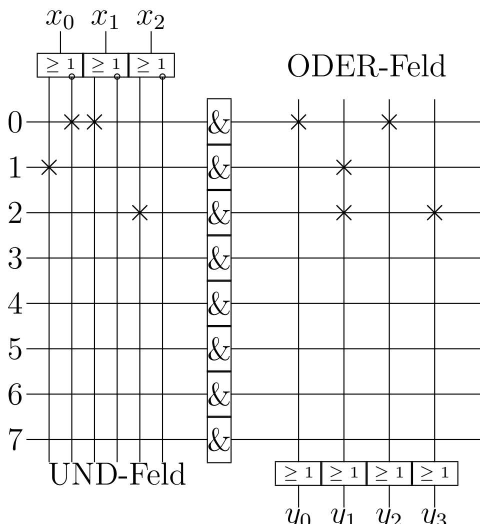


Beim PAL gibt es keine Probleme, da pro Ausgangsvariable maximal zwei Primterme benötigt werden. In manchen Zeilen werden alle Eingänge verknüpft, da dann das UND-Gatter konstant 0 liefert.



Beim PLA können mehrfach vorhandene Zeilen im UND-Feld gemeinsam benutzt werden. Die Programmierung entspricht einer "schlampigen" Wahrheitstabelle mit Don't-Cares bei den Eingangsvariablen.

x_0	x_1	x_2	y_0	y_1	y_2	y_3
0	1	X	1	0	1	0
1	X	X	0	1	0	0
X	X	1	0	1	0	1



Anmerkung: PLEs (PROMs) brauchen für eine vorgegebene Schaltfunktion meist die größten Programmiermatrizen. Allerdings können sie jede Schaltfunktion realisieren. PALs sind problematisch, wenn die Zahl der wesentlichen Primterme für verschiedene Ausgangsvariablen groß ist. PLAs sind sehr flexibel, allerdings sind sie aufwendiger herzustellen.

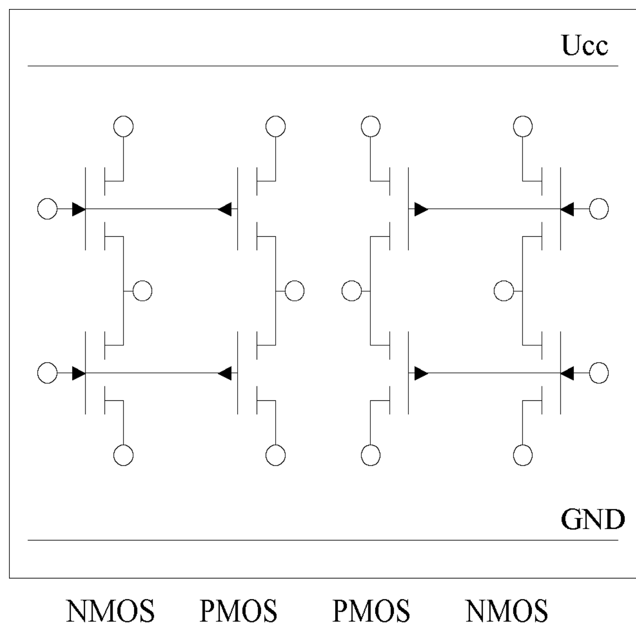
5.3.2 Gate Arrays

Definition 5.75 (Gate Array)
Ein Gate Array besteht aus einer matrixartigen Anordnung von Grundschaltungen (*Zellen*). Der Schaltkreis ist entweder komplett gefertigt und konfigurierbar oder — bis auf die Verbindungen der Zellen — komplett gefertigt. Verbindungen sind sowohl innerhalb von Zellen möglich als auch zwischen den Zellen (*sea of gates*). Diese Verbindungen werden in einem separaten Schritt aufgebracht oder softwaremäßig konfiguriert. Dieser Schritt ist kundenspezifisch und der Entwickler wird dabei durch CAD-Software unterstützt.

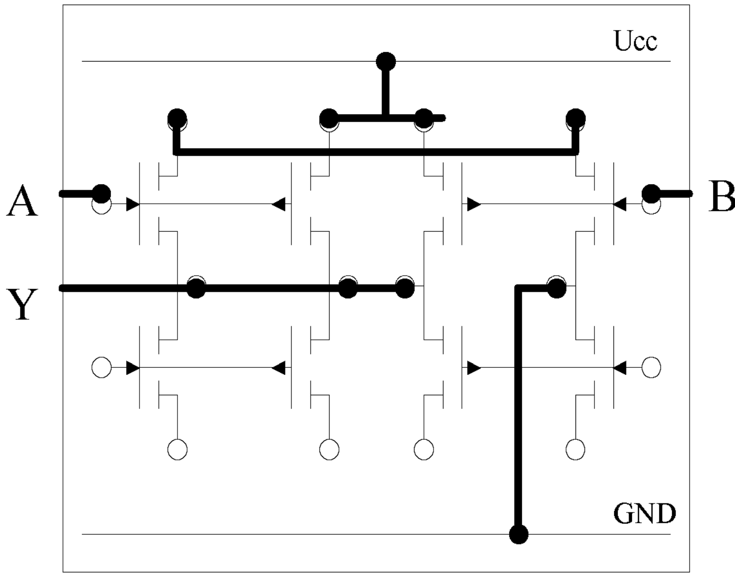
Anmerkung: Ein Gate Array hat meistens zwei Arten von Zellen: *interne Zellen* im Innenbereich, die auf schnelle Signalverarbeitung ausgelegt sind, und *Schnittstellenzellen* im Randbereich, die als Ein- und Ausgangstreiber dienen.

Einfache Gate Arrays

Erklärung 5.76 (Aufbau von einfachen Gate Arrays)



Realisierung von $Y = \overline{A \wedge B}$ mit dieser Zelle durch Aufbringen einer Metallisierungsebene (Verdrahtung):



Anmerkung: Der Entwurf einfacher Zellen mit unterschiedlichen Anschlüssen erfolgt meist durch den Hersteller, nicht durch den Entwickler (in Makrobibliotheken). Diese Bibliotheken enthalten auch komplexere Standardschaltungen (Schieberegister, ALU), die mehrere Zellen benötigen.

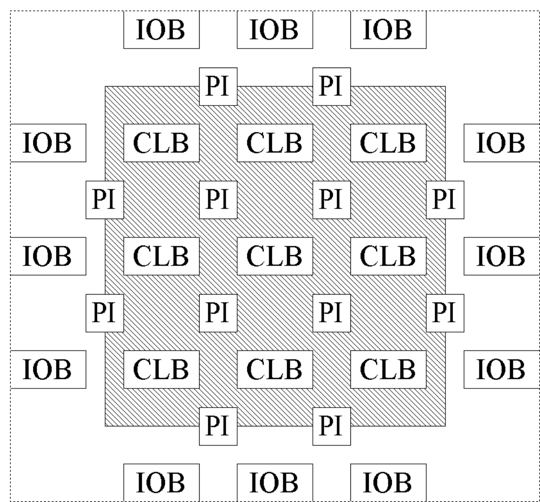
Anmerkung: Oft werden Testschaltungen mitintegriert, die eine automatische Überprüfung der Speicherelemente und der Schaltnetze gestatten. Dadurch gibt es manchmal Einschränkungen in den zulässigen Verdrahtungsmöglichkeiten.
Der Test derartiger Schaltungen erfolgt oft so:

Flipflops: können in Reihe geschaltet als Schieberegister benutzt werden; eine Ausgangsimpulsfolge wird mit einer Eingangsimpulsfolge verglichen;

Schaltnetz: für charakteristische Testfälle werden die Flipflops vorbelegt und Eingangswerte angelegt; diese ergeben überprüfbare Ausgangswerte

Programmierbare Gate Arrays (FPGAs)

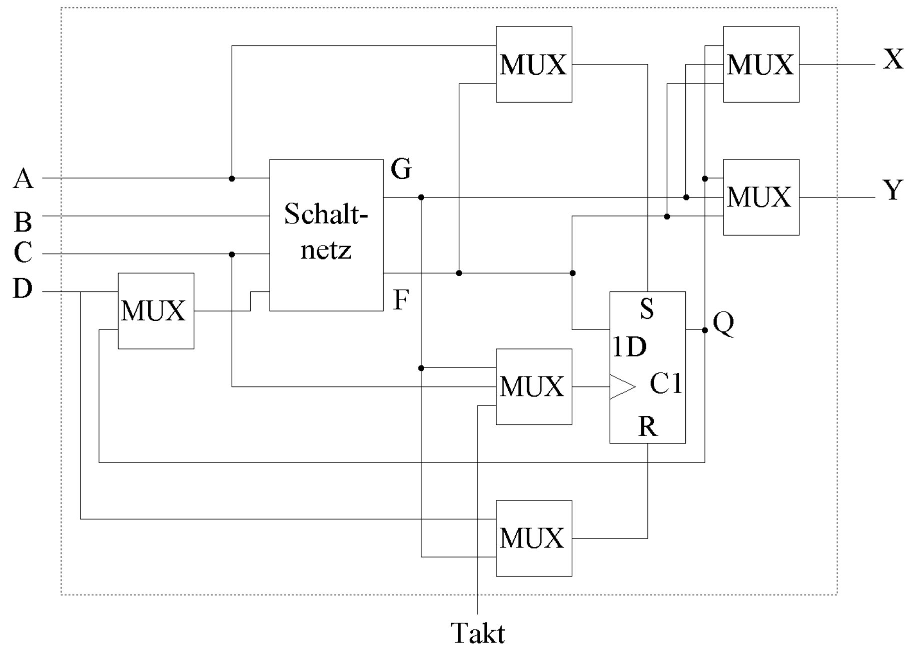
Definition 5.77 (Programmierbares Gate Array)
Bei einem *programmierbaren Gate Array* werden vollständig vorgefertigte Schaltungen benutzt. Konfigurationsdaten sind in einem internen EPROM gespeichert und werden beim Einschalten der Schaltung benutzt, um die internen und Randzellen in ihrer Funktion festzulegen und die Verbindungen sämtlicher Blöcke untereinander zu bestimmen. Man unterscheidet daher die Blocktypen I/O-Block (IOB), zentraler Logikblock (CLB) und Verbindungsblock (PI).



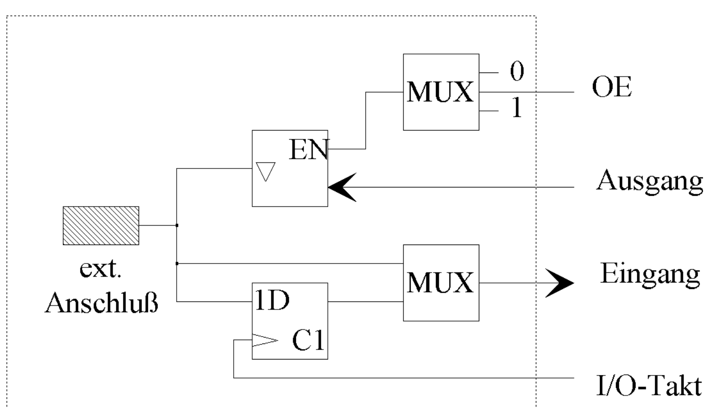
Erklärung 5.78 (Aufbau eines speziellen FPGAs)

Logikblock (configurable logic block): Der CLB hat vier Eingänge A, B, C, D, Takteingang T und Ausgänge X und Y. Das Schaltnetz für F und G wird mit einem 16bit-RAM konfiguriert (als eine Schaltfunktion von 4 Eingangsvariablen und G=F oder zwei Schaltfunktionen für F und G mit je 3 Eingangsvariablen[1]).
Über jeweils mittels 1 oder 2 Bit programmierbaren Multiplexern wird die interne Weiterleitung von Information vorgenommen.
Das Flipflop ist über 2 Bit programmierbar als flanken- oder zustandsgesteuert und in Flanke (positiv oder negativ) bzw. Zustand.

[1]Es gibt für 4 Variablen 16 Wertkombinationen, deren jeweiliger Funktionswert in einem Bit kodiert wird.



Ein/Ausgabeblock (I/O block): Ein IOB enthält einen programmierbaren Eingangspfad und einen Ausgangspfad. Der Ausgang ist über einen Multiplexer programmierbar als steuerbarer Tristate-Ausgang bzw. aktiver ("0") oder hochohmiger ("1") Ausgang.
Der Eingang wird entweder direkt oder taktsynchronisiert vom Anschluß ins Schaltungsinnere weitergeleitet.



Verbindungsblock (programmable interconnect): Ein Verbindungsblock stellt eine frei konfigurierbare Zuordnung von seinen Eingängen zu seinen Ausgängen her. Die Verbindungen werden über mehrere programmierbare Multiplexer hergestellt. Diese Multiplexer verbinden

Ausgänge von CLBs mittelbar oder unmittelbar mit Eingängen anderer CLBs. Daneben können Informationen auf horizontale und vertikale Busleitungen gelegt werden. Abschätzung: 16 Eingänge und Ausgänge in Verbindungsblöcken erfordert Konfigurationsaufwand von $4 \cdot 16$ Bit.

Anmerkung: Gängige Größen von FPGAs:

Typ	Gatteräquiv.	CLB	IOB	Flipflops
AMD 2018	1800	100	74	174
AMD 3064	6400	224	120	688
AMD 4020	20.000	900	240	28.800

Zusammenfassung zu Gate Arrays

- Vorteile des einfachen Gate Arrays:
  - große Packungsdichte;
  - bei gleicher Komplexität geringerer Teileaufwand;
  - hohe Verarbeitungsgeschwindigkeit (bis GHz);
  - bei großen Stückzahlen preiswerter;
- Vorteile eines programmierbaren Gate Arrays:
  - keine kundenspezifische Fertigung;
  - fertige, getestete Hardware;
  - kurze Entwicklungszeit, geringe Entwicklungskosten;
  - auch nachträglich änderbar;

5.3.3 Standardzellenschaltkreise

Definition 5.79 (Standardzellenschaltkreise)
Standardzellenschaltkreise nutzen Zellen aus einer Zellbibliothek für Standardschaltungen (z.B. Gatter, Zähler, Flipflops usw.), die nahezu beliebig auf Chip plaziert und verdrahtet werden können.
Die Zellen sind flächenoptimiert und oft in Breite oder Höhe normiert (für gerasterte Anordnung der Zellen).

Anmerkung: Standardzellen sind insbesondere interessant für Hybridschaltungen, in denen analoge und digitale Komponenten gemeinsam auf einem Chip vorkommen.

Anmerkung: Eigenschaften von Standardzellenschaltkreisen:

- anpaßbare Chipgröße;
- sehr hohe Packungsdichte;
- hohe Verarbeitungsgeschwindigkeit;
- hoher Fertigungsaufwand;

Kapitel 6

Mikroprozessoren

6.1 Grundlagen

6.1.1 Allgemeines

Anmerkung: Problem: Bei zunehmender Integrationsdichte wurden die Chips immer leistungsfähiger, aber auch zu spezialisiert.
Der Mikroprozessor entstand 1971 als einfacher programmierbarer Baustein zur Terminalsteuerung. Das Resultat (der Intel 4004) war für die vorgegebene Aufgabe zu langsam, wurde aber als Universallogikbaustein vermarktet
$\rightsquigarrow$Mikroprozessor als konfigurierbare Hardware
Wesentlicher Vorteil: die Konfiguration erfolgt durch Software!

Anmerkung: Ein Mikroprozessor besteht aus einem oder mehreren Chips und kooperiert mit Speichern und Interfacebausteinen. Die gesamte Einheit heißt Mikrorechner oder Mikrocomputer.
Der heutige Anwendungsbereich liegt zunächst in der Steuerungs- und Regelungstechnik, aber auch immer stärker als Universalrechner.

Anmerkung: Spezielle Mikroprozessoren:

> Mikrocontroller: vollständiger Mikrocomputer (inkl. Speicher und Interface) auf einem Chip; Einsatz in der Steuerungstechnik
>
> Signalprozessoren: spezialisiert auf schnelle Ausführung von Operationen für Signalverarbeitung (Addition, Multiplikation) z.B. für schnelle Fouriertransformation

Arithmetikprozessoren: spezialisiert auf arithmetische Operationen; kooperieren in der Regel mit Mikroprozessoren

6.1.2 Aufbau eines Mikrocomputers

Erklärung 6.1 (Von-Neumann-Maschine)
Ein *von-Neumann-Rechner* kann in folgende Teilsysteme zerlegt werden:

- Steuerwerk: steuert den Ablauf des Systems
- Rechenwerk (ALU): führt arithmetische Verknüpfungen durch
- Speicherwerk: speichert Programme und Daten
- Ein-/Ausgabewerk: dient zur Kommunikation des Systems nach außen
- Verbindungen zwischen den Teilsystemen

Folgende Regeln gelten für von-Neumann-Systeme:

- Die Anpassung an das Problem erfolgt durch eine im Speicherwerk abgelegte Bearbeitungsvorschrift (das Programm).
- Der Zugriff auf das Speicherwerk erfolgt über Adressen. Im Speicherwerk werden Daten und Programmbefehle nicht prinzipiell unterschieden.
- Programmbefehle werden in der Regel aus aufeinanderfolgenden Adressen geholt.

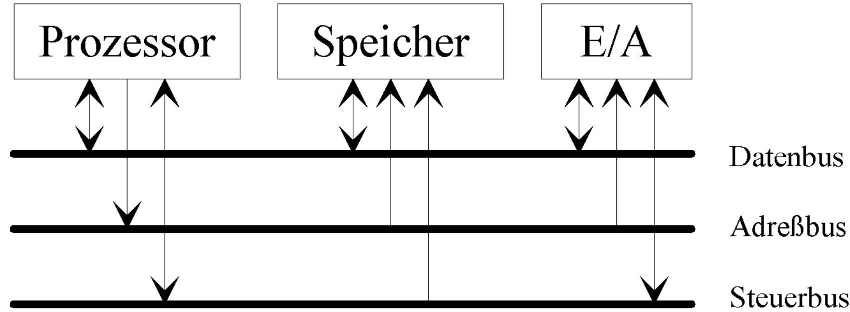
Definition 6.2 (Zentraleinheit, CPU)
Eine *CPU* ist die Kombination aus Steuerwerk und Rechenwerk. Ein Mikroprozessor ist eine CPU.

Definition 6.3 (Bussystem)
Die Verbindung zwischen den diversen Teilsystemen wird in einem Mikrocomputer durch ein *Bussystem* hergestellt.
Ein Bussystem ist ein Bündel funktional zusammengehörender Verbindungsleitungen, an die mehrere Bausteine parallel angeschlossen sind. Es besteht aus:

Adreßbus: Hier überträgt die CPU unidirektional die Adressen von Speicherplätzen oder E/A—Kanälen, auf die zugegriffen werden soll.

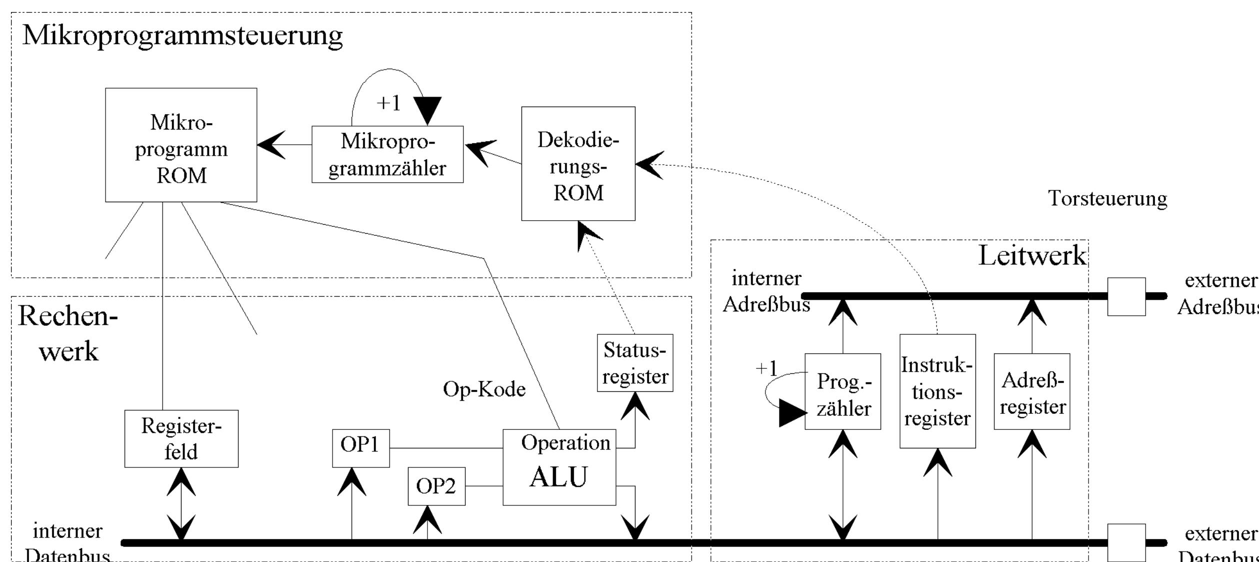
Datenbus: Er dient zur (bidirektionalen) Übertragung von Daten zwischen CPU und Speicher oder Ein-/Ausgabe (unter Ausnutzung der Tristate-Eigenschaft von inaktiven Bausteinen!).

Steuerbus: Mit ihm koordinieren sich CPU und anderes Teilsystem (z.B. "Schreiben/Lesen erwünscht", "Daten liegen an" usw.).



Anmerkung: Bussysteme sind meist inkompatibel aufgrund unterschiedlicher Protokolle.

6.1.3 Aufbau eines Mikroprozessors



Erklärung 6.4 (Aufbau eines Mikroprozessors)
Prinzipiell gibt es zwei Teile: das Steuerwerk und das Rechenwerk.

Rechenwerk: Im Rechenwerk sind das Registerfeld, die temporären Operandenregister und die ALU mit Statusregister.

Registerfeld: Vom Steuerwerk kontrolliert kann ein Register lesend oder schreibend auf den internen Datenbus gelegt werden.

Operandenregister: Sie dienen zur Zwischenspeicherung der Operanden für die ALU vom internen Datenbus.

ALU: Sie verknüpft die Werte der Operandenregister und legt das Resultat auf den internen Datenbus und in das Statusregister (z.B. das Carry-Bit). Die Operation wird durch das Steuerwerk bestimmt.

Statusregister: In ihm wird Statusinformation (z.B. ein Bit für Übertrag) abgelegt. Diese Information kann für Verzweigungen genutzt werden.

Steuerwerk: Das Steuerwerk besteht aus der Mikroprogrammsteuerung und dem Leitwerk.

Leitwerk: Das Leitwerk besteht aus Programmzähler, Instruktionsregister und Adreßregister. Alle diese Register können lesend und schreibend auf den internen Datenbus zugreifen; PC und AR können auch auf den internen/externen Adreßbus gelegt werden.

Programmzähler: Er gibt an, an welcher Adresse im Hauptspeicher die nächste auszuführende Instruktion steht. Er wird meist inkrementiert.

Instruktionsregister: Es enthält den Kode der aktuell auszuführenden Instruktion.

Adreßregister: Es dient zur Adressierung von Daten im Hauptspeicher.

Mikroprogrammsteuerung: Abhängig vom Statusregister und dem Instruktionsregister wird der Mikroprogrammzähler vorbelegt (über ein ROM). Der Mikroprogrammzähler adressiert das Mikroprogramm—ROM. Die Datenwerte dieses ROMs sind Steuerbits für die diversen Einheiten des MP und für die Durchschaltung von Datenwegen (z.B. zur Kopplung interner mit externen Leitungen). Durch Sequenzen von Mikroprogrammschritten wird eine Maschineninstruktion ausgeführt.

Definition 6.5 (Befehlszyklus)
Der durch den Mikroprogrammspeicher definierte interne Ablauf bei der

Ausführung einer Maschineninstruktion heißt *Befehlszyklus*.
Prinzipiell gibt es eine Hierarchie von Zyklen:

Mikroprogrammzyklus ist die Ausführung einer Mikroinstruktion bei jedem Taktimpuls des Prozessors.

Maschinenzyklus ist eine Teiloperation eines Befehls (Holen eines Operanden usw.).

Befehlszyklus umfaßt die Abarbeitung eines kompletten Befehls inklusive des Holens der Operanden und Ablegen des Resultats.

Beispiel 6.6 (Stark vereinfachter Befehlszyklus)
Es soll eine Addition mit einem externen Operand durchgeführt werden. Der Operationscode steht im ersten Befehlswort, die Adresse des externen Operands im zweiten.
Dafür könnte ein Befehlszyklus wie folgt aussehen (in Pseudo-PASCAL, der Hauptspeicher sei wortweise organisiert (16Bit)):

1	IR:=Hauptspeicher[PC]; PC:=PC+1;	Lesen des ersten Befehlsworts
2		Decodierung (Mikroprogrammsequenz anspringen)
3	AR:=Hauptspeicher[PC]; PC:=PC+1;	Lesen der Adresse des zweiten Operanden
4	OR2:=Hauptspeicher[AR];	Einlesen des zweiten Operanden in Operandenregister 2
5	OR1:=Registerfeld[i];	i ist durch Mikroprogramm gegeben
6	Registerfeld[i]:=OR1+OR2; SR:=Statusinformation	

Beispiel 6.7 (Maschinenzyklus)
Aus obigem Befehlszyklus wird das Mikroprogramm für Maschinenzyklus 6 näher beschrieben:

1. Adressierung von Registerfeld i und Chipauswahl des Registerfelds;
2. Auswahl des Schreibmodus für Registerfeld;
3. Anlegen des Kodes für Addition an Steuerleitungen der ALU;
4. Durchschalten der Ausgangsleitungen der ALU auf Statusregister und auf internen Datenbus (damit Übernahme in Registerfeld i)

Die Realisierung dieser Steuerung erfolgt durch Adreßleitungen für das Registerfeld, Operationskodeleitungen für die ALU sowie CS- und R/$\overline{W}$-Leitungen für sämtliche beteiligte Register. Jede Leitung wird durch ein Bit im Mikroprogrammwort gesteuert.

Definition 6.8 (n-Bit-Mikroprozessor)
Die Anzahl der Bits auf dem inneren Datenbus bestimmt die Kategorie des Mikroprozessors (bei 16 Bit Breite ein 16-Bit-Prozessor).

6.2 CISC- und RISC-Prozessoren

In diesem Abschnitt werden je ein CISC- und RISC-Modellprozessor vorgestellt. Aus didaktischen Gründen werden idealisierte Prozessoren diskutiert. Die Architektur beider Prozessoren stammt aus Flik, Liebig: Mikroprozessortechnik.

6.2.1 Ein 32Bit-CISC-Prozessor

Anmerkung: Im Grundlagenabschnitt wurde ein simpler 16Bit-Prozessor vorgestellt. Ein 32-bit-CISC-Prozessor hat in der Regel folgende Strukturmerkmale:

- interne 32Bit-Struktur für Register, Datenwege und ALU;
- externer Daten- und Adreßbus mit 32Bit;
- diverse Datentypen (Bit, Byte, Halbwort, Wort usw.);
- viele Adressierungsarten;
- umfangreicher Befehlssatz mit byte- oder halbwortorganisierter Darstellung (1 Befehl=n Byte oder $n \times 16$ Bit);
- Mechanismen zur Ausnahmebehandlung und für Privilegebenen;
- Fließbandbearbeitung von Befehlen (Pipelining);
- interner Daten- und Befehlspuffer (Cache);
- virtuelle Speicherverwaltung (durch spezielle Einheit: MMU);

Anmerkung: Folgende Aspekte des Modell-CISC-Prozessors werden in diesem Abschnitt behandelt.

- Struktur (Registersatz, Datentypen, Adressierungsarten, Befehlsformate);
- Befehlssatz;
- Betriebsarten und Ausnahmebehandlung;

Allgemeine Prozessorstruktur

Anmerkung: Zur Erläuterung benötigen wir in diesem Abschnitt einfache Befehle. Daher wird die Befehlsstruktur kurz diskutiert:
`MOVE s,d` ist ein Transportbefehl, der Information von Quelle s (source) zum Ziel d (destination) transportiert (s und d werden durch die Adressierungsart bestimmt).
Kleinste adressierbare Einheit ist das Byte, es können aber auch Operationen auf Halbworten (16Bit) oder Worten (32Bit) ausgeführt werden (der Prozessor liest und schreibt korrekt in Speicher beliebiger Zugriffsbreite). Das Datenformat wird durch einen Befehlssuffix angegeben (".B" für "Byte", ".H" für "Halbwort", ".W" für "Wort").

Registersatz

Definition 6.9 (Registersatz)
Der *Registersatz* eines Prozessors ist die Menge aller in einem Programm direkt ansprechbaren Prozessorregister.
Register dienen zur Speicherung von Rechengrößen, Adressen und Indices. In einem Spezialregister wird der *Prozessorstatus* abgelegt.

Anmerkung: Ein wichtiges Merkmal einer Architektur ist, ob die Register allgemein einsetzbar sind (Universalregisterarchitektur) oder für spezielle Operationen spezielle Register verwendet werden müssen (Spezialregisterarchitektur).

Erklärung 6.10 (Registersatz des Modellprozessors)
Der Modellprozessor hat folgende Register:

allgemeine Register: Sie sind mit R0 bis R7 bezeichnet. Sie werden unmittelbar adressiert und können Daten und Adressen aufnehmen.

Halbwort- und Bytezugriffe lesen oder ändern die jeweils niederwertigsten Bits eines Registers.

Stackpointerregister: Es gibt hier USP und SSP. Beide adressieren den obersten Eintrag eines Kellers von Einträgen (z.B. Parameter von Funktionen oder Rücksprungadressen von Unterprogrammen).
Spezielle Befehle für Stackpointer: PUSH s, POP d.
Es gibt zwei SP: einen für die normale Betriebsart (User-Modus) und einen für die privilegiert Betriebsart (Supervisor-Modus).

Framepointerregister: Dieses Register zeigt auf die erste Adresse des Datenbereichs der aktuellen Funktion (Kellerrahmen). Damit können Parameter und lokale Daten dieser Funktion relativ zu diesem Register adressiert werden.
Es wird bei jedem Aufruf einer neuen Funktion belegt.

Basepointerregister: Es enthält eine Basisadresse für das aktuelle Programm (zur Adressierung globaler Daten).

Vectorbaseregister: In diesem Register steht die Basisadresse der Tabelle von Sprungvektoren für Traps und Interrupts.

Befehlszähler: Er enthält die Adresse des 1. Bytes des nächsten auszuführenden Befehls. Er wird nach einem Befehlszugriff oder Sprungbefehl entsprechend geändert.

Statusregister: Es ist zweigeteilt: ein Teil enthält Information die nur im privilegierten Modus geändert werden kann, der andere Statusbits für den Normalmodus.

- Statusbits: Carry (Überlauf bei arithmetischen Operationen), Zero (Resultat=0), oVerflow (Zahlenbereich bei Zweierkomplementrechnung zu klein) und Negative (Resultat< 0);
- Supervisorbits: im Supervisormodus (Prozessor im privilegierten Betriebsmodus), T0,T1 (schrittweise Abarbeitung von Programmen) und IM0-IM2 (Prioritätsstufe des aktuellen Programms);

Anmerkung: Bei einem Prozeßwechsel oder einer Ausnahmebehandlung werden Befehlszähler und Statusregister automatisch auf dem Supervisorkeller abgelegt (zur späteren Wiederaufnahme der Bearbeitung). Die anderen Register können nach Bedarf gesichert werden.

Datentypen, Zugriffsarten auf Daten und Bytewertigkeit

Definition 6.11 (Datentypen)
Datentypen (im Kontext von Prozessoren) sind Informationskategorien, die durch Befehle des Prozessors direkt verarbeitet werden können.
Diese Kategorien sind gekennzeichnet durch ihre Länge in Bits (*Datenformat*) und die *Interpretation* dieser Bits.

Anmerkung: Der Modellprozessor kennt die Standarddatentypen Byte (8 Bit), Halbwort (16 Bit), Wort (32 Bit) und Doppelwort (64 Bit).
Darauf aufbauend gibt es folgende Datentypen:

Datentyp	Interpretation	Datenformat
Wahrheitswert	0=falsch, 1=wahr	1 Bit
Bitvektor	Menge von Werten oder Feld von Wahrheitswerten	$\leq$ 32 Bit
BCD-Zahlen gepackt	2, 4, oder 8 Halbbytes	8, 16, 32, 64 Bit
positive Ganzzahl	als Dualzahl	8, 16, 32, 64 Bit
Ganzzahl	als Dualzahl im Zweierkomplement	8, 16, 32, 64 Bit
Strings	Zeichen- oder Zahlenfelder	aufeinanderfolgende Bytes Halbworte oder Worte
Fließpunktzahl	IEEE754-Format mit Mantisse und Exponent	32, 64 Bit

Erklärung 6.12 (Fließpunktzahlen)
Dualzahlen können halblogarithmisch dargestellt werden: Führendes Bit ist das *Vorzeichen*, danach folgen e Bits *Exponent* und m Bits *Mantisse*. Die Darstellung wird wie folgt interpretiert:

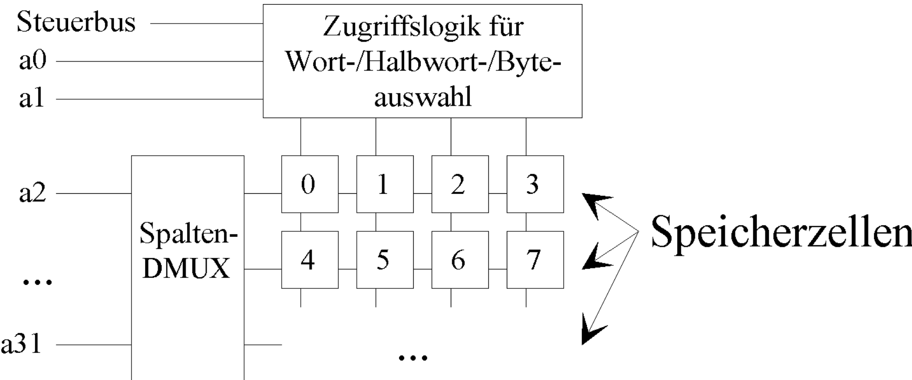
$$Wert := (-1)^{\text{Vorzeichen}} \cdot (1 + \frac{\text{Mantisse}}{2^m}) \cdot 2^{\text{Exponent}-\text{Offset}}, \text{Offset} := 2^{e-1} - 1$$

Bei einer 32 Bit-Fließpunktzahl sind $m = 23, e = 8$, bei 64 Bit $m = 52, e = 11$.
Die größten und kleinsten Werte von Exponent sind für Sonderzahlen (Null, unendlich, undefiniert) und denormalisierte Darstellungen reserviert.
Beispiel:

$$\underbrace{1}_{\text{Vorzeichen}}\ \underbrace{10000001}_{\text{Exponent}}\underbrace{01000000000000000000000}_{\text{Mantisse}} = -5$$

Erklärung 6.13 (Zugriffsarten auf Daten)
Bei unserem Modellprozessor können einzelne Bytes im Speicher angesprochen werden, der Speicher ist *byteadressiert*.
Prinzipiell kann der Prozessor aber in einem Zugriff auf Einheiten in Datenbusbreite zugreifen (d.h. auf Worte); die Zugriffsbreite kann aber bei Bedarf reduziert werden (automatic bus sizing).

Erklärung 6.14 (Sinn von Datenausrichtung)
Wenn Halbwortinformation so im Speicher abgelegt ist, daß sie die Adressen 4k+3 und 4(k+1) belegt, so muß der Prozessor zwei Zugriffe machen, da er in einem Zugriff nur Adressen mit gleichem k ansprechen kann (analoge Überlegung gilt für Worte mit Startadresse $\neq 4k$). Eine derartige Zuordnung von Operanden zu Speicheradressen nennt man *Fehlausrichtung* (misalignment).



Korrekt ausgerichtete Daten lassen sich schneller lesen und schreiben, da die Zugriffslogik oft wortweise auf den Speicher zugreift (siehe Bild). Bei Zugriffen über Wortgrenzen hinweg muß man den Speicher zweimal adressieren.

Definition 6.15 (Bytewertigkeit)
Werden die Bytes von Wort- oder Halbwortoperanden mit höherer Stelligkeit an höheren Adressen im Speicher abgelegt, spricht man von einem *Prozessor mit aufsteigender Bytewertigkeit* oder *little-endian byte ordering* anderenfalls von *fallender Bytewertigkeit* oder *big-endian byte ordering*.

Anmerkung: Unterschiedliche Bytewertigkeit führt zu Problemen bei der naiven (!) Übertragung von Binärinformation zwischen Maschinen.

Adressierungsarten

Anmerkung: Moderne Maschinen bieten diverse Möglichkeiten, die Adresse eines Operanden eines Befehls anzugeben. Gründe für die Vielzahl von Adressierungsarten sind Anforderungen höherer Programmiersprachen und systemtechnische Überlegungen wie Unterstützung von frei verschiebbaren Programmkode (mit relativer Adressierung).

Definition 6.16 (Adressierungsarten)
(Im folgenden wird in Pseudo-PASCAL-Notation die tatsächliche Operandenadresse beschrieben.)
Der Modellprozessor unterstützt folgende Adressierungsarten

Angabe einer Konstante (immediate): Der Operand steht als Konstante im Befehl. Derartige Operanden können nur Quellen sein.
Assemblerschreibweise: # vor der Konstante
Beispiel: `MOVE.W #176,R0` (mit Bedeutung: R0:=176;)

absolute Adresse (direct): Die Adresse des Operanden steht als Konstante im Befehl.
Assemblerschreibweise: keine (nur Zahl oder Name)
Beispiele: `MOVE.B 176,R0` (mit Bedeutung: R0:=Hauptspeicher[176];)
`MOVE.B #2,176` (mit Bedeutung: Hauptspeicher[176]:=2;)

Registeradresse (register): Der Operand ist ein Register.
Assemblerschreibweise: symbolischer Name des Registers
Beispiel: `MOVE.W #176,FP` (mit Bedeutung: Framepointer:=176;)

registerindirekte Adressierung (register indirect): Die Operandenadresse steht in einem Register. Das Register enthält einen *Zeiger* auf den Operanden.
Assemblerschreibweise: Registername ist eingeklammert
Beispiel: `MOVE.H (R1),R0` (mit Bedeutung: R0:=Hauptspeicher[R1..R1+1];)

registerindirekte Adressierung mit Prädekrement

oder Postinkrement (register indirect with autode-/increment): Vor der Operation wird der Registerinhalt um die Bytezahl des Operanden (1,2 oder 4) verringert bzw. danach erhöht. Zur Operation wird der Inhalt des Registers wiederum als Operandenadresse interpretiert.
Assemblerschreibweise: - vor Klammer bzw. + nach Klammer

Beispiel: `MOVE.W #0,(R0)+` (mit Bedeutung: Hauptspeicher[R0..R0+3]:=0; R0:=R0+4;)
Anwendungsbeispiel: Füllen von Speicherbereichen oder Kopieren

```
LOOP:
    MOVE.B (R1)+,R2
    MOVE.B R2,(R0)+
    CMP.B R2,#0
    BNE LOOP
```

registerindirekte Adr. mit Offset (r.i. with displacement): Die Operandenadresse ergibt sich aus Registerinhalt und einem konstanten Offset (anderer Begriff: basisrelative Adresse).
Assemblerschreibweise: Zahl oder Name vor Klammer
Beispiel: `MOVE.B #0,22(R0)` (mit Bedeutung: Hauptspeicher[R0+22]:=0;)
Anwendungsbeispiel: Variablenadressierung in Prozeduren

PROCEDURE P();
    VAR
        x : BYTE;
        y : ARRAY [0..2] OF INTEGER;
        z : BYTE;
        t : INTEGER;
    BEGIN
        x:=z;
    END P;
	x EQU 0 Offset von x = 0
	    y EQU 2 Ausrichtung!!
	    z EQU 8
	    t EQU 10
	BEGIN_P:
	    // Framepointer wird vorbelegt
	    ...
	    MOVE z(FP),x(FP)

indizierte Adressierung (indexed): Die Operandenadresse ergibt sich aus einer Anfangsadresse mit Offset, erhöht um den mit 1, 2 oder 4 skalierten Wert eines Registers. Der Registerwert kann als Index eines Feldelements betrachtet werden.
Assemblerschreibweise: Anfangsadresse ergänzt um geklammertes Indexregister
Beispiel: `MOVE.W #33,x(R0)(R1*4)` (mit Bedeutung: Hauptspeicher[(R0+x)+4*R1..(R0+x+3)+4*R1]:=33;)

Anwendungsbeispiel: Feldzugriff in Prozedur P aus vorigem Abschnitt; die Zuweisung `y[x]:=t;` würde zu `MOVE.H x(FP),R1; MOVE.H t(FP),y(FP)(R1*2)`

speicherindirekte Adressierung (memory indirect): Die Operandenadresse ergibt sich als Inhalt einer Speicherzelle. Diese ist wiederum ein *Zeiger* auf den Operanden. Häufig wird hierbei eine registerindirekte Adressierung mit Offset um ein weiteren Offset ergänzt.
Assemblerschreibweise: eckige Klammer um adressierende Speicherzelle
Beispiel: `MOVE.B R1,[ptr(FP)]k` (mit Bedeutung: Zeiger:=Hauptspeicher[FP+ptr]; Hauptspeicher[Zeiger+k]:=R1;)
Anwendungsbeispiel: Adressierung von Zeigern auf Verbunde (Records)

```
TYPE T=RECORD
    x, z : BYTE;
    q : ARRAY[0..2] OF BYTE;
END T;

PROCEDURE Q(VAR a, b : T)
BEGIN
    a.z:=b.x;
END Q;
```

```
    x EQU 0     Offset von x in T
    z EQU 1     Offset von z in T
    q EQU 2     Offset von q in T
    a EQU 0     Offset von a in Q
    b EQU 5     Offset von b in Q
BEGIN_Q:
    // Framepointer wird vorbelegt
    ...
    MOVE.B [b(FP)]x,[a(FP)]z
```

Vor- und Nachindizierung bei sp.ind. Adressierung: Dies ist eine Variation obiger Adressierung. Dabei wird der skalierte Wert eines Indexregisters vor oder nach dem Indirektionsschritt aufaddiert.
Assemblerschreibweise: skaliertes Register in eckige Klammer oder dahinter
Beispiel: `MOVE.B #3,[ptr(FP)](R1*1)k` (mit Bedeutung: Zeiger:=Hauptspeicher[FP+ptr]; Hauptspeicher[Zeiger+k+R1]:=3;)
Anwendungsbeispiel: In obiger Prozedur Q würde die Zuweisung `b.q[b.x]:=5;` zum Assemblercode `MOVE.B [b(FP)]x,R1; MOVE.B #5,[b(FP)](R1*1)q`.

befehlszählerrelative Adressierung (PC relative): Die Operandenadresse ergibt sich mittelbar aus dem Befehlszähler. In der Regel wird ein Offset addiert (bei Sprungbefehlen) und eventuell eine Indizierung vorgenommen (bei Sprungtabellen).

Assemblerschreibweise: analog zu anderen Registern
Anwendungsbeispiel: CASE-Anweisung in PASCAL

```
VAR x : 0..5;
BEGIN
    CASE x OF
        0,3 : x:=x+2;
        1,2 : x:=x-1;
    END (* CASE *)
END
```

```
x EQU 0
MOVE.W x(FP),R1
JUMP 4(PC)(R1*4)     Sprungbefehl habe Länge 4
JUMP CASE_03         für x=0
JUMP CASE_12         für x=1
JUMP CASE_12         für x=2
JUMP CASE_03         für x=3
JUMP CASE_END        für x=4
JUMP CASE_END        für x=5
CASE_03:
...                  Code für x:=x+2
JUMP CASE_END
CASE_12:
...                  Code für x:=x-1
CASE_END:
```

Befehlsformat

Definition 6.17 (Befehlsformat)
Die Abbildung von Maschinenbefehlen und deren Operanden auf Bitmuster in aufeinanderfolgenden Bytes, Halbworten oder Worten im Speicher nennt man *Befehlsformat des Prozessors.*
In der Regel haben Befehle variable Länge. Diese Länge wird bestimmt durch

- die Anzahl der Operanden (man spricht von Ein-, Zwei- oder Dreiadreßbefehlen),
- deren Adressierungsarten und
- Spezialitäten (z.B. Information über Befehlswiederholung)

Beispiel 6.18 (Befehlsformate)
Vergleichend werden die Formate von Intel- und Motorola-Prozessoren betrachtet:

Motorola 680X0: Die Befehle werden in ein bis fünf Halbworten abgelegt. Im ersten Halbwort steht der Befehlskode und in je 3 Bit Adressierungsmodus und beteiligtes Register (R0–R7). Sofern Quell- und/oder Zieloperand eine zusätzliche Information benötigen, werden maximal je zwei Halbworte angefügt.
Beispiel für ADD-Befehl mit mindestens einem Registeroperanden:

$$\text{Bit}: \underbrace{16\ 15\ 14\ 13}_{\text{ADD}-\text{Code}}\underbrace{12\ 11\ 10}_{R_A}\ \underbrace{9\ 8}_{\text{Wortbreite(B/H/W)}}\ \underbrace{7}_{\text{R}_\text{A}=\text{dest?}}\ \underbrace{\underbrace{6\ 5\ 4}_{Art}\underbrace{3\ 2\ 1}_{R_B}}_{\text{Operand B}}$$

Bit 7 legt fest, ob R_A Quelle oder Ziel ist. Nur Operand B kann komplexer adressiert werden.

Intel 80X86/Pentium: Die Befehle werden in ein bis fünfzehn Bytes abgelegt.
Der Befehlskode benötigt ein bis zwei Bytes, Adreßmodifikation und skalierte Indizierung 0 bis zwei Bytes und ein Offset und eine Konstante als Operand je vier Bytes.
Vor den Operationskode können vier Modifikationsbytes vorangestellt werden (z.B. für Nichtunterbrechbarkeit und zur Befehlswiederholung).

Anmerkung: Die genaue Art der Abbildung spielt lediglich eine Rolle für Entwickler von Assemblierern und Disassemblierern. Daher wird für den Modellprozessor kein Befehlsformat definiert.

Befehlssatz

Definition 6.19 (Befehlssatz)
Ein *Befehlssatz* ist die Gesamtheit der Befehle, die ein Prozessor ausführen kann. Man unterteilt Befehle in *Befehlsgruppen* von Befehlen mit ähnlicher Funktion.

Anmerkung: Es gibt zwei Entwurfsphilosophien:

Systeme mit reduzierten Befehlssätzen (RISC): Sie haben einfache Befehle, die in einem Maschinenzyklus ausführbar sind.
Weitere Eigenschaften: Load/Store-Befehle, viele gleichartige Register, Pipelining.

Systeme mit umfangreichen Befehlssätzen (CISC): Sie haben komplexe Befehle mit ausgefeilten Adressierungsarten.
Weitere Eigenschaften: unterschiedliche Zykluszeiten pro Befehl, komplexe interne Struktur.

Definition 6.20 (orthogonaler Befehlssatz)
Ein *orthogonaler Befehlssatz* ist so strukturiert, daß es möglichst wenig Einschränkungen gibt in der Kombination von

- Befehl,
- Adressierung des Quell- und Zieloperanden sowie
- Datentyp und Datenformat.

Anmerkung: Orthogonalität ist nützlich, wenn Assemblercode generiert wird (z.B. durch einen Compiler). Aus Designgründen ist sie nicht immer möglich.

Erklärung 6.21 (Befehlssatz des Modellprozessors)
Bei unserem Modellprozessor gibt es folgende Befehlsgruppen:

- Transportbefehle,
- arithmetische und logische Befehle,
- Rotations- und Schiebebefehle,
- Sprungbefehle,
- Systembefehle und
- Befehle für Mehrprozessorbetrieb.

Wir gehen davon aus, daß der Befehlssatz weitgehend orthogonal ist, d.h. jede Adressierungsart ist bei einem Befehl erlaubt (aber keine Konstanten als Zieloperanden!).
Die Bedeutung wird in Pseudo-PASCAL beschrieben; Zahlen in spitzen Klammern (z.B. FP≪9≫) sind Bitpositionen einer Speicherzelle.
Änderungen auf Statusregisterbits werden durch Angabe des Namens des geänderten Bits (N/Z/V/C) gekennzeichnet und des Werts (0/1/?).

Transportbefehle

MOVE s,d	move (d:=s)	N,Z,V0,C0
MOVSX s,d	move sign extended (d:=s(mit Ergänzung um höchstes Bit von s))	N,Z,V0,C0
MOVZX s,d	move zero extended (d:=s(mit Ergänzung um Nullen))	N,Z,V0,C0
LEA s,d	load effective address (d:=ADR(s)), s kein konstanter Operand und kein Register	—

Der MOVE-Befehl kopiert von Quelle nach Ziel für beliebige Operanden (mit fallender Bytewertigkeit). Die Befehle mit Ergänzung erweitern einen zu kurzen Quelldatenwert mit 0- oder 1-Bits, sodaß er in das Ziel paßt (bei MOVSX um Vorzeichen, bei MOVZX um 0-Bits); es wird die Länge beider Operanden als Suffix angegeben. LEA lädt die Adresse eines Operanden als Wort in den Zieloperanden.

Beispiel 6.22 (Transportbefehle)
Sei Hsp[10]=FA_{16}/ und Hsp[11]=7. Dann wirken folgende Befehle so:

MOVSX.B.H 10,R0	R0:= (R0 AND $FFFF0000_{16}$) OR $FFFA_{16}$;
MOVZX.B.W 10,R1	R1:= $000000FA_{16}$;
MOVE.H 10,R2	R2:=(R2 AND $FFFF0000_{16}$) OR ($FA07_{16}$);
LEA 11(R1),R3	R3:=$0000010B_{16}$;

Anmerkung: Bei Prozessoren ohne registerindirekte Adressierung mit Prädekrement und Postinkrement gibt es die Transportbefehle PUSH und POP, die Daten auf die durch den aktuellen SP adressierten Speicherzellen ablegen und diesen erhöhen oder erniedrigen.

Arithmetische und logische Befehle

ADD s,d	add; d:=s+d;	N, Z, V, C
ADDC s,d	add with carry; d:=s+d+C;	N, Z, V, C
SUB, SUBC	analog	N, Z, V, C
MULU s,d	multiply unsigned; $d_W := d_H * s_H$; bzw. $d_{DW} := d_W * s_W$;	N, Z, V, C0
DIVU s,d	divide unsigned; $(\text{Quotient}, \text{Rest}) := d_W/s_H$; $d_W :=$ Quotient$*2^{16}$+Rest; bzw. $(\text{Quotient}, \text{Rest}) := d_{DW}/s_W$; und $d_{DW} := \text{Quotient} * 2^{32} + \text{Rest}$;	N, Z, V, C0
MULS, DIVS	multiply bzw. divide signed; analog für Zweierkomplementzahlen	N, Z, V, C0
CMP s,d	compare; setze SR aus d-s;	N, Z, V, C

Die Additions- und Subtraktionsbefehle operieren auf beliebigen Ganzzahlen unter eventueller Berücksichtigung des Carry-Bits C. C ist Übertrag; bei Zahlen im Zweierkomplement liefert V Information über den Überlauf des Zahlenbereichs.

Die Multiplikation faßt zwei Halbwortoperanden zu einem Wortergebnis zusammen beziehungsweise zwei Wortoperanden zu einem Doppelwortergebnis. Der Divisionsbefehl benötigt einen Zieloperanden, der doppelt so lang ist wie sein Partner. Rest und Quotient werden nacheinander im Zieloperand abgelegt. Beide Operationen gibt es auch für Zahlen im Zweierkomplement.

Der CMP-Befehl vergleicht d mit s und setzt das Statusregister.

Anmerkung: Moderne Prozessoren können auch BCD-Zahlen verarbeiten und von und nach Binärzahlen konvertieren.

Beispiel 6.23 (Multiplikation, Division)
Sei R1=400, R2=0, R3=300 und R4=7. Dann wirken die Befehle wie folgt:

MULU.H R1,R3	R3:=120000;
DIVU.W R4,R2	R2:=Rest$((0 \cdot 2^{32} + 120000)/7)$=6; R3:=Quotient$((0 \cdot 2^{32} + 120000)/7)$=17142;

AND s,d	and; d:=s AND d;	N, Z, V0, C0
OR, XOR	analog	N, Z, V0, C0
NOT d	not; d:=NOT d; Einerkomplement	N, Z, V0, C0

Diese Befehle führen die vorgegebene Operation bitweise auf den Operanden aus und dienen meist zur Maskierung oder Zusammenfassung der Operanden.

Rotations- und Schiebebefehle Bei allen Rotations- und Schiebebefehlen gibt s die Zahl der Schiebeschritte an. Der Wert des Bits, das bei der Rotation oder Verschiebung zuletzt den Operanden verlassen hat, wird im Carry gespeichert.

ROL s,d	rotate left; Verschiebung nach links und Nachziehen der herausgeschobenen Bits	N, Z, V0, C
LSL s,d	logical shift left; Verschiebung nach links und Nachziehen von 0	N, Z, V0, C
ROR, LSR	jeweils Operation analog nach rechts	N, Z, V0, C
ASL s,d	LSL s,d, aber Interpretation des Überlaufs	N, Z, V, C
ASR s,d	arithmetic shift right; Verschiebung nach rechts und Nachziehen des Vorzeichenbits	N, Z, V, C

Die arithmetische Verschiebung entspricht einer Multiplikation mit 2^n bzw. 2^{-n} für Zweierkomplementzahlen; die logische Verschiebung nach rechts erhält nicht das Vorzeichen einer Zweierkomplementzahl ist aber korrekt für Zahlen ohne Vorzeichen.

Beispiel 6.24 (Verschiebungen)
Sei Hsp[42]=10001010_2. Als Zweierkomplementzahl entspricht das -118, als Zahl ohne Vorzeichen 138. Bei Division durch 8 erhält man -15 (Rest 2) bzw. 17 (Rest 2).

ASR.B #3,42	Hsp[42]:=11110001_2=-15 bzw.=232
LSR.B #3,42	Hsp[42]:=00010001_2=17.

Für Hsp[43]=01000000_2=64 wird bei einem arithmetischen Linksshift um 1 das Überlaufbit V gesetzt, da sich das Vorzeichen ändern würde.

Sprungbefehle Es gibt unbedingte und bedingte Sprünge. Bei beiden ist der Operand die Adresse, die angesprungen wird, im zweiten Fall aber nur, wenn die Statusregisterbits bestimmte Werte haben. Diese Bits werden oft durch vorhergehende arithmetische Befehle gesetzt.

Meistens wird die befehlszählerrelative Adressierung verwendet, weil das Programm dann verschiebbar ist.

JMP d	jump; unbedingter Sprung;PC:=d	-
Bbed d	branch conditionally; bedingter Sprung; wenn Bedingung erfüllt	-
BGT, BGE, BLE, BLT	branch on greater, greater or equal… für vorzeichenbehaftete Zahlen	-
BHI, BHS, BLS, BLO	branch on higher, higher or same… für Ganzzahlen	-
BEG, BNE	branch on equal, not equal	-

Beispiel 6.25 (Sprungbefehle)
Sei X=00000001_2 und Y=10000000_2.

CMP.B Y,X   berechnet X-Y und setzt die Statusbits: Z=0, C=1, N=1, V=1

Ein BGT würde danach ausgeführt (X ist als Ganzzahl größer als Y), aber ein BHI nicht (weil X ist als vorzeichenbehaftete Zahl kleiner als Y).

Spezialsprünge sind Unterprogrammsprünge und Rückkehrsprünge von Unterprogrammen oder Ausnahmebehandlungen.

JSR d	jump to subroutine (Unterprogrammsprung): Hauptspeicher[SP..SP+3]:=PC; SP:=SP+4; PC:=d	-
RTS	return from subroutine (Unterprogrammrückkehrsprung): SP:=SP-4; PC:=Hauptspeicher[SP..SP+3];	-
RTE	return from exception (Ausnahmerückkehrsprung): SSP:=SSP-2; SR:=Hauptspeicher[SSP..SSP+1]; SSP:=SSP-4; PC:=Hauptspeicher[SSP..SSP+3];	-

Beim Unterprogrammsprung JSR wird der aktuelle Befehlszählerinhalt (Adresse des aus JSR folgenden Befehls) als 32Bit-Wert auf den Keller gespeichert (je nach Betriebsmodus auf User- oder Systemkeller). Der Kellerzeiger wird erhöht. Beim Unterprogrammrücksprung wird der Kellerzeiger verringert und der Wert des Befehlzählers restauriert.

Nach Ausnahmebehandlungen wird sowohl Statusregister als auch Befehlszähler vom Systemkeller durch RTE restauriert. Dies ist ein privilegierter Befehl.

Systembefehle Mit Systembefehlen werden Änderungen des Systemzustands erreicht oder spezielle Register gesetzt oder gelesen.

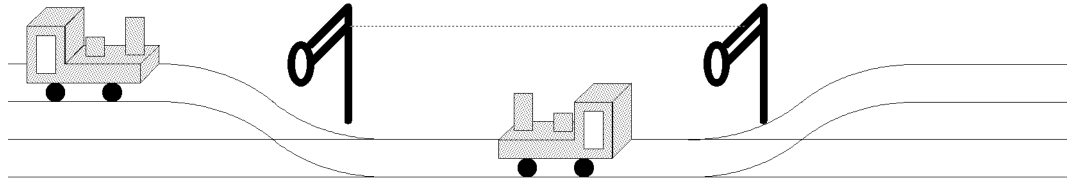
MOVSR SR,d MOVSR s,SR	move status register: d:=SR; bzw. SR:=s	-
MOVCC CC,d MOVCC s,SR	move condition codes: d:=SR«7-0»; bzw. SR«7-0»:=s;	-
MOVUSP USP,d MOVUSP s,USP	move user stack pointer: analog	-
TRAP #n	trap (Ausnahmebehandlung): Hauptspeicher[SSP..SSP+3]:=PC; SSP:=SSP+4; Hauptspeicher[SSP..SSP+1]:=SR; SSP:=SSP+2; Zeiger:=VB+4*n; PC:=Hauptspeicher[Zeiger..Zeiger+3];	-

Die diversen MOV??-Befehle erlauben lesenden und schreibenden Zugriff auf Statusregister oder Benutzerkellerzeiger. Zugriffe auf das komplette SR und USP sind nur im Systemmodus möglich.

Durch TRAPs wird ein Übergang in den Systemmodus durchgeführt. Dabei wird der Status und der Befehlszähler gekellert. Es gibt mehrere durch das Vektorbasisregister gegebene Sprungziele für unterschiedliche Ausnahmen. Eine Ausnahme ist oft der Sprung ins Betriebssystem.

Befehle für Mehrprozessorbetrieb In modernen Systemen können mehrere Prozessoren parallel auf gemeinsame Betriebsmittel zugreifen (z.B. auf Speicher). Diese Zugriffe müssen koordiniert werden.

Dazu dienen Synchronisationsbefehle. Ihre Zugriffe können nicht durch andere Prozessoren unterbrochen werden. Sie arbeiten oft wie Semaphore (Flügelsignale) bei der Eisenbahn (im Bild sind die Flügelsignale mechanisch gekoppelt, sodaß nur eine Lok die eingleisige Strecke befahren kann).



TAS d CAS R_a, R_b, d	test and set: setze SR aus d-0; d:=11...11; compare and swap: setze SR aus d-R_a; wenn Z=1 dann d:=R_b sonst R_a:=d	N, Z, V0, C0 N, Z, V, C

TAS prüft, ob der Operand Null ist, setzt ihn auf jeden Fall auf -1. Der Zugriff auf den Operanden kann nicht durch einen anderen Prozessor unterbrochen werden. CAS bietet eine ähnliche Funktion wie TAS (ohne Diskussion).

Beispiel 6.26 (Sinn der Ununterbrechbarkeit)
Ein System habe nur einen Drucker, aber mehrere Prozessoren. Der Drucker sei frei, wenn die Zelle BELEGT=0 ist. Folgendes Programm klappt nicht, um den Drucker zu belegen:

NOCHMAL: CMP.B BELEGT,#0
BNE NOCHMAL ; nicht frei, wenn ≠ 0
MOVE.B #FF, BELEGT ; belegen
. . . drucken . . .
MOVE.B #0, BELEGT ; freigeben

Problem: es können gleichzeitig mehrere Prozessoren feststellen, daß BELEGT=0 ist. Die Ununterbrechbarkeit von TAS hilft:

NOCHMAL: TAS.B BELEGT
BNE NOCHMAL ; nicht frei, wenn ≠ 0
. . . drucken . . . ; BELEGT ist bereits ≠ 0
MOVE.B #0, BELEGT ; freigeben

Anmerkung: In Programmiersprachen für parallele Systeme wird immer ein Konstrukt zum gegenseitigen Ausschluß von Prozessen angeboten. Manchmal gibt es einen Datentyp "Semaphor" mit Operationen "belegen" und "freigeben" oder ein Schleifenkonstrukt (in dem anfangs belegt und am Ende freigegeben wird).

Spezielle Betriebsarten

Ausnahmen Ein Mikroprozessor kann Ausnahmesituationen behandeln (*exceptions*). Diese können auf unterschiedliche Weise entstehen:

1. durch einen Fehler bei der Abarbeitung eines Befehls (z.B. Teilen durch 0);
2. durch einen TRAP-Befehl;

3. prozessorextern und befehlssynchron (z.B. einen fehlerhaften Buszyklus);

4. durch ein asynchrones Signal (z.B. von E/A-Einheiten).

Asynchrone Signale von außen (Fall 4) heißen *Interrupts*, die anderen *Traps*.

Erklärung 6.27 (Ablauf bei Auftreten einer Ausnahme)
In Ausnahmesituationen wird

1. Die aktuelle Befehlsabarbeitung wird unterbrochen.

2. Der Statusregisterinhalt wird prozessorintern gemerkt.

3. Der Prozessor schaltet in den Systemmodus und löscht eventuelle Tracebits.

4. Auf dem Systemkeller werden das gemerkte SR und der Befehlszählerstand abgelegt.

5. Es wird über eine Sprungtabelle in ein spezielles Unterprogramm verzweigt. Das VB-Register zeigt auf den Anfang dieser Sprungtabelle, die Ausnahmenummer ist ein (um die Adreßbreite zu skalierender) Offset.

6. Je nach Art der Ausnahme wird die Prioritätsstufe belegt in Statusbits IM0 bis IM2. Nur Ausnahmen einer höheren Prioritätsstufe unterbrechen die aktuelle Ausnahmebearbeitung.

7. Nach Abarbeitung der Ausnahmebehandlung wird an der Unterbrechungsstelle fortgesetzt.

Folgende Ausnahmen werden in der Regel behandelt (mit fallender Priorität):

- RESET (Rücksetzsignal);

- Busfehler: bei Ausbleiben der Quittierung beim Lesen im Speicher;

- Einzelschrittmodus (gekennzeichnet durch gesetzte Tracebits im SR);

- Befehlskodefehler;

- Teilen durch 0;

- explizite Traps (benutzt als Schnittstelle zum Betriebssystem): Sie haben eine Nummer als Parameter, die als zusätzlicher Offset in die Sprungtabelle benutzt wird (d.h. es gibt mehrere Einträge für Traps);

- Interrupts: für Anforderungen externer Geräte; Analog zu Traps gibt es mehrere Einträge für Interrupts. Die Nummer eines Interrupts wird als Kode an Interrupteingängsleitungen des Prozessors übermittelt (Kode 0=kein Interrupt). Interrupts sind untereinander umso höher priorisiert, je höher ihre Nummer ist.

Benutzer- und Systemmodus In modernen Systemen möchte man Benutzerprozesse so abschotten, daß sie weder andere Prozesse noch den Systemkern (zer)stören können. $\rightsquigarrow$ Hardwareunterstützung für Schichten mit unterschiedlichen Privilegien.

Bei zwei Privilegebenen wird der Betriebsmodus des Prozessors durch ein Bit im Statusregister gekennzeichnet. Diese Modusunterscheidung wird wie folgt genutzt:

- durch eine Speicherverwaltungseinheit (MMU): Sie stellt fest, ob die Adresse im jeweiligen Betriebsmodus lesbar, schreibbar oder ausführbar ist. Bei Privilegverletzung wird eine Ausnahme ausgelöst.
- durch getrennte Keller für Betriebsmodi: Damit ist eine Manipulation des Systemkellers durch einen Benutzerprozeß unmöglich.
- durch geschützte Befehle für Änderung des Betriebsmodus: Der Übergang von Benutzer in Systemmodus findet nur durch Ausnahmen statt.

Anmerkung: Gängige Prozessoren haben zwei (680XX, SPARC, PowerPC, MIPS) oder vier Privilegebenen (80X86).

6.2.2 Ein 32Bit-RISC-Prozessor

Anmerkung: RISC-Prozessoren besitzen im Gegensatz zu CISC-Prozessoren häufig folgende Strukturmerkmale:

- simplere Befehlsstruktur; $\rightsquigarrow$ Bearbeitung in einem Takt;
- einfachere Adressierungsarten; load-store-Architektur;
- gleichförmige Befehlslänge und einfaches -format;
- mehr Register mit komplexer Adressierung (z.B. durch Registerfenster);

- starkes Pipelining (forciert durch Umordnung von Befehlen; dabei teuer: HSP-Zugriffe);

Im folgenden werden die Spezialitäten des RISC-Modellprozessors analog zur Gliederung bei CISC diskutiert.

Allgemeine Prozessorstruktur

Registersatz

Anmerkung: RISC haben neben allgemeinen Registern nur Befehlszähler und Statusregister. Es fehlen Spezialregister, insbesondere die Kellerzeiger. Verarbeitende Befehle (z.B. Addition) arbeiten nur registersatzintern.

Um Pipelining zu ermöglichen, werden Parameter an Unterprogramme bevorzugt in Registern übergeben.

Erklärung 6.28 (Gängige Registerstrukturen)
Bei RISCs sind zwei Registerstrukturen gebräuchlich:

Registersatz ohne Struktur: Es gibt eine große Zahl von Registern (Zweierpotenz z.B. 32), wobei mindestens zwei eine Spezialfunktion haben (Konstante-0-Register und Register für Rücksprungadresse). RSA-Register ist nötig, weil SP fehlt; Konstante-0-Register wird insbesondere als Zielregister benötigt, wenn durch eine Operation nur Statusbits gesetzt werden sollen.
Problem: Parameterübergabe an Unterprogramm kompliziert, weil meist Auslagerung von Registerinhalten nötig ist.

Registersatz mit Fenstern: Jedes Unterprogramm X hat Zugriff auf eine feste Zahl n von Registern (numeriert von 0 bis n-1). Ein derartiger Ausschnitt heißt *Fenster*. Dabei sind n_I Register für Parameter von X (I-Register), n_L lokale Register (L-Register) und n_O Register (O-Register) für Parameter eines Unterprogramms Y von X (mit $n = n_I + n_L + n_O$). Bei Aufruf von Y sind lediglich die O-Register von X als I-Register verfügbar. Dazu kommen neuen $n_L + n_O$ Register für Y. Sie werden so umnumeriert, daß Registernummern analog wie bei X sind. In der Regel ist die Zahl der Fenster beschränkt und sie wird über einen Zähler der Fensterstufe verwaltet. Bei Über- oder Unterlauf dieses Zählers wird eine Ausnahme ausgelöst.

Beispiel 6.29 (SPARC)
Dort gibt es je 8 globale Register (g0..g7), I-Register (i0..i7), L-Register (l0..l7) und O-Register (o0..o7). Die globalen Register sind für alle UPs zugreifbar. Bei einem Unterprogrammsprung werden die O-Register als I-Register verfügbar gemacht und 16 neue zur Verfügung gestellt. Maximal können 8 derartige Fenster benutzt werden.

Bei vielen Prozessoren ist n_I, n_L und n_O fixiert, manche Prozessoren erlauben, daß diese Werte durch Registerinhalte variabel definiert sind.

Anmerkung: Statusregister und Befehlszähler sind identisch zu CISC. Oft wird im SR der Fensterstufenzähler (s.o.) abgelegt.

Datenformate, Datentypen und -zugriff Analog zu CISC-Prozessoren können Bytes, Halbworte, Worte und Doppelworte geladen und gespeichert werden (von Speicher in Register und umgekehrt). Die Verarbeitung erfolgt jedoch immer in Wortbreite; es gibt also keine Operationen für andere Operandenbreiten.

Es ist nur ein Zugriff ohne Fehlausrichtung erlaubt. Die Bytewertigkeit eines RISC-Prozessors kann oft durch ein Steuerregister festgelegt werden.

Adressierungsarten Im Gegensatz zu CISC-Prozessoren haben RISC-Prozessoren wesentlich weniger Adressierungsarten, die von den Befehlen abhängen, in denen sie vorkommen. Im Modellprozessor seien folgende Abhängigkeiten gegeben:

Lade- und Speicherbefehle: In ihnen kommt genau eine Speicheradresse und eine Registeradresse vor. Die Speicheradresse kann registerindirekt mit 16-Bit-Offset errechnet werden (Adresse := r_i + Offset, Notation: "offset(r_i)") oder indiziert aus zwei Registerinhalten (Adresse := $r_i + r_j$, Notation: "$(r_i)(r_j)$").

verarbeitende Befehle: Hier kann der Befehl für die beiden Operanden nur auf den Registersatz zugreifen (Notation: "r_i") oder den Quelloperanden durch Adressierung als Konstante angeben (Notation: "operand"). Ziel ist immer ein Register.

Sprungbefehle: In ihnen ist nur eine befehlszählerrelative Adressierung mit einem Offset von 26 Bit möglich. Dieser Offset wird zuvor mit 4 multipliziert, da Befehle nur an durch 4 teilbaren Adressen beginnen

können. Im Assemblerbefehl steht nur das Sprungziel; der Assemblierer berechnet den Offset selbst.

Befehlsformate Aufgrund der stark eingeschränkten Zahl der Befehle und Adressierungsarten können RISC-Befehle des Modellprozessors sehr gleichförmig strukturiert sein.

Alle Befehle haben eine Länge von 32 Bit. Die ersten sechs Bit sind der Befehlskode. In den restlichen Bits ist jedes Register mit 5 Bits kodiert und Offsets können in übrigen Bits abgelegt werden.

Bei Lade- und Speicheroperationen verteilen sich die Informationen wie folgt (am Beispiel von "stb r18,456(r24)" (d.h. 'store byte aus r18 in HSP[r24+456]')):

$$\text{Bit}: \underbrace{32----27}_{\text{Befehlskode(stb)}}\underbrace{26---22}_{\text{Indexregister(24)}}\underbrace{21---------------6}_{\text{Offset(456)}}\quad\underbrace{5---1}_{\text{Quell}-/\text{Zielregister(18)}}$$

Bei verarbeitenden Befehlen wird die Dreiadreßform benutzt. Der zweite Operand kann auch eine Konstante sein, die in $32 - 6 - 3 \cdot 5 = 11$ Bits abgelegt wird (am Beispiel von "add r1,#1996,r8", d.h. r8:=r1+1996):

$$\text{Bit}: \underbrace{32----27}_{\text{Befehlskode(add)}}\underbrace{26---22}_{\text{Op.RegisterA(1)}}\underbrace{21---17}_{\text{Op.RegisterB(0)}}\underbrace{16----------6}_{\text{Operand(1996)}}\underbrace{5---1}_{\text{Zielregister(8)}}$$

Durch die gleichförmige Darstellung ist die Dekodierung sehr einfach.

Befehlssatz

RISC-Prozessoren haben nur wenige Befehle (oft weniger als 60). Für unseren Beispielprozessor werden tabellarisch die Befehle angegeben. Zur Kürze werden bei den Befehlsnamen geschweifte Klammern (für Varianten) und eckige Klammern (für optionale Teile) verwendet.

Die Auswirkung auf das Statusregister wird nur bei arithmetischen oder logischen Befehlen beschrieben, da es nur bei ihnen geändert wird.

Transportbefehle

LD$\begin{Bmatrix}Z\\S\end{Bmatrix}\begin{Bmatrix}B\\H\end{Bmatrix}$ mem, reg	load zero-/sign-extended byte/halfword; Speicherinhalt wird in Register transferiert und ergänzt
LD mem, reg	load; reg:=HSP[mem..mem+3]
ST$\begin{Bmatrix}B\\H\end{Bmatrix}$ reg,mem	store; Registerinhalt wird in Speicher transferiert (in entsprechender Länge)
SWAP mem,reg	swap; Speicherinhalt und Registerinhalt werden ununterbrechbar ausgetauscht;
SETU src,reg	set upper halfword; src ist Konstante;
READ sr,reg	read status register; REG:=SR;
WRITE reg,sr	write status register; SR:=REG; (privilegiert)

Der SWAP-Befehl dient als Synchronisationsbefehl bei Mehrprozessorbetrieb. Alle anderen Befehle sind klar.

Arithmetische und logische Befehle Als arithmetische Befehle gibt es nur Addition und Subtraktion; es wird unterschieden, ob die Statusbits verändert werden sollen oder nicht. Wenn keine Änderung stattfindet, dann können oft Operationen vertauscht werden.

ADD[C][CC] reg1,src2,reg3	add [with carry] [and modify condition codes]; Addition von reg1 und src2 (Register oder Konstante) unter eventueller Berücksichtigung von Carry und eventuellem Setzen der Statusbits; Ergebnis kommt nach reg3;
SUB[C][CC] reg1,src2,reg3	subtract. . . analog

Bei den logischen Befehlen gibt es auch die Möglichkeit, die alten Statusbits zu erhalten. Arithmetischer und logischer Linksshift werden nicht unterschieden, da Shifts die Statusbits nicht ändern.

$\left\{\begin{array}{c}\text{AND}\\ \text{OR}\\ \text{XOR}\\ \text{NXOR}\end{array}\right\}$[CC] reg1,src2,reg3	and/or/xor/ $\overline{xor}$ [and modify condition codes]; Operation zwischen reg1, src2 (Register oder Konstante) mit eventuellem Setzen der Statusbits; Ergebnis kommt nach reg3;
SLA reg1,src2,reg3	shift left arithmetic; Statusbits spielen keine Rolle
SRA, SRL	analog

Anmerkung: Wo sind die MOVE-Befehle zwischen Registern geblieben? Antwort: Sie werden durch "OR r0,regA,regB" ersetzt. Aufgrund der Reduktion der Befehle verstehen RISC-Assembler meistens sogenannte Pseudobefehle (wie z.B. "MOVE regA,regB"), die dann in einen äquivalenten RISC-Befehl übersetzt werden.

Sprungbefehle

Anmerkung: Bei Pipelining werden die verschiedenen Abarbeitungsstufen eines Befehls für mehrere aufeinanderfolgende Befehle fließbandartig ausgeführt. In der Regel sind das:

1. Befehl holen,
2. Operand holen,
3. Operation ausführen und
4. Resultat ablegen.

Bedingte Sprungbefehle sind problematisch, weil nicht klar ist, welcher Folgebefehl in die Pipeline eingespeist werden soll. Man kann dieses Problem auf verschiedene Arten lösen:

- Die Pipeline wird blockiert, bis klar ist welcher Befehl kommt (interlocking).
- Man versucht zu raten, welcher Befehl kommt und speist diesen vorsorglich in die Pipeline ein (branch prediction). Wurde falsch geraten, dann wird die Pipeline geleert. Das Raten wird oft unterstützt durch einen Assoziativspeicher der letzten Sprungziele (branch-target-buffer).

- Nach dem Sprungbefehl kommt ein Befehl, der in jedem Fall ausgeführt wird, egal ob gesprungen wird oder nicht. Man nennt das "Füllen des Delay-Slots". Dies kann oft durch Umordnung von Befehlen erreicht werden. Beim Modellprozessor gibt es diesen Delay-Slot.

Beispiel 6.30 (Delay-Slot)
Das linke Pseudo-RISC-Programm wird zum rechten RISC-Programm mit Ausnutzung des Delay-Slots (R3 sei mit 1 vorbelegt):

move R1,R2
LOOP:
dec R2
cmp R0,R2
bne LOOP
nop
move R2,R3
	OR R0,R1,R2
	SUB R2,R3,R2
	LOOP:
	SUBCC R0,R2,R0
	BNE LOOP
	SUB R2,R3,R2
	OR R0,R2,R3

Leider ist das erhaltene Programm nicht korrekt. Wenn der Rücksprung nicht stattfindet, wird trotzdem dekrementiert. Daher gibt es die Möglichkeit, bei einem Sprungbefehl zu sagen, daß der Folgebefehl annulliert wird, wenn der Sprung nicht stattfindet.

Es gibt folgende Sprungbefehle:

BA mem	branch always
BCC[.A] mem	branch conditional [and annull delay slot]; analog wie CISC, der Delay-Slot-Befehl wird nur ausgeführt, wenn Sprung stattfindet oder wenn Annullierung nicht gesetzt

Unterprogrammbefehle Befehle bezüglich Unterprogrammen sind analog zu CISC. Allerdings wird der Programmzählerinhalt nicht über einen Keller verwaltet. Sowohl Aufruf als auch Rückkehrsprung haben Delay-Slots.

CALL mem	call; r15:=PC; PC:=mem; Adressierung: befehlszählerrelativ
RTS mem	return from subroutine; PC:=mem; Adressierung: registerindirekt mit Offset oder registerindiziert;
SAVE	save registers; Fensterzähler:=Fensterzähler+1; wenn Überlauf, dann Fensterüberlaufausnahme;
RESTORE	restore registers; Fensterzähler:=Fensterzähler-1; wenn Unterlauf, dann Fensterunterlaufausnahme;

SAVE ist meist der erste Befehl in einem Unterprogramm, RESTORE meist der Befehl im Delay-Slot des Rückkehrsprungs.

Ausnahmebehandlung Die Ausnahmebehandlung ist wesentlich einfacher als bei CISC-Prozessoren. Zwei weitere Bits im Statusregister werden benutzt: Die Unterbrechungssperre und der vorhergehende Betriebsmodus. Eine Priorisierung von Unterbrechungen gibt es nicht, sondern sie werden zunächst pauschal gesperrt. Wenn ein TRAP während der Unterbrechungssperre auftritt, geht der Prozessor in einen Fehlerzustand (nur mit RESET aufhebbar). Ein Interrupt oder ein TRAP mit niedrigerer Priorität als der aktuelle wird in jedem Fall blockiert.

TA nummer	trap always; Unterbrechungssperre:=WAHR; vorhergehender_Modus:=Betriebsmodus; Fensterzähler++; (eventuell Ausnahme) r17:=PC; Betriebsmodus:=Systemmodus; PC:=Sprungtabelle[nummer]; Adressierung: registerindirekt mit Offset oder registerindiziert;
TCC nummer	trap conditional; analog wie TA bei Eintreten einer Bedingung (wie bei Sprungbefehlen)
RTE mem	return from exception; Unterbrechungssperre:=FALSCH; PC:=mem; Fensterzähler–; (eventuell Ausnahme) Betriebsmodus:=vorhergehender_Modus;

Der RTE-Befehl wird in Kombination mit einem RTS-Befehl ausgeführt (in dessen Delay-Slot). RTE kann dafür sorgen, daß der Befehl der den TRAP ausgelöst hat, wiederholt wird (Rückkehradresse bei RTE um 1 verringert) oder der darauf folgende Befehl. RTE hat ebenfalls einen Befehl in seinem Delay-Slot (nämlich den Befehl, auf den RTS springt).

6.2.3 Vergleich zwischen CISC und RISC bzgl. Programmierung

Folgende prinzipielle Unterschiede gibt es in der Programmierung von CISC- und RISC-Systemen:

- CISC haben einen kleinen Registersatz, RISC große mit Unterstützung von Unterprogrammen (Fenster).
- Bei CISC können Daten direkt im Speicher verarbeitet werden, bei RISC wegen stark eingeschränkter Adressierungsarten nicht (load/store-Architektur).
- Rekursion ist bei RISC von der Laufzeit aufwendiger als UP-Aufrufe mit geringer statischer Schachtelungstiefe.
- Ausnahmen werden bei CISC komfortabler behandelt.

Wir betrachten im folgenden einige RISC-spezifischen Probleme:

Load-Store: Aufgrund der Beschränkung auf Befehle in Worten lassen sich große Adreßoperanden nicht vollständig angeben. Lösung: statt in einem Befehl mit 2 Worten (CISC) erfolgt das Laden in zwei Befehlen (mit einem Wort).

CISC	RISC
MOVE #12345678,R2	SETU #1234,R2 OR R2,#5678,R2

Laden und Speichern eines Operanden erfolgen durch Transport der Adresse in ein Register und anschließenden LD- oder ST-Befehl also z.B.

CISC	RISC
MOVE 12345678,R2	SETU #1234,R5 OR R5,#5678,R5 LD (R5),R2

Ausnahmen Bei RISC wird beim Auftreten einer Ausnahme nur das Nötigste erledigt (s.o.). Zusätzlich muß ausprogrammiert werden:

1. Sichern des Statusregisters (auf Registerkeller);
2. Setzen der Priorität der Ausnahme im SR;
3. Lösen der Unterbrechungssperre;

Prinzipielles Problem: Ausnahme währen Ausnahmeanfangsbehandlung...
Am Ende der Ausnahmebehandlung muß das Statusregister restauriert werden (alle anderen Schritte müssen nicht rückgängig gemacht werden.)

6.2.4 Einfache CISC- und RISC-Programme

Ergänzung der Assemblersprachen um Pseudoanweisungen

Folgende Sprachmittel können sowohl bei CISC- als auch bei RISC-Programmen verwendet werden:

- Definition einer Startadresse eines Code- oder Datenabschnitts (origin):

  ```
  ORG Ausdruck
  ```

  bestimmt, daß nachfolgende Befehle oder Datendefinitionen ab Adresse `Ausdruck` abgelegt werden, z.B. bei `ORG 2FFE` werden alle folgenden Informationen ab Adresse 2FFE abgelegt.

- Definitionen einer Konstante ohne Speicherbedarf (equate):

  ```
  Symbol EQU Ausdruck
  ```

  definiert einen symbolischen Namen für den Ausdruck z.B. `x EQU 45`. An allen Stellen, an denen `Symbol` vorkommt, wird stattdessen der Ausdruck eingesetzt.

- Variablendeklarationen mit Vorbelegung des belegten Speichers (declare constant):

$$\texttt{[Symbol] DC.}\left\{\begin{array}{c}\texttt{B}\\\texttt{H}\\\texttt{W}\end{array}\right\}\texttt{ Ausdrucksliste}$$

definiert einen Variablennamen für eine Folge von Bytes, Halbworten oder Worten, die nacheinander im Speicher abgelegt werden. Symbol kann auch fehlen. Die Folge ist durch die Ausdrucksliste initialisiert, d.h. Ausdrücke, die durch Kommata getrennt sind. Das Symbol steht für die Adresse des ersten Elements der Liste.

- Variablendeklarationen ohne Vorbelegung des belegten Speichers (define storage):

$$\texttt{[Symbol] DS.}\left\{\begin{array}{c}\texttt{B}\\\texttt{H}\\\texttt{W}\end{array}\right\}\texttt{ Ausdruck}$$

definiert einen Variablennamen für eine Folge von Bytes, Halbworten oder Worten, die nacheinander im Speicher stehen. Ihre Zahl ist durch den Ausdruck festgelegt, ihr (anfänglicher) Inhalt ist 0. `Symbol` kann auch fehlen. Wenn es vorhanden ist, steht es für die Adresse des ersten Elements der Folge.

- Ausrichtung von Daten an Halbwort oder Wortgrenzen (alignment):

$$\texttt{ALIGN.}\left\{\begin{array}{c}\texttt{H}\\\texttt{W}\end{array}\right\}$$

fügt Bytes ein, sodaß der folgende Speicherplatz an einer durch 2 (bei H) oder 4 (bei W) teilbaren Adresse liegt.

- (bei RISC) Aliasnamen für Register (define register):

```
Symbol DEFREG Registername
```

definiert einen Aliasnamen für ein Register.

Anmerkung: Im folgenden werden — wenn nichts anderes angegeben ist — Zahlen als Hexadezimalzahlen interpretiert. Ausdrücke müssen immer einen zur Übersetzungszeit feststehenden Wert (eine Zahl oder eine Adresse) liefern.

Beispiel 6.31 (Pseudoanweisungen)
Beispielsweise werden durch

	ORG	3000
Hugo	DC.B	'H','e','i','n','i'
Otto	DC.W	1234,78990
	ALIGN.W	
xyz	DS.H	22

die Bytes 3000 bis 3005 durch den Text "Heini", die Bytes 3005 bis 3008 durch 00001234 und die Bytes 3009 bis 300C durch 00078990 belegt. Bytes 3010 bis 3053 sind mit 0 vorbelegt.

Fallbeispiel Bubble-Sort

Als Vergleich zwischen RISC und CISC realisieren wir den Bubble-Sort-Algorithmus zur Sortierung eines Feldes von Ganzzahlen in C, CISC-Assembler und RISC-Assembler. Es wird ein Feld definiert mit einer Vorbelegung und die Bubble-Sort-Prozedur aufgerufen.

Als Konvention für die Assembler nehmen wir an, daß Ganzzahlen in einem Halbwort und Adressen in einem Wort abgelegt werden.

Bubble-Sort in C

```
{
    int ein_feld[]={5,4,22,2,3,1};
    bubble(ein_feld, 6);
}

void bubble (int feld[], int laenge) {
    int i,j, temp;
    for (i=laenge-1; i>1; i–) {
        for (j=0; j<i-1; j++) {
            if (feld[j]>feld[j+1]) {
                temp=feld[j+1]; feld[j+1]=feld[j]; feld[j]=temp;
```

```
            }
        }
    }
}
```

Bubble-Sort in CISC-Assembler Die Parameterübergabe erfolgt hier auf dem Keller. Wir gehen dabei davon aus, daß die aufgerufene Prozedur die Parameter vom Keller nimmt und sie von rechts nach links abgelegt wurden.

```
         ORG       1000                    ; beliebige Anfangsadresse für aufrufende Prozedur
ein_feld DC.H      5,4,22,2,3,1            ; Vorbelegung von 12 Bytes
         PUSH.H    #6                      ; Länge als 2. aktueller Parameter
         PUSH.W    ein_feld                ; Adresse "ein_feld" als 1. aktueller Parameter
         CALL      bubble
         ...
         ORG       5000                    ; Anfangsadresse für bubble
feld     DS.W      1                       ; 1 Wort für Feldadresse
laenge   DS.H      1
i        DS.H      1
j        DS.H      1
temp     DS.H      1
bubble   POP.W     feld                    ; 1. Parameter holen
         POP.H     laenge                  ; 2. Parameter holen
         MOVE.H    laenge,R5
         DEC.H     R5
         MOVE.H    R5,i
fori     MOVE.H    i,R5
         CMP.H     R5,#0                   ; Prüfen der Schleifenbedingung
         BGT       endfori
         DEC.H     R5
         MOVE.H    #0,j
forj     CMP.H     j,R5
         BGE       endforj
         MOVZX.H   j,R1                    ; j wird in R1 abgelegt
         MOVE      #0,R0
         LEA       [feld(R0)](R1*2),R3     ; Adresse von feld[j] nach R3
         LEA       [feld+2(R0)](R1*2),R4   ; Adresse von feld[j+1] nach R4
         CMP.H     (R3),(R4)               ; if feld[j]>feld[j+1]...
         BLE       endif
         MOVE.H    (R3),temp               ; Vertauschung
```

```
         MOVE.H   (R4),(R3)
         MOVE.H   temp,(R4)
endif    INC.H    j
         JMP      forj
endforj  DEC.H    i
         JMP      fori
endfori  RTS                          ; Rückkehrsprung (Keller ist bereits o.k.)
```

Anmerkung: Folgende Eigenschaften gelten für das CISC-Programm:

- Die Parameter werden auf dem Keller übergeben.
- Komplexe Adressierungsarten können verwendet werden.
- Es wird oft direkt im Speicher gearbeitet.

Bubble-Sort in RISC-Assembler Die Parameterübergabe erfolgt durch Registerfensterumschaltung. Wir gehen dabei davon aus, daß die aufrufende Prozedur die Rückkehradresse in Register o0 (=Register r24) und die Parameter in o1 bis o7 übergeben kann (Register r25 bis r31). Sie seien von links nach rechts abgelegt.

Makroanweisungen werden in Großbuchstaben angegeben. Es gibt SETL, MOVE, INC, DEC, CLR, CMP mit unten beschriebenen Bedeutungen.

Das folgende Programm ist nicht direkt vergleichbar mit dem CISC-Programm, denn es nutzt aus, daß Variablen soweit wie möglich in Registern gehalten werden.

```
         org        123456               ; beliebige Anfangsadresse für aufrufe
         align.h                         ; Fehlausrichtung auf jeden Fall verm
ein_feld DC.H       5,4,22,2,3,1         ; Vorbelegung von 12 Bytes
         ; Adresse "ein_feld" in zwei Teilen übergeben
         setu       upper(ein_feld),o1   ; o1:=120000;
         SETL       lower(ein_feld),o1   ; or o1,lower(ein_feld),o1, d.h. o1:=o1
         MOVE       #6,o2                ; or r0,#6,o2, d.h. o2:=6
         call       bubble
         ...
         org        5000                 ; Anfangsadresse für bubble
; die Rückkehradresse steht in Register i0 = r8
feld     defreg     i1                   ; feld steht in Register i1 = r9
laenge   defreg     i2
```

```
i         defreg      l0             ; i steht in Register l0 = r16
j         defreg      l1
temp      defreg      l2
bubble    save                       ; Fensterumschaltung
          MOVE        laenge,l3
          DEC         l3             ; sub l3,#1,l3
fori      cmp         i,r0           ; Prüfen der Schleifenbedingung
          ble         endfori
          CLR         j              ; or r0,r0,l1 (im Delay-Slot!)
forj      subcc l0,#1,temp
          CMP         j,temp         ; subcc l1,l2,r0
          bge         endforj
; ADR(feld[j]) wird in l3, feld[j] in l4 abgelegt
          add         j,j,l3         ; im Delay-Slot
          add         l3,feld,l3
          ldsh        (l3),l4
; ADR(feld[j+1]) wird in l5, feld[j+1] in l6 abgelegt
          add         l3,#2,l5
          ldsh        (l5),l6
          CMP         l4,l6          ; if feld[j]>feld[j+1]...
          ble         endif
          NOP                        ; or r0,r0,r0 im Delay-Slot...
          sth         l6,(l3)        ; Vertauschung
          sth         l4,(l5)
endif     ba          forj
          INC         j              ; add l1,1,l1 im Delay-Slot
endforj   ba          fori
          DEC         i              ; Dekrementierung im Delay-Slot
endfori   rts (i0)                   ; Rückkehrsprung
          restore
```

Anmerkung: Folgende Eigenschaften gelten für das RISC-Programm:

- Es gibt sehr eingeschränkte Adressierungsarten.
- Es wird fast nur in Registern gearbeitet.
- Die Parameter werden im Registersatz übergeben.
- Die Delay-Slots sollten mit sinnvollen Befehlen gefüllt werden. Insbesondere ist darauf zu achten, daß die Ausführung des Befehls o.k. ist, auch wenn der Sprung stattfindet.

6.3 Prozessoren für höhere Programmiersprachen

Anmerkung: Herkömmliche Prozessoren sind in ihrer Architektur stark am von-Neumann-Modell orientiert. Eine Übersetzung von höheren Programmiersprachen in dieses Modell ist kompliziert und der erzeugte Code oft nicht sehr performant.
$\rightsquigarrow$Konzeption einer kellerbasierten Maschine mit einfachen Befehlen auf Kellerinhalt
$\rightsquigarrow$Kompilation für eine *virtuelle Maschine*
Gängige virtuelle Maschinen:

UCSD-p-Code: Zwischenkode des PASCAL-Compilers der ETH (1975); Implementierung von p-Code-Interpretern auf diversen Plattformen durch UCSD

Smalltalk 80: virtuelle Maschine für Smalltalk

Lilith: Hardwareimplementierung einer virtuellen Maschine für M-Code für MODULA-2 [6]

Java-Bytekode: Objektcode für JAVA [3]

Anmerkung: Eigenschaften einer virtuellen Maschine:

Vorteile:

- extrem kompakter Objektkode
- einfachere Übersetzung, da geringere semantische Lücke zwischen Objektkode und Quellprogramm
- portabel; lediglich Kodegenerierung oder Interpretierung auf neuer Plattform ("write once, run everywhere")
- Ausführung kann kontrolliert erfolgen, d.h. unsichere Operationen werden unterbunden

Nachteil: meist erheblich schlechtere Ablaufgeschwindigkeit (auch bei semikompilierenden System ("just in time-Compiler")) auf realen Maschinen

⇝Implementation der virtuellen Maschine in Hardware, d.h. Entwicklung eines Prozessors, der keine reine von-Neumann-Maschine mehr ist

⇝Für die Vorlesung erfolgt die Demonstration am Beispiel der Java-VM und der Umsetzung im Pico-Java-Kern [Sun1996].

6.3.1 Abriß der Programmiersprache JAVA

Anmerkung: Java ist eine OO-Programmiersprache mit folgenden Eigenschaften:

- vereinfachte OO-Sprache auf Basis von C++, Notation ist ähnlich zu C/C++
- stark objektorientiert, d.h. keine freien Funktionen und globalen Variablen; alle Information ist in Klassen organisiert
- statische Typisierung
- Verzicht auf Zeiger/Adressen; alle Objekte sind als Referenzen realisiert (keine Strukturtypen)
- explizite Schnittstellen
- fest definierte Größe von Basistypen (s.u.)
- Speichermanagement erfolgt nicht durch Programmierer, sondern durch automatische Speicherbereinigung
- etwas klareres Schnittstellen- und Modulkonzept als in C/C++
- Sprachmittel für Nebenläufigkeit in der Sprache
- keine Mehrfachvererbung; Implementierung mehrerer Schnittstellen möglich (vermeidet "Diamanten des Todes")
- systemnahe Programmierung nicht möglich
- Internet-"Darling"; als strategische Sprache für Internet-Anwendungen propagiert (hohe Umgebungssicherheit)

Beispiel 6.32 (Beispiel eines JAVA-Programms)

```
public class B_KELLER {
    public B_KELLER(int m_size) {
        size=0; maxsize=m_size; info=new Element_Typ[m_size];
```

```
    }
    // kein Destruktor!
    public boolean is_empty() { return size==0; };
    public void push (Object e) {
        if (size<maxsize) { info[size++]=e; }
    }
    private Element_Typ info[];
    private int size, maxsize;
}
```

6.3.2 Die virtuelle Maschine für Java (JVM)

Datentypen, Zugriffsarten und Bytewertigkeit

Erklärung 6.33 (Datentypen)
Die JVM kennt Referenztypen (für Objekte und Feldtypen) und einfache Datentypen:

Datentyp	Interpretation	Datenformat
byte	-2^7 bis $2^7 - 1$	8 Bit
short	-2^{15} bis $2^{15} - 1$	16 Bit
int	-2^{31} bis $2^{31} - 1$	32 Bit
long	-2^{63} bis $2^{63} - 1$	64 Bit
char	vorzeichenlose Zeichen des Unicodes	16 Bit
float, double	Fließpunktzahl im IEEE754-Format mit Mantisse und Exponent	32, 64 Bit
reference	Adresse	32 oder 64 Bit

Anmerkung: Der Zugriff in der JVM auf Daten erfolgt wortorientiert, wobei ein Wort eine implementierungsabhängige Größe hat. In jedem Fall muß ein *Wort* ein Exemplar vom Typ byte, char, short, int, float und reference aufnehmen können, zwei Worte einen Wert vom Typ long oder double.
Die JVM ist big-endian.

Registersatz

Erklärung 6.34 (Registersatz der JVM)
Die JVM besitzt als einziges Register einen Befehlszähler. Die Berechnungen werden auf einem Keller durchgeführt.

Erklärung 6.35 (Keller und Kellerrahmen)
Rahmen sind Bereiche auf dem Keller, die einer Methode zugeordnet sind und zur Speicherung von Daten und Ergebnissen dienen. Rahmen erhalten Platz für die zur Methode gehörenden

- Parameter und lokale Variablen
- Rahmenstatusinformation (z.B. Rücksprungadresse oder Typkennung als Referenz auf einen Deskriptor) und
- Operanden von Ausdrücken.

Die Informationen sind immer ein oder zwei Worte lang (je nach Typ der Information).

Befehlssatz

Erklärung 6.36 (Befehlssatz der JVM)
Der Befehlssatz der virtuellen Javamaschine hat folgende Eigenschaften:

- Ein Befehl der JVM besteht aus einem Byte langen Operationskode (d.h. es gibt 256 Befehle!), der von null oder mehr Operandenbytes gefolgt werden kann. Die Anzahl der Operandenbytes ist fast immer aus dem Operationskode erkennbar. Es gibt keine Ausrichtung der Befehle an Wortgrenzen (mit zwei Ausnahmen).
- Die Befehle wirken oft auf die oberen Elemente des Operandenkeller ein. Sie sind explizit oder implizit typisiert.
  Typkürzel: "b" für Byte-, "c" für Char-, "s" für Short-, "i" für Int-, "f" für Fließkomma-, "d" für Doppelfließkomma- und "a" für Referenz(Adreß)-Operation.
- Der Befehlssatz ist nicht orthogonal (aufgrund der geringen Zahl von Operationskodes im Vergleich zur Zahl der Typen).

- Byte, Char und Short werden bei Berechnungen meist wie Int behandelt. Beim Schreiben in den Keller oder in lokale Variablen wird eine Vorzeichenerweiterung bzw. eine Füllung mit 0 vorgenommen (bei Char!).
- Große und ungewöhnliche Konstanten werden nicht als Operand geholt, sondern aus einem speziellen Speicherbereich, dem Konstantenpool. Für simple Konstanten (0,1,2) gibt es implizite Befehle.

Lade- und Speicherbefehle Es gibt folgende Transportbefehle zum Laden und Speichern von Werten zwischen lokalen Variablen und Operandenkeller:

- Transport von lokaler Variable auf Operandenkeller:

Befehl	Name	Semantik
ILOAD index	integer load	push(local_word[index]);
ILOAD_<index>	integer load with implicit index	push(local_word[index]);
LLOAD index	long load	push(local_word[index]); push(local_word[index+1]);
LLOAD_<index>	long load with implicit index	push(local_word[index]); push(local_word[index+1]);

analog FLOAD, FLOAD_<index>, DLOAD, DLOAD_<index>, ALOAD und ALOAD_<index>

- Transport von Operandenkeller in lokale Variable:

Befehl	Name	Semantik
ISTORE index	integer store	local_word[index]:=pop();
ISTORE_<index>	integer store with implicit index	local_word[index]:=pop();
LSTORE index	long store	local_word[index+1]:=pop(); local_word[index]:=pop();
LSTORE_<index>	long store with implicit index	local_word[index+1]:=pop(); local_word[index]:=pop();

analog FSTORE, FSTORE_<index>, DSTORE, DSTORE_<index>, ASTORE und ASTORE_<index>

- Transport einer Konstante auf Operandenkeller:

Befehl	Name	Semantik
BIPUSH b1	byte int push	push(b1);
SIPUSH b1 b2	short int push	push(b1*256+b2);
LDC num	load constant from pool	push(pool_constant[num]);
χCONST_<index>	push implicit constant ($\chi \in \{i, l, f, d\}$)	push(constant);

analog ACONST_NULL, LDC_W, LDC2_W

Arithmetische Befehle Es gibt folgende arithmetische Befehle, die jeweils auf dem Operandenkeller operieren und das Ergebnis dort wieder ablegen:

- arithmetische Befehle:

Befehl	Name	Semantik
χADD	add ($\chi \in \{i, l, f, d\}$)	bei i und f: t1:=pop(); t2:=pop(); push(t1+t2)
χSUB	subtract	analog
χMUL	multiply	analog
χDIV	divide	analog
χREM	remainder	analog
χNEG	negate ($\chi \in \{i, l, f, d\}$)	bei i und f: push(-pop());

- logische Befehle ($\chi \in \{i, l\}$):

Befehl	Name	Semantik
χSHL	shift left	der zweite Wert wird modulo 32 genommen
χSHR	arithmetic shift right	analog
χUSHR	logical shift right	analog
χOR	bitwise or	analog
χAND	bitwise and	analog
χXOR	bitwise and	analog

- Inkrementbefehl:

Befehl	Name	Semantik
IINC index val	increment integer	local_word[index]:=local_word[index]+val;

Anmerkung: Es gibt keinen Über- oder Unterlauf. Eine Ausnahme wird nur bei Division durch 0 ausgelöst. Bei Fließpunktzahlen werden die Konventionen der IEEE754 bezüglich Bereichsüberschreitungen und Operanden angewandt.

Typtransferbefehle Sie wandeln den Typ des "oberen Eintrags" des Operandenkellers in einen anderen um:

- Typerweiterungen: Die Bitbreite des Operandentypen ist kleiner als oder gleich wie die des Zieltypen.

Befehl	Name	Semantik
I2L, I2F, I2D	integer to long usw.	Vorzeichenerweiterung bzw. Einbettung
L2F, L2D, F2D	analog	

Bei I2F, L2F und L2D tritt eventuell ein Verlust an gültigen Ziffern ein!

- Typverengungen: Die Bitbreite des Operandentypen ist größer als die des Zieltypen.

Befehl	Name	Semantik
I2B, I2S, I2C	integer to byte, short, char...	Abschneiden auf entsprechende Länge
L2I	analog	
F2I, F2L		Konversion; Sonderwerte werden plausibel abgebildet (z.B. $\infty \rightarrow MaxInt$)
D2I, D2L, D2F	analog	

Vorsicht: Reines Abschneiden kann Vorzeichen und/oder Wert verfälschen! Es gibt aber keine Laufzeitausnahmen!

Befehle auf Objekten und Feldern Es werden unterschiedliche Befehle für Objekte und Feldtypen verwendet:

- Erzeugung: die resultierende Referenz wird auf Keller abgelegt

Befehl	Name	Semantik
NEW b1 b2	new instance	erzeugen einer leeren Hülle für Exemplar von Klasse mit Index b1*256+b2
NEWARRAY typkode		länge:=pop(); typkode bestimmt Elementtyp
ANEWARRAY b1 b2		länge:=pop(); Elementtyp ist Klasse mit Index b1*256+b2

  analog: "MULTIANEWARRAY b1 b2 dim" für mehrdimensionale Felder

- Zugriff auf Exemplar- oder Klassenattribute (Exemplarreferenz auf Operandenkeller)

Befehl	Semantik
GETFIELD b1 b2	stelle Wert der b1*256+b2-te Komponente des Exemplars auf Operandenkeller
PUTFIELD b1 b2	setze Komponente des Exemplars auf obersten Kellerwert
GETSTATIC	analog für Klassenattribute
PUTSTATIC	analog für Klassenattribute

- Feldzugriff: Operandenkeller enthält Feldreferenz, Index und eventuell Wert (beim Speichern)

Befehl	Name	Semantik
χALOAD	array element load ($\chi \in \{b, c, s, i, l, f, d, a\}$)	Laden von Wert mit vorgegebenen Typ auf Operandenkeller
χASTORE	array element store ($\chi \in \{b, c, s, i, l, f, d, a\}$)	Speichern von Wert mit vorgegebenen Typ in Feld

- Sonstiges: Objekt- oder Feldreferenz ist am Anfang auf Operandenkeller

Befehl	Semantik
ARRAYLENGTH	Länge des referenzierten Feldes
INSTANCEOF b1 b2	prüfe, ob Referenz eine Exemplar von Klasse b1*256+b2 ist
CHECKCAST	analog zu INSTANCEOF; legt aber keinen Wahrheitswert auf Keller

Anmerkung: Warum eine derartig feine Unterscheidung von Typen in Objektkode? Idee dabei ist, daß der Java-Interpreter nicht sicher sein kann, daß der Java-Code vertrauenswürdig ist! Prinzipiell kann zur Verifikation beim Laden einer Klasse eine Datenflußanalyse gemacht werden, die beispielsweise prüft, ob die Operandentypen stets passen. $\Longrightarrow$Java-Typen müssen auch in der VM vorkommen!

Kellerbefehle

- Entfernen: POP
- Duplikation:

Befehl	Semantik
DUP	Duplikation des obersten Wortes
DUP_X1	Duplikation des obersten Wortes und dabei ein Wort überspringen
DUP_X2	Duplikation des obersten Wortes und dabei zwei Worte überspringen

- Vertauschen: SWAP

POP2, DUP2, DUP2_X1, DUP2_X2 sind die analogen Befehle für Doppelworte (allerdings werden auch nur *Worte* übersprungen). Es gibt kein SWAP2.

Kontrollbefehle Sie dienen zur Steuerung des Kontrollflusses; Sprungziel sind als Offset in beiden Folgebytes gegeben.

- bedingte Verzweigung: starke Betonung auf int;

Befehl	Semantik
IF_ICMPχb1 b2	($\chi \in \{EQ, NE, LT, LE, GT, GE\}$) Vergleich der oberen Ganzzahloperanden (mit Vorzeichen)
IFχb1 b2	($\chi \in \{EQ, NE, LT, LE, GT, GE\}$) Vergleich des obersten Ganzzahloperanden (mit Vorzeichen) mit 0
LCMP	Vergleich zweier Long-Werte; liefert -1,0,+1 als Resultat
FCMPG,FCMPL	analog für float; liefert -1,0,+1 als Resultat (Unterschied zwischen beiden in der Behandlung von NaN)
DCMPG,DCMPL	analog für double
IF_ACMPEQ	Vergleich der oberen Referenzoperanden (analog IF_ACMPNE)
IFNULL	Referenzvergleich mit NULL (analog IFNONNULL)

- unbedingte Verzweigung:

Befehl	Semantik
GOTO b1 b2	PC:=PC+(b1*256+b2);
GOTO_W	mit 32Bit-Offset in vier Folgebytes
JSR b1 b2	push(PC); PC:=PC+(b1*256+b2);
JSR_W	mit 32Bit-Offset in vier Folgebytes
RET index	PC:=local_word[index]; (!!!)

- Sprungtabellen:

Befehl	Semantik
TABLESWITCH	ganzzahlige CASE-Anweisung; Befehlskode wird gefolgt von 0 bis 3 Füllbytes (Ausrichtung auf Wortgrenze), 32Bit-Standardoffset d, Untergrenze l, Obergrenze h und $h - l + 1$ 32Bit-Offsets; oberster Operand gibt Wert für Fallunterscheidung vor
LOOKUPSWITCH	ganzzahliger Suchbefehl; Befehlskode wird gefolgt von 0 bis 3 Füllbytes (Ausrichtung auf Wortgrenze), 32Bit-Standardoffset d, Zahl der Vergleichspaare n, und n Paare mit 32Bit-Schlüssel und 32Bit-Offsets (Schlüssel aufsteigend sortiert); oberster Operand gibt Wert für Suche vor

Methodenbefehle Argumente für Methodenaufrufe liegen auf dem Operandenkeller. Bei Exemplarmethoden ist die Objektreferenz oberster Operand.

Befehl	Semantik
INVOKEVIRTUAL index	Aufruf der Exemplarmethode mit Nummer index; Anlage eines neuen Kellerrahmens mit Referenz und Parametern als obersten Einträgen; PC:=Startadresse der Methode
INVOKESPECIAL	analog für Konstruktoren
INVOKESTATIC	analog für Klassenmethoden
INVOKEINTERFACE	analog für Schnittstellenmethoden
χRETURN	($\chi \in \{I, L, F, D, A\}$) Aufräumen des Kellerrahmens, Rückkehrsprung und Ablegen des Resultats in Operandenkeller des Aufrufers
RETURN	analog bei keinem Resultattyp

Sonderbefehle Hier werden Befehle zusammengefaßt, die nicht in anderen Kategorien abgedeckt werden konnten:

Befehl	Semantik
ATHROW	Ausnahme auslösen; oberster Kellereintrag ist Ausnahmeobjekt
MONITORENTER	zur Synchronisation von nebenläufigen Prozessen; oberster Kellereintrag ist "bewachtes" Objekt
MONITOREXIT	zur Synchronisation von nebenläufigen Prozessen; oberster Kellereintrag ist "bewachtes" Objekt und wird freigegeben

Codebeispiele

Beispiel 6.37 (Java-Programm mit ganzen Zahlen und Schleife)

```
void xyz () {
    int i;
    for (i=0; i<100; i++) {
        ;
    }
}
```

Java-Bytecode:

Methode void xyz ()

```
 0  iconst_0      0 kellern
 1  istore_1      ablegen in lokales Wort #1 (i)
 2  goto 8        zum Schleifenende
 5  iinc 1 1      inkrementiere lokales Wort #1 um 1
 8  iload_1       i kellern
 9  bipush 100    Bytekonstante 100 kellern
11  if_icmplt 5   wenn i<100 dann wiederhole
14  return        fertig
```

Erläuterung: Die Java-VM ist eine Kellermaschine. Bei jedem Funktionsaufruf wird ein eigener Kellerrahmen angelegt. Variablen werden auf dem Keller beginnend mit 1 indiziert (sie belegen ein oder zwei Worte).
Auffällig ist, daß die Bytecodes typisiert sind und daß spezielle Befehle für spezielle Operanden existieren (iconst_...).

Beispiel 6.38 (Java-Programm mit langen Fließkommazahlen und Schleife)

```
void double_xyz () {
    double i;
    for (i=0.0; i<100.0; i++) {
        ;
    }
}
```

Java-Bytecode:

```
Methode void double_xyz ()
 0  dconst_0     0.0 kellern
 1  dstore_1     ablegen in lokales Wort #1 und #2 (i)
 2  goto 9       zum Schleifenende
 5  dload_1      i kellern
 6  dconst_1     1.0 kellern
 7  dadd         i+1.0
 8  dstore_1     i:=i+1.0
 9  dload_1      i kellern
10  ldc2_w #4    Konstante 100.0 aus Konstantenpool holen und kellern
13  dcmpg        Vergleich
14  iflt 5       wenn i<100.0 dann wiederhole
17  return       fertig
```

Erläuterung: i wird in zwei Worten abgelegt. Für Laden und Speichern gibt es aber trotzdem Befehle. Diverse Operationen müssen jedoch — im Gegensatz zu Operationen auf Ganzzahlen — kompliziert ausprogrammiert werden (Inkrementierung, Vergleich). Komplizierte Konstanten werden aus dem Konstantenpool geholt.

Beispiel 6.39 (Java-Programm mit Parametern)

```
int addieren (int i, int j) {
    return i+j;
}
```

Java-Bytecode:

Methode int addieren (int,int)		
0	iload_1	lokales Wort #1 kellern (i)
1	iload_2	lokales Wort #2 kellern (j)
2	iadd	addieren
3	ireturn	fertig, Resultat steht auf Keller

Erläuterung: Parameter werden analog wie lokale Variablen im Kellerrahmen abgelegt. Der oberste Eintrag im Kellerrahmen ist der Zeiger auf das aktuelle Objekt der Methode (this), wenn es keine Klassenmethode ist.

Beispiel 6.40 (Java-Programm mit Methodenaufruf)

```
int add12_13 () {
    return addieren(12, 13);
}
```

Java-Bytecode:

Methode int add12_13 ()		
0	aload_0	lokales Wort #0 kellern (this)
1	bipush 12	Konstante 12 kellern
3	bipush 13	Konstante 13 kellern
5	invokevirtual #22	Eintrag im Konstantenpool für “addieren” aufrufen
3	ireturn	fertig, Resultat steht auf Keller

Erläuterung: Die Übergabe der Parameterablage ist klar (siehe oben). Wenn neuer Rahmen erstellt wird, werden für “addieren” die drei obersten Einträge

übernommen. #22 ist kein Offset in irgendeine Methodenzeigerliste, sondern ein Index in dem Konstantenpool, d.h. eine symbolische Referenz, die entweder beim Laden der Klasse oder beim erstmaligen Benutzen der Methode aufgelöst wird.

Beispiel 6.41 (Java-Programm mit Objekten)

```
public class Testklasse {
    public int i;
    public void aaa (int wert) {
        i = wert;
        Testklasse x = new Testklasse();
    }
}
```

Java-Bytecode:

Methode int aaa ()

0	aload_0	lokales Wort #0 kellern (this)
1	iload_1	lokales Wort #1 kellern (wert)
2	putfield #4	Feld i mit wert belegen
5	new #2	Hülle (und Zeiger) erzeugen für Exemplar von Testklasse
8	dup	Zeiger als Parameter übergeben an Initialisierung
9	invokespecial #5	Methode Testklasse.< *init* > aufrufen
12	astore_2	in lokalem Wort #2 speichern (x)
13	return	

Erläuterung: Der Zugriff auf Attribute eines Objekts erfolgt über Aufruf von getfield oder putfield, die mit einer symbolischen Referenz auf das Attribut arbeiten. Beim Erzeugen von Objekten wird erst der Speicher reserviert und danach der Konstruktor aufgerufen.

Beispiel 6.42 (Java-Programm mit Feldern)

```
public void felder () {
    int buffer[] = new int[200];
    buffer[10] = buffer[12];
    Testklasse b[] = new Testklasse[42];
    b[0] = new Testklasse();
}
```

Java-Bytecode:

```
Methode void felder ()
 0  bipush 200          Länge des Felds kellern
 2  newarray int        neues Int-Feld anlegen
 4  astore_1            Zeiger in buffer ablegen
 5  aload_1             buffer kellern
 6  bipush 10           10 kellern
 8  aload_1             buffer kellern
 9  bipush 12           12 kellern
11  iaload              buffer[12] holen
12  iastore             in buffer[10] ablegen
13  bipush 42           Länge des Felds kellern
15  anewarray #4        Feld von Zeigern für Typ "Testklasse" anlegen
18  astore_2            Zeiger in b ablegen
19  aload_1             buffer kellern
20  iconst_0            0 kellern
21  new #4              Hülle (und Zeiger) erzeugen für Exemplar von Testklasse
24  dup                 Zeiger als Parameter übergeben an Initialisierung
25  invokespecial #5    Methode Testklasse.< init > aufrufen
28  aastore             in b[0] speichern
29  return
```

Erläuterung: Es wird unterschieden zwischen Feldern von einfachen Datentypen und Feldern von Objekten. Indizierung ist eine eigene Operation, da die Java-VM keine Adreßarithmetik zuläßt.

Beispiel 6.43 (Java-Programm mit Sprungtabellen)

```
int einfacherSwitch (int i) {
    switch(i) {
        case 0: return 2;
        case 1: return 1;
        case 2: return 0;
        default: return -1;
    }
}
```

Java-Bytecode:

```
Methode int aaa ()
 0  iload_1             lokales Wort #1 kellern (i)
```

1	tableswitch 0 to 2:	gültige Werte sind 0 bis 2
	28	für 0 gehe zu Befehl 28
	30	für 1 gehe zu Befehl 30
	32	für 2 gehe zu Befehl 32
	34	sonst gehe zu Befehl 34
28	iconst_2	i war 0
29	ireturn	⟹return 2
30	iconst_1	i war 1
31	ireturn	⟹return 1
32	iconst_1	i war 2
33	ireturn	⟹return 0
32	iconst_m1	i war außerhalb
35	ireturn	⟹return -1

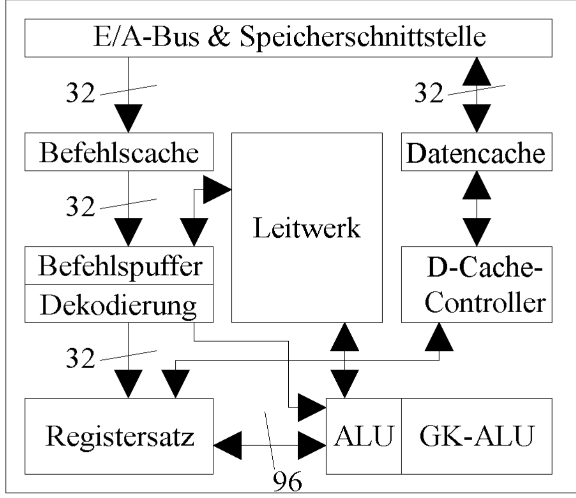
Erläuterung: tableswitch erzeugt vollständige Sprungtabelle für alle Werte im Intervall 0 bis 2 (jeweils 4 Byte Sprungoffset pro Wert). Zusätzlich enthält der Befehl die Unter- und Obergrenze in je 4 Bytes und den Sprungoffset, falls der Wert außerhalb des Intervalls liegt. Da zusätzlich eine Ausrichtung der Sprungtabelle auf eine 32Bit-Grenze stattfindet, ergibt sich die angegebene Startadresse für die Fälle.
Für nicht dicht besetzte Sprungtabellen ist der assoziative lookupswitch besser geeignet.

6.3.3 Der picoJava-Prozessor

Anmerkung: Der picoJava-Prozessor besitzt eine RISC-ähnliche Struktur optimiert für den Code der virtuellen Maschine für Java mit folgenden Eigenschaften:

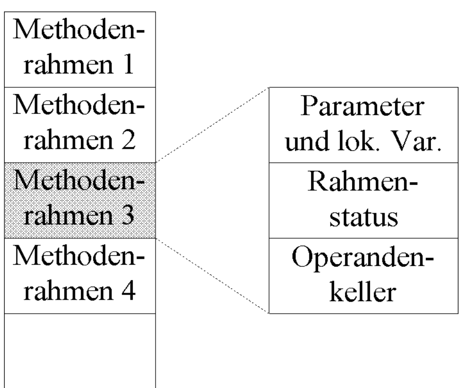
- Load-/Store-Architektur
- großer Registersatz als Cache auf Kellerrahmen
- Ausführung der meisten Instruktionen in einem Taktschritt
- Operationskode in einem Byte kodiert; variable Länge der Befehle
- Mechanismen zur Ausnahmebehandlung;
- Fließbandbearbeitung von Befehlen (Pipelining);
- interner Daten- und Befehlspuffer (Cache);

Prozessorkern



Erklärung 6.44 (Prinzipieller Aufbau des Prozessorkerns)
In der Regel sind Befehle festverdrahtet. Komplexere Befehle erfordern Ablaufsteuerung über Mikroprogramm. Bei einigen extrem komplexen Befehlen (NEW) wird eine Ausnahme ausgelöst und eine externe Folge von Befehlen ausgeführt (Emulation).
Zwischen Bus bzw. optionalem Befehlscache und Dekodierung liegt ein 12-Byte großer Instruktionspuffer mit 32 Bit breitem Datenpfad$\Longrightarrow$Lesen von ≈ 2 Befehlen parallel zur Dekodierung möglich.

Kellercache

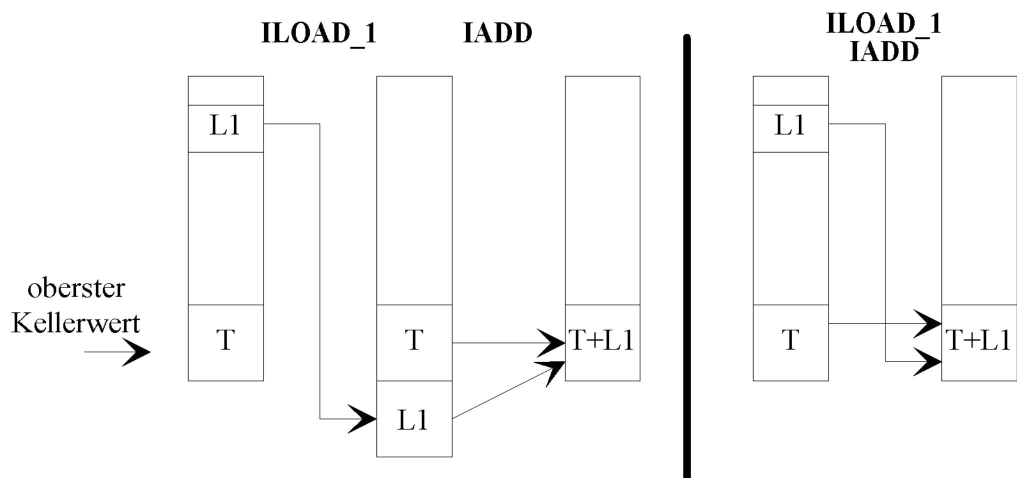


Erklärung 6.45 (Registersatz)
Der Registersatz besteht aus 64 Registern (mit 32Bit Breite). Diese Register sind als Ringpuffer (!) angeordnet. Ein Kellerrahmen belegt aufeinanderfol-

gende Register und aktuelle Parameter auf dem Keller werden zu lokalen Variablen eines neuen Kellerrahmens (wie Registerfenster bei RISC).

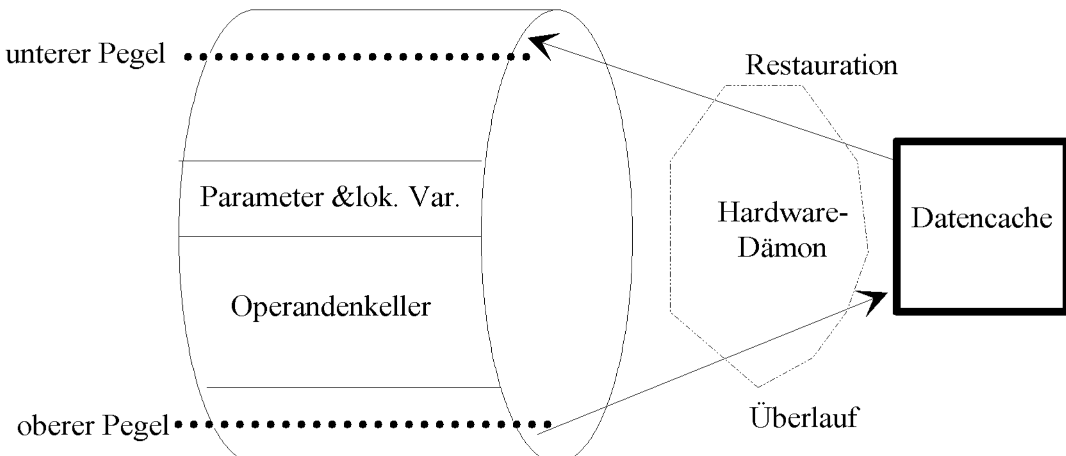
Anmerkung: Bei einer derartigen Architektur gibt es zwei offensichtliche Probleme:

1. Bei kellerartigen Systemen gibt es inhärente Leistungseinbußen durch nicht wahlfreien Zugriff im Keller. Daher wird ein Trick zur Leistungsverbesserung angewandt: man faßt einen Transport- und einen arithmetischen Befehl in einen Schritt zusammen. Der Prozessor prüft, ob im Instruktionspuffer solche Befehle aufeinanderfolgen und ob die referenzierte Variable zugänglich ist.

   In obigem Bild ist der Ladebefehl einzusparen, wenn bei der Addition direkt auf die lokale Variable zugegriffen wird.

2. Unklar ist weiterhin das Verhalten des Systems, wenn der Keller voll ist: Der Registersatz ist ein Dual-Port-Speicher. Damit kann eine parallel laufende Einheit die altesten Einträge vom Keller in den Datencache des Prozessors oder Hauptspeicher zu kopieren, um Platz für neue Kellerrahmen zu schaffen. Wenn die Inkarnation beendet ist, werden diese Einträge wieder zurückkopiert. Zur Triggerung diese Mechanismus werden programmierbare “Hoch-/Niedrigwassermarken” verwendet.


unterer Pegel
Restauration
Parameter &lok. Var.
Hardware-
Dämon
Datencache
Operandenkeller
oberer Pegel
Überlauf

Literaturverzeichnis

[1] Karl Heinz Fasiol: Binäre Steuerungstechnik.
Springer-Verlag, Heidelberg

[2] Thomas Flik, Hans Liebig: Mikroprozessortechnik.
Springer-Verlag, Heidelberg, 1998

[3] James Gosling: Die JAVA-Virtuelle Maschine.
Addison Wesley, 1996

[4] Manfred Seifart: Digitale Schaltungen.
Verlag Technik, Berlin, 1998

[5] Klaus Urbanski, Roland Woitowitz: Digitaltechnik.
Springer-Verlag, Heidelberg, 2000

[6] Niklaus Wirth: The Personal Computer Lilith.
ETH Zürich, Institut für Informatik, Technischer Bericht 40, April 1981