Grundlagen der Technischen Informatik 2

Prof. Dr. Martin Middendorf
Parallelverarbeitung und Komplexe Systeme
middendorf@informatik.uni-leipzig.de

Übersicht

- Einleitung
- Schaltnetze
  - KV-Diagramme
  - Minimierung nach Quine-McCluskey
- Speicherglieder
  - RS-Flipflop
  - D-Flipflop
  - JK-Flipflop
  - T-Flipflop

Übersicht

- Schaltwerke
  - Darstellung endlicher Automaten
  - Minimierung der Zustandszahl
  - Einfluss der Zustandskodierung
- Spezielle Schaltnetze und Schaltwerke
  - Multiplexer, Demultiplexer, Addierer
  - Register, Schieberegister, Zähler
- Ein einfacher Beispielrechner
  - Befehlssatz
  - Realisierung
  - Arbeitsweise und Programmierung
- Aufbau von Rechnersystemen
  - Komponenten eines Rechnersystems
  - Prinzipieller Aufbau eines Mikroprozessors
  - Steuerwerk und Mikroprogrammierung
  - Rechenwerk
  - Das Adresswerk

Übersicht

- Rechner- und Gerätebusse
  - interne Busse
  - externe Busse
- E/A-Steuerungen
  - Prinzip der Datenein-und -ausgabe
  - Parallele Schnittstellen
  - Serielle Schnittstellen
  - Analoge Ein- und Ausgabe
- Peripheriegeräte
  - Tastatur
  - Graphikadapter
  - Festplatten- und Diskettenlaufwerke
  - Sonstige E/A-Geräte

Literatur

Die Vorlesung basiert u.a. auf den Lehrbüchern:

- **W. Schiffmann, R. Schmitz: „Technische Informatik 1 Grundlagen der digitalen Elektronik“ Springer-Verlag (1992)**
- **W. Schiffmann, R. Schmitz: „Technische Informatik 2 Grundlagen der Computertechnik“ Springer-Verlag (1992)**
- **H. Bähring: „Mikrorechnersysteme“ Springer-Verlag (1994)**

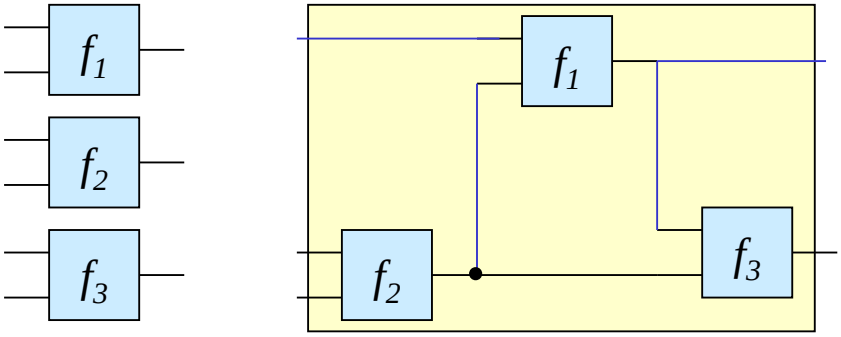
1 Einleitung

Der Entwurf elektronischer Systeme ist gekennzeichnet durch:

- **Zunahme der Komplexität und Integrationsdichte**
- **höhere Packungsdichten aufgrund geringerer Strukturgrößen**
- **steigende Anforderungen (Platzbedarf, Taktrate, Leistungsverbrauch, Zuverlässigkeit)**
- **kurze Entwicklungszeiten (time to market)**
- **Wiederverwendung von Entwurfsdaten (Re-use)**

☞ **Die Entwicklung elektronischer Systeme ist bei der heutigen Komplexität nur durch eine strukturierte Vorgehensweise beherrschbar!**

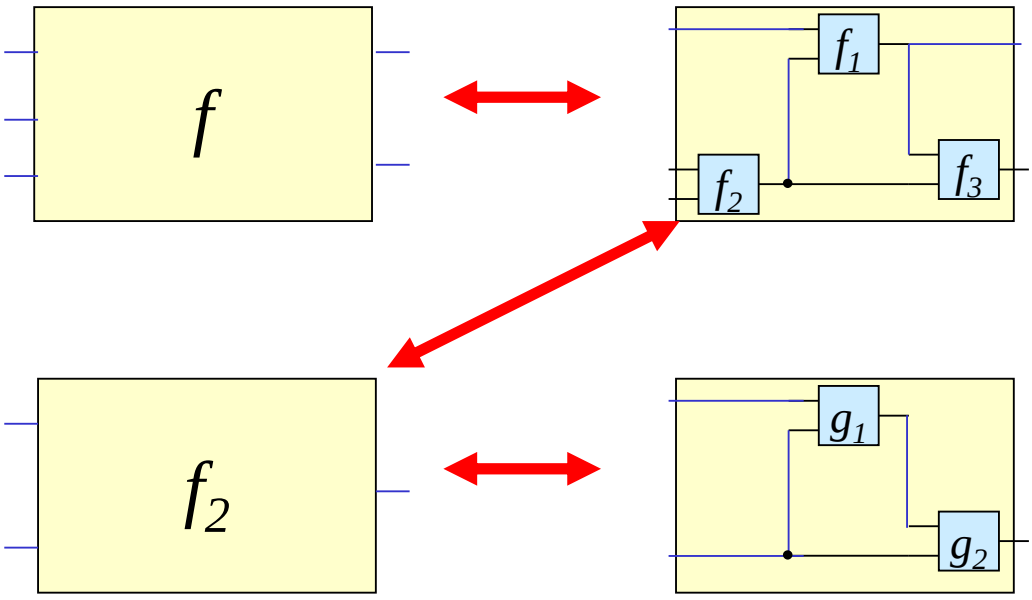
Grundprinzip des Entwurfs



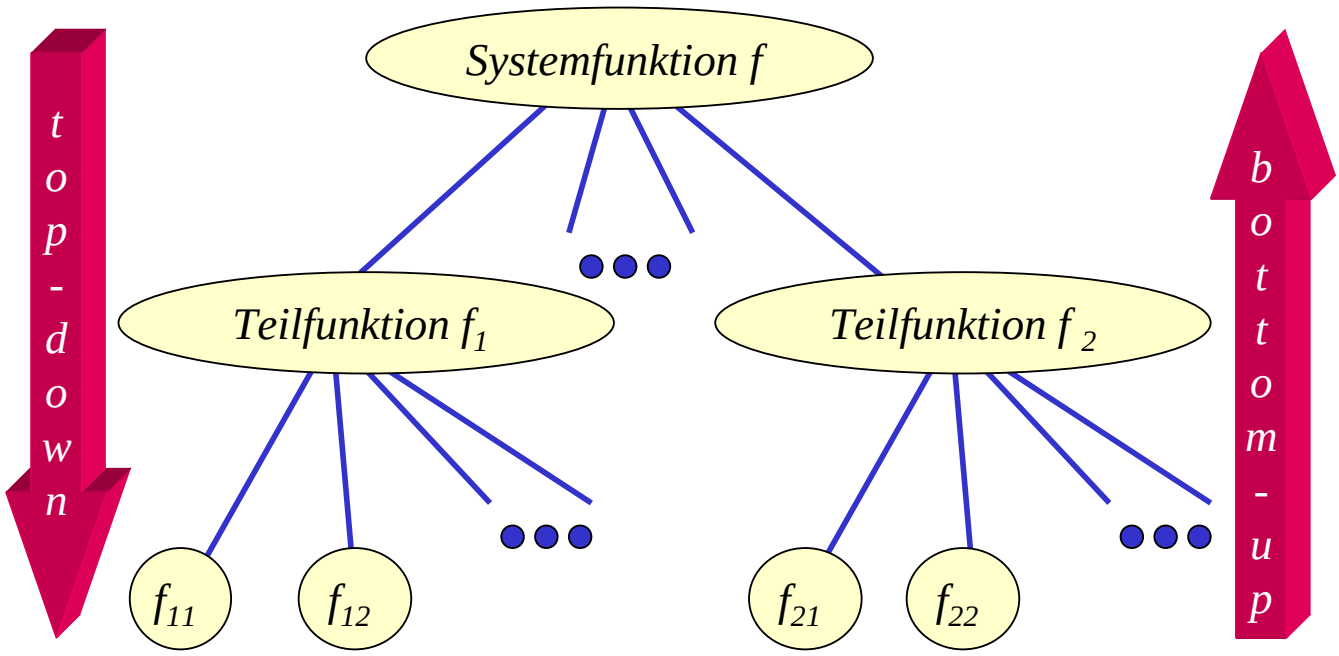
Komponenten + Struktur

= gewünschtes Verhalten

Abstraktion und Detaillierung



„top-down“ und „bottom-up“



Technische Kriterien für den Entwurf von Schaltnetzen

- **Korrekte Realisierung unter Beachtung des statischen und dynamischen Verhaltens der verwendeten Bauelemente**
- **Berücksichtigung technischer Beschränkungen (Anzahl der Eingänge, begrenzte Belastbarkeit der Ausgänge, zur Verfügung stehende Bausteine (Bausteinbibliothek), Temperaturgrenzen, Speicherplatz (bei PLAs), Taktfrequenz)**
- **Gewährleistung hoher Systemzuverlässigkeit (leichte Testbarkeit, Selbsttest, Fehlertoleranz, zuverlässiger Betrieb)**
- **Berücksichtigung von Forderungen an die Gebrauchseigenschaften (universelle Einsatzmöglichkeit, großer Funktionsumfang)**
- **Berücksichtigung technologischer Nebenbedingungen (Kühlung, Versorgungsspannung)**
- **Vermeidung von Störeinflüssen (elektromagnetische Felder)**

Ökonomische Kriterien für den Entwurf von Schaltnetzen

- Geringe Kosten für den Entwurf (Entwurfsaufwand)
  - ⇨ Lohnkosten
  - ⇨ Rechnerbenutzung, Softwarelizenzen
- Geringe Kosten für die Realisierung (Realisierungsaufwand)
  - ⇨ Bauelemente, Gehäuseformen
  - ⇨ Kühlung
- Geringe Kosten für die Inbetriebnahme
  - ⇨ Kosten für den Test
  - ⇨ Fertigstellung programmierbarer Bauelemente
- Geringe Kosten für den Betrieb
  - ⇨ Wartung
  - ⇨ Stromverbrauch

Entwurfsziele

- Einige Kriterien stehen miteinander im Widerspruch
  - ⇨ zuverlässigere Schaltungen erfordern einen höheren Realisierungsaufwand
  - ⇨ Verringerung des Realisierungsaufwand erfordert eine Erhöhung der Entwurfskosten
- Ziel des Entwurfs ist das Finden des günstigsten Kompromisses aus
  - ⇨ Korrektheit der Realisierung
  - ⇨ Einhaltung der technologischen Grenzen
  - ⇨ ökonomische Kriterien

☞ **Wir betrachten in dieser Vorlesung nur die Minimierung des Realisierungsaufwands**

2 Minimierungsverfahren

- Finden von Minimalformen Boolescher Funktionen
  - ohne Betrachtung der Zieltechnologie
  - mit Betrachtung der Zieltechnologie

- Drei Minimierungsansätze
  - algebraische Verfahren
  - graphische Verfahren
  - tabellarische Verfahren

- Man unterscheidet
  - exakte Minimierungsverfahren (z.B. Quine-McCluskey), deren Ergebnis das Minimum einer Schaltungsdarstellung ist
  - heuristische Minimierungsverfahren auf der Basis von iterativen Minimierungsschritten

Darstellung Boolescher Funktionen durch Funktionstabellen

- Darstellung des Verhaltens einer Booleschen Funktion mit Hilfe einer vollständigen Funktionstabelle
  - Jeder Belegung der Booleschen Variablen wird ein Funktionswert zugeordnet
  - $f(x_2, x_1, x_0) \rightarrow y$, *mit* $x_i, y \in \{0,1\}$

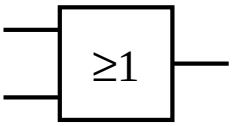
Index	x_2	x_1	x_0	y
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

$$f(x_2, x_1, x_0) = x_1\bar{x}_0 \vee x_2x_1 \vee x_2\bar{x}_1\bar{x}_0$$

Wichtige Funktionen

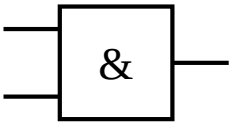
ODER

x_1	x_2	$x_1 \vee x_2$
0	0	0
0	1	1
1	0	1
1	1	1



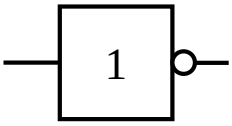
UND

x_1	x_2	$x_1 \wedge x_2$
0	0	0
0	1	0
1	0	0
1	1	1



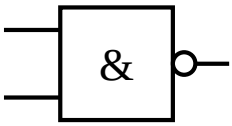
Nicht

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



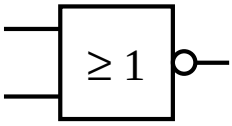
NAND

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



NOR

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



Zusammenfassung wichtiger Begriffe aus TI 1

- **Boolesche Variable** — Variable, die den Wert wahr (1) oder falsch (0) annehmen kann
- **Produktterm:** — UND-Verknüpfung von Booleschen Variablen
- **Implikant (einer Bool. Fkt.):** — Produktterm, der eine oder mehrere „1“-Stellen einer Booleschen Funktion beschreibt (impliziert)
- **Implikat (einer Bool. Fkt.):** — Disjunktion (ODER-Verknüpfung) von Literalen, die ein oder mehrer Nullstellen einer Booleschen Funktion beschreibt
- **Minterm:** — Implikant, der genau eine „1“-Stelle einer booleschen Funktion beschreibt
- **Maxterm:** — Implikat, das genau eine „0“-Stelle einer booleschen Funktion beschreibt
- **disjunktive Normalform:** — Darstellung der Funktion, die nur aus Mintermen besteht (DNF)
- **konjunktive Normalform:** — Darstellung der Funktion, die nur aus Maxtermen besteht (KNF)

Beispiel

Implikant $x_2 \wedge x_0$ → $x_2 \wedge x_1 \wedge x_0$, $x_2 \wedge \bar{x}_1 \wedge x_0$

- **DNF** $f(x_2, x_1, x_0) = x_2 x_1 x_0 \vee x_2 \bar{x}_1 x_0 \vee \bar{x}_2 x_1 \bar{x}_0 \vee \bar{x}_2 \bar{x}_1 \bar{x}_0$

Minterm

Boolesche Variable

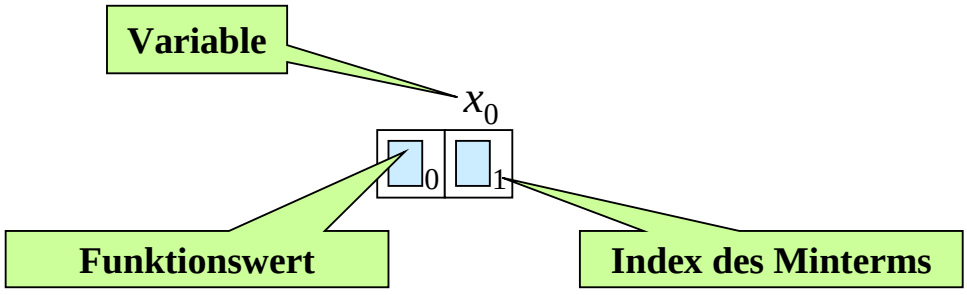
Maxterm

- **KNF** $f(x_2, x_1, x_0) = (x_2 \vee x_1 \vee x_0) \wedge (x_2 \vee x_1 \vee \bar{x}_0) \wedge (x_2 \vee \bar{x}_1 \vee \bar{x}_0)$

Implikat $x_2 \vee x_0$ → $x_2 \vee x_1 \vee x_0$, $x_2 \vee \bar{x}_1 \vee x_0$

2.1 KV-Diagramme

- **Nach Karnaugh und Veitch**
- **Möglichkeit, Boolesche Funktionen übersichtlich darzustellen**
  - ⇨ **bis 6 Variablen praktisch einsetzbar**
- **Ausgangspunkt ist ein Rechteck mit 2 Feldern**



KV-Diagramme

- **Beispiele**

x_0
0_0 | 1_1

$f(x_0) = x_0$

x_0
1_0 | 0_1

$f(x_0) = \bar{x}_0$

- **Erweiterung durch Spiegelung**
  - ⇨ **für jede zusätzliche Variable verdoppelt sich die Zahl der Felder**

x_0	
0	1
2	3

x_1

$\overline{x_0}$			
0	1	5	4
2	3	7	6

x_1 $\quad$ $\overline{x_2}$

$\overline{x_0}$			
0	1	5	4
2	3	7	6
10	11	15	14
8	9	13	12

x_3 $\quad$ x_1 $\quad$ $\overline{x_2}$

Eigenschaften von KV-Diagrammen

- **Jedes Feld ist ein Funktionswert**
  - ⇨ **ein Minterm der Funktion**
  - ⇨ **eindeutige Variablenzuordnung**
- **Oft werden x_1 und x_2 vertauscht**
  - ⇨ **ist lediglich andere Numerierung der Felder**
  - ⇨ **kein Einfluss auf das Minimierungsverfahren**
- **Aufstellen der KV-Diagramme über die Funktionstabelle:**

Index	x_2	x_1	x_0	y
0	0	0	0	0
1	0	0	1	0
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

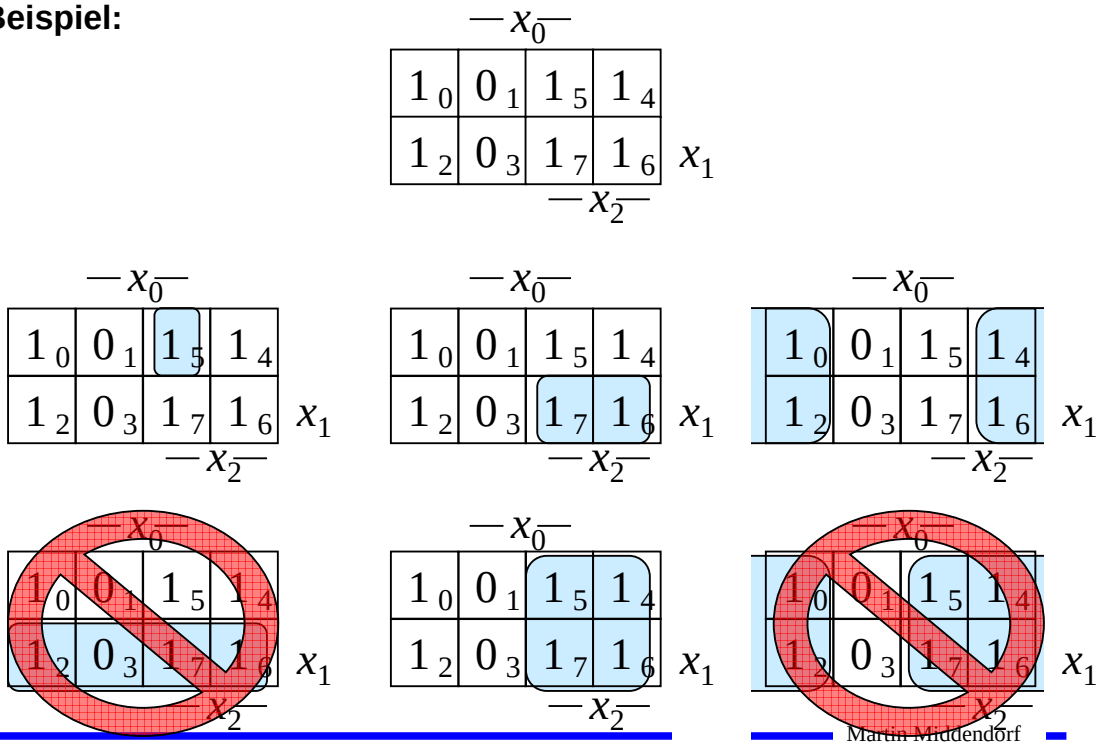
$f(x_2, x_1, x_0) = x_1\bar{x}_0 \vee x_2x_1 \vee x_2\bar{x}_1\bar{x}_0$

$\overline{x_0}$			
0_0	0_1	0_5	1_4
1_2	0_3	1_7	1_6

x_1 $\quad$ $\overline{x_2}$

Minimalformen aus KV-Diagrammen

- **Zusammenfassen von Mintermen zu Implikanten höherer Ordnung**
- **Beispiel:**



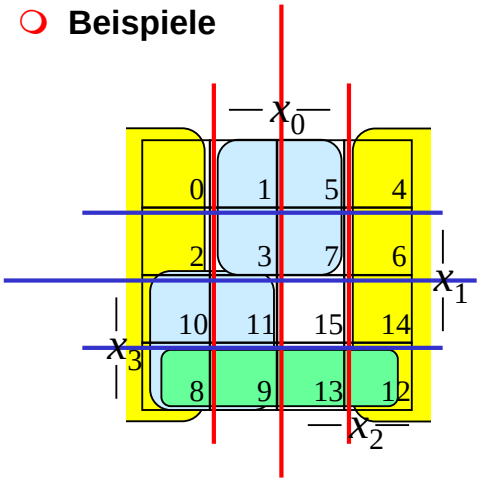
Implikant k-ter Ordnung

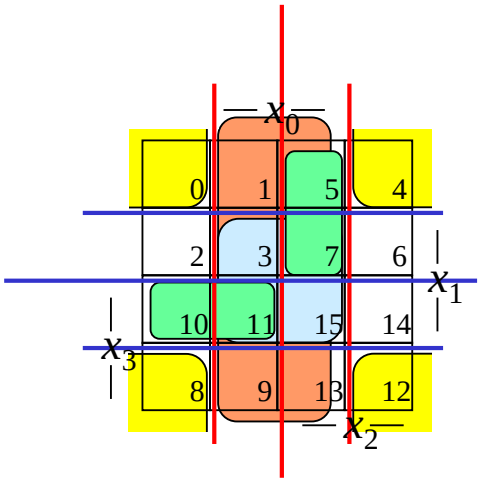
Def. 10.1: Es sei eine Boolesche Funktion $f(x_0,...,x_{n-1}):B^n \rightarrow B$ gegeben. Ein Implikant k-ter Ordnung umfaßt 2^k Felder eines KV-Diagramms.

- **Man erhält**
  - ⇨ **Implikanten 0-ter Ordnung** — **Minterme**
  - ⇨ **Implikanten 1-ter Ordnung** — **Zusammenfassung zweier Minterme**
  - ⇨ **Implikanten 2-ter Ordnung** — **Zusammenfassung zweier Implikanten 1-ter Ordnung**
  - ⇨ **usw.**

Minimalformen aus KV-Diagrammen

- Finden von 1-Blöcken, die symmetrisch zu denjenigen Achsen sind, an denen eine Variable von 0 auf 1 wechselt
- Jede Funktion läßt sich als disjunktive Verknüpfung solcher Implikanten darstellen
- Beispiele



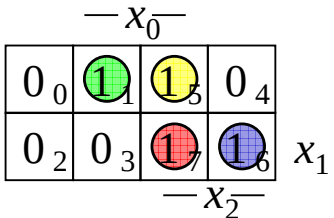


Primimplikant

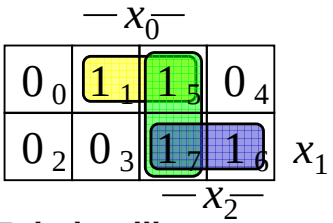
Def. 10.2: Es sei eine Boolesche Funktion $f(x_0,\ldots,x_{n-1}):B^n \rightarrow B$ gegeben. Ein Implikant p heißt Primimplikant, wenn es keinen Implikanten q gibt, der p impliziert.

- Ein Primimplikant p ist von größtmöglicher Ordnung
  - ⇨ Primimplikanten sind einfach aus einem KV-Diagramm herauszulesen
  - ⇨ man sucht die größtmöglichen Implikanten

$$f(x_2,x_1,x_0) = x_2x_1\bar{x}_0 \vee x_2x_0 \vee \bar{x}_2\bar{x}_1x_0$$



Minterme



Primimplikanten

Überdeckung

Satz 10.1: Zu jeder Booleschen Funktion *f* gibt es eine minimale Überdeckung aus Primimplikanten

Bew. (Skizze):

Angenommen wir haben eine minimale Überdeckung der Funktion, die einen Implikanten *k* besitzt, der kein Primimplikant ist.

⇒ Dieser Implikant *k* kann durch einen Primimplikant *p* ersetzt werden, der *k* enthält

⇒ Das Ergebnis ist eine Überdeckung der Funktion *f* aus Primimplikanten mit der gleichen Anzahl von Termen und echt mehr Primimplikanten enthält

⇒ Die Überdeckung ist minimal

- **Einschränkung des Suchraums**
  - ⇨ **man braucht nur die Primimplikanten für die Minimierung betrachten**

Kernprimimplikant

Def. 10.3: Es sei eine Boolesche Funktion $f(x_0,\ldots,x_{n-1}):B^n \rightarrow B$ gegeben. Ein Implikant *p* heißt Kernprimimplikant, wenn er einen Minterm überdeckt, der von keinem anderen Primimplikant überdeckt wird.

- **Man nennt solche Primimplikanten auch essentielle Primimplikanten**
  - ⇨ **Ein Kernprimimplikant muss auf jeden Fall in der disjunktiven Minimalform vorkommen**

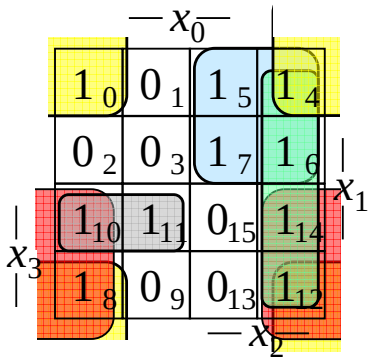
- **Ziel der Minimierung:**
  - ⇨ **Überdecken der Funktion durch Kernprimimplikanten und möglichst wenige zusätzliche Primimplikanten**

- **Zwei Schritte**
  1. **Finde alle Primimplikanten**
  2. **Suche eine Überdeckung der Funktion mit möglichst wenigen Primimplikanten**

Beispiel

$$f(x_3,x_2,x_1,x_0) = \bar{x}_3\bar{x}_2\bar{x}_1\bar{x}_0 \vee \bar{x}_3x_2\bar{x}_1\bar{x}_0 \vee \bar{x}_3x_2\bar{x}_1x_0 \vee$$
$$\bar{x}_3x_2x_1\bar{x}_0 \vee \bar{x}_3x_2x_1x_0 \vee x_3\bar{x}_2\bar{x}_1\bar{x}_0 \vee$$
$$x_3\bar{x}_2x_1\bar{x}_0 \vee x_3\bar{x}_2x_1x_0 \vee x_3x_2\bar{x}_1\bar{x}_0 \vee x_3x_2x_1\bar{x}_0$$
$$= MINt(0, 4, 5, 6, 7, 8, 10, 11, 12, 14)$$

DNF



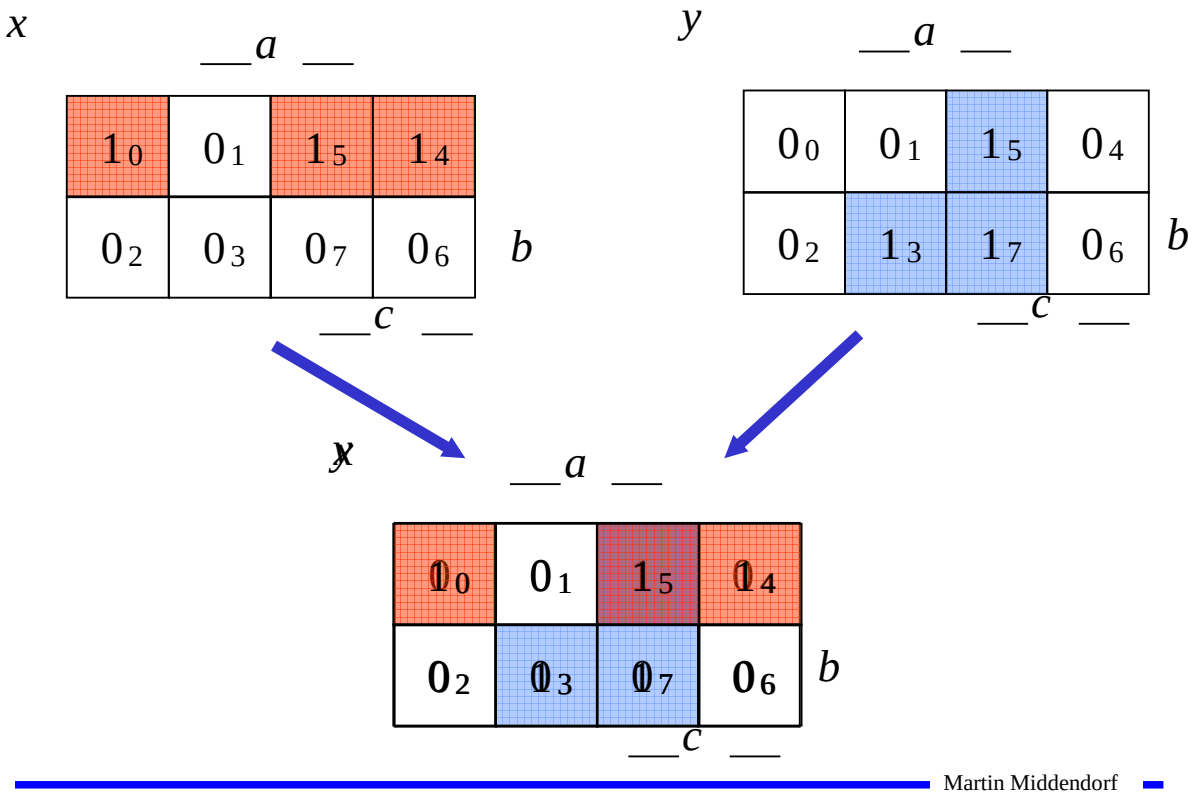
e	$\bar{x}_1\bar{x}_0$	(0,4,8,12)
e	$\bar{x}_3x_2$	(4,5,6,7)
	$x_2\bar{x}_0$	(4,6,12,14)
	$x_3\bar{x}_0$	(8,10,12,14)
e	$x_3\bar{x}_2x_1$	(10,11)

e = essentiell

$$f(x_3,x_2,x_1,x_0) = \bar{x}_1\bar{x}_0 \vee \bar{x}_3x_2 \vee x_3\bar{x}_0 \vee x_3\bar{x}_2x_1$$
$$= \bar{x}_1\bar{x}_0 \vee \bar{x}_3x_2 \vee x_2\bar{x}_0 \vee x_3\bar{x}_2x_1$$

DMF

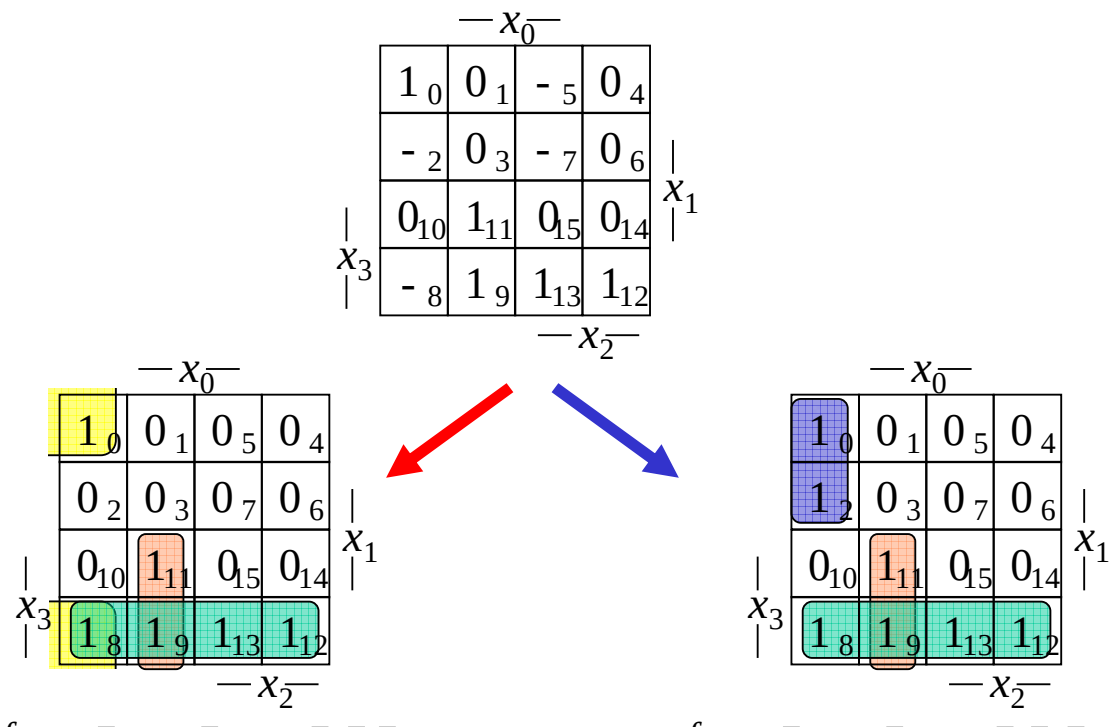
2.2 Bündelminimierung



2.3 Unvollständig definierte Funktionen

- Bisher war für alle Variablenbelegungen ein Funktionswert festgelegt
  - ⇨ in praktischen Fällen kommt es sehr häufig vor, dass die Funktionswerte für bestimmte Variablenbelegungen frei wählbar sind
- Solche Funktionen heißen unvollständig oder partiell definierte Funktionen
  - ⇨ die nicht verwendeten Variablenbelegungen heißen auch Don‘t-care-Belegungen
  - ⇨ in KV-Diagrammen werden diese Felder mit einem „-“ gekennzeichnet
- wichtiges Potential für die Minimierung!
  - ⇨ um eine DMF zu erhalten, müssen diese mit „0“ oder „1“ belegt werden

Minimierung unvollständiger Boolescher Funktionen



2.4 Das Verfahren nach Quine-McCluskey

- KV-Diagramme mit mehr als 6 Variablen werden sehr groß und unübersichtlich
  - ⇨ dieses Problem wurde zuerst von Quine und McCluskey erkannt und gelöst
  - ⇨ Verfahren nach Quine-McCluskey ist Tabellenbasiert
  - ⇨ es führt auf eine DMF (disjunktive minimale Form)

- Ausgangspunkt ist die Funktionstabelle der Funktion
  - ⇨ nur die Minterme werden berücksichtigt

- Der Suchraum wird gemäß Satz 1.1 eingeschränkt:
  - ⇨ zu jeder Booleschen Funktion *f* gibt es eine minimale Überdeckung aus Primimplikanten

- Verfahren nach Quine McCluskey in 2 Schritten:
  1. Schritt: berechne alle Primimplikanten
  2. Schritt: suche eine minimale Überdeckung aller Minterme

Beispiel: Die vollständige Funktionstabelle

Nr.	e	d	c	b	a	y
0	0	0	0	0	0	0
1	0	0	0	0	1	0
2	0	0	0	1	0	1
3	0	0	0	1	1	0
4	0	0	1	0	0	1
5	0	0	1	0	1	1
6	0	0	1	1	0	1
7	0	0	1	1	1	0
8	0	1	0	0	0	0
9	0	1	0	0	1	0
10	0	1	0	1	0	1
11	0	1	0	1	1	0
12	0	1	1	0	0	1
13	0	1	1	0	1	1
14	0	1	1	1	0	1
15	0	1	1	1	1	0

Nr.	e	d	c	b	a	y
16	1	0	0	0	0	0
17	1	0	0	0	1	0
18	1	0	0	1	0	1
19	1	0	0	1	1	0
20	1	0	1	0	0	0
21	1	0	1	0	1	0
22	1	0	1	1	0	1
23	1	0	1	1	1	0
24	1	1	0	0	0	0
25	1	1	0	0	1	0
26	1	1	0	1	0	1
27	1	1	0	1	1	0
28	1	1	1	0	0	0
29	1	1	1	0	1	0
30	1	1	1	1	0	1
31	1	1	1	1	1	0

1. Schritt: Berechnung aller Primimplikanten

- Schreibweise
  - ⇨ 1 steht für eine nicht negierte Variable
  - ⇨ 0 steht für eine negierte Variable
  - ⇨ - steht für eine nicht auftretende Variable
- Man betrachtet nur die Minterme (1-Stellen der Funktion)

- Die Minterme werden geordnet
  - ⇨ Gruppen mit der gleichen Anzahl von Einsen
  - ⇨ innerhalb der Gruppen: aufsteigende Reihenfolge
  - ⇨ man erhält die 1. Quinesche Tabelle, 0. Ordnung

- Minterme benachbarter Gruppen die sich nur in einer Variable unterscheiden werden gesucht
  - ⇨ diese können durch Streichen der Variable zusammengefaßt werden
  - ⇨ man erhält die 1. Quineschen Tabellen höherer Ordnung

Beispiel: 1. Quinesche Tabelle

Nr.	e	d	c	b	a
2	0	0	0	1	0
4	0	0	1	0	0
5	0	0	1	0	1
6	0	0	1	1	0
10	0	1	0	1	0
12	0	1	1	0	0
18	1	0	0	1	0
13	0	1	1	0	1
14	0	1	1	1	0
22	1	0	1	1	0
26	1	1	0	1	0
30	1	1	1	1	0

0. Ordnung

Nr.	e	d	c	b	a
2,6	0	0	-	1	0
2,10	0	-	0	1	0
2,18	-	0	0	1	0
4,5	0	0	1	0	-
4,6	0	0	1	-	0
4,12	0	-	1	0	0
5,13	0	-	1	0	1
6,14	0	-	1	1	0
6,22	-	0	1	1	0
10,14	0	1	-	1	0
10,26	-	1	0	1	0
12,13	0	1	1	0	-
12,14	0	1	1	-	0
18,22	1	0	-	1	0
18,26	1	-	0	1	0
14,30	-	1	1	1	0
22,30	1	-	1	1	0
26,30	1	1	-	1	0

1. Ordnung

Nr.	e	d	c	b	a
2,6,10,14	0	-	-	1	0
2,6,18,22	-	0	-	1	0
2,10,18,26	-	-	0	1	0
4,5,12,13	0	-	1	0	- *A*
4,6,12,14	0	-	1	-	0 *B*
6,14,22,30	-	-	1	1	0
10,14,26,30	-	1	-	1	0
18,22,26,30	1	-	-	1	0

2. Ordnung

Nr.	e	d	c	b	a
2,6,10,14 18,22,26,30	-	-	-	1	0 *C*

3. Ordnung

2. Schritt: Suche einer minimalen Überdeckung

- Aufstellen der 2. Quineschen Tabelle
  - ⇨ alle Primimplikanten werden zusammen mit den Nummern der Minterme aus denen sie hervorgegangen sind in eine Überdeckungstabelle eingetragen
- Kosten für einen Primimplikanten:
  - ⇨ Anzahl der UND-Eingänge (Anzahl der Variablen des Terms)

Primimplikant	2	4	5	6	10	12	13	14	18	22	26	30	Kosten
A		X	X			X	X						3
B		X		X		X		X					3
C	X			X	X			X	X	X	X	X	2

- Aufgabe: Finden einer Überdeckung aller Minterme mit minimalen Kosten

Systematische Lösung des Überdeckungsproblems

- Aufstellung einer Überdeckungsfunktion $\ddot{u}_f$
  - ⇨ w_A, w_B und w_C sind Variablen, die kennzeichnen, ob ein entsprechender Primimplikant in der vereinfachten Darstellung aufgenommen wird, oder nicht
  - ⇨ Konjunktive Form über alle den jeweiligen Minterm überdeckenden Primimplikanten

Primimplikant	2	4	5	6	10	12	13	14	18	22	26	30
A		X	X			X	X					
B		X		X		X		X				
C	X			X	X			X	X	X	X	X

$$\ddot{u}_f = w_C(w_A \vee w_B)w_A(w_B \vee w_C)w_C(w_A \vee w_B)w_A(w_B \vee w_C)w_C w_C w_C w_C$$
$$= w_C(w_A \vee w_B)w_A(w_B \vee w_C)$$
$$= (w_C w_A \vee w_C w_B)(w_A w_B \vee w_A w_C)$$
$$= w_C w_B w_A \vee w_A w_C$$
$$(= w_A w_C)$$

Systematische Lösung des Überdeckungsproblems

- **Ergebnis nach der Vereinfachung:** $\ddot{u}_f = w_C w_B w_A \vee w_A w_C$
- **Man sucht einen konjunktiven Term mit minimalen Kosten**

$$w_C w_B w_A \text{ Kosten}: 2+3+3=8$$
$$w_A w_C \quad \text{Kosten}: 3+2=5$$

- **Als Endergebnis der Minimierung für die Funktion f erhält man**

$$f(e,d,c,b,a) = \bar{e}c\bar{b} \vee b\bar{a}$$

Vereinfachung des Überdeckungsproblems

- **Die Primimplikantentabelle kann folgerndermaßen reduziert werden:**

1. **Kernimplikantenregel: essentielle Primterme (Kernprimimplikanten) und die von ihnen überdeckten Minterme können gestrichen werden**
   - ⇨ **tragen mit einem einzigen „X“ zu einer Spalte bei**
   - ⇨ **müssen auf jeden Fall in der Überdeckung enthalten sein**

Beispiel: Hier sind dies die beiden Primimplikanten A und C

Primimplikant	2	4	5	6	10	12	13	14	18	22	26	30	Kosten
A			X	X			X	X					3
B			X		X		X		X				3
C		X			X	X			X	X	X	X	X	2

- ⇨ **A: 5, 13**
- ⇨ **C: 2, 10, 18, 22, 26, 30**
- ⇨ **B ist vollständig überdeckt und kann ebenfalls gestrichen werden**

Weitere Vereinfachungen

- Weitere Reduktionsregeln zur Anwendung auf die Primimplikantentabelle

Def.: Ein Minterm n dominiert einen Minterm m, wenn jeder Primimplikant, der n überdeckt auch m überdeckt

2. **Spaltenregel: Entferne alle Minterme, die von einem anderen Minterm dominiert werden**

Primimplikant	4	6	10	13	18	22	26
A	X	X	X		X		
B			X		X	X	X
C	X	X		X		X	X
D		X		X		X	

Entferne: 6, 10 oder 18, 22

Weitere Vereinfachungen

Primimplikant	4	10	13	26	Kosten
A	X	X			z
B		X		X	z
C	X		X	X	z
D			X		z

3. **Zeilenregel: Entferne alle Primimplikanten, die durch einen anderen nicht teureren Primimplikanten dominiert werden.**

Annahme: Im obigen Beispiel haben alle Primimplikanten die gleichen Kosten z → Entferne: D

Primimplikant	4	10	13	26
A	X	X		
B		X		X
C	X		X	X

Beobachtung: Es kann jetzt wieder die erste Reduktionsregel angewendet, da C essentiell ist.

Weitere Vereinfachungen

- Wenn man keine der Vereinfachungsregeln mehr anwenden kann, erhält man eine reduzierte Tabelle, auf die man andere Verfahren anwendet.

  Das Minimierungsproblem auf der so reduzierten Tabelle ist NP-vollständig.

Aufwandsbetrachtungen

- Die Zeit zum Aufstellen der 1. Quineschen Tabellen ist in $O(3^n n^2)$.

Beweis: Es gibt $\binom{n}{i}$ Möglichkeiten i Variable aus n Variablen auszuwählen.

Jede Variable kann entweder positiv oder negiert vorkommen.

Die maximale Anzahl von (Prim)implikanten ist (verwende den Binomialsatz) höchstens:

$$\sum_{i=0}^{n}\binom{n}{i}2^i = \sum_{i=0}^{n}\binom{n}{i}2^i 1^{n-i} = (2+1)^n = 3^n$$

Jeder Primimplikant wird mit höchstens n weiteren verglichen.

Jede Such- und Einsetzoperation kann man mit Hilfe von geeigneten Datenstrukturen (Heaps) in Zeit $O(\log 3^n)=O(n)$ durchführen.

Aufwandsbetrachtungen

- Alle Verfahren benötigen 2 Schritte
  - ⇨ 1. Erzeugen aller Primimplikanten (Primimplikate)
  - ⇨ 2. Auswahl der Primiplikanten (Primimplikate), welche die Minterme (Maxterme) mit minimalen Kosten überdecken

- Die Anzahl der Primimplikanten (Primimplikaten) kann exponentiell steigen
  - ⇨ Es gibt Funktionen mit $\frac{3^n}{n}$ Primimplikanten

- Das Überdeckungsproblem ist NP-vollständig
  - ⇨ Es besteht wenig Hoffnung einen Algorithmus zu finden, der dieses Problem in einer Zeit löst, die polynomiell in der Zahl der Eingabevariablen ist.

Heuristische Verfahren

- Heuristische Minimierungsverfahren werden eingesetzt,
  - ⇨ wenn die zweistufige Darstellung optimiert werden muss, aber
  - ⇨ nur begrenzt Rechenzeit und Speicherplatz zur Verfügung steht
- Die meisten heuristischen Minimierungsansätze basieren auf einer schrittweisen Verbesserung der Schaltung

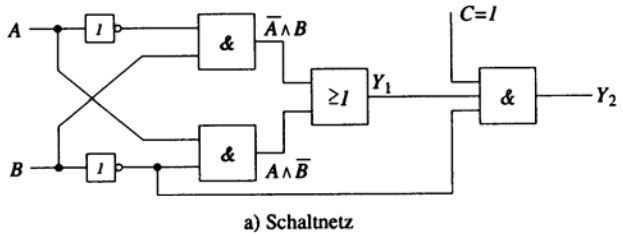
- Unterschiede zu exakten Verfahren:
  - ⇨ man wendet eine Menge von Transformationen direkt auf die Überdeckung des *ON-Sets* an
  - ⇨ man definiert die Optimierung als beendet, wenn diese Transformationen keine Verbesserungen mehr bringen

- Mehr dazu in der Vorlesung „Entwurf hochintegrierter Schaltungen“

2.5 Laufzeiteffekte in Schaltnetzen

- Bisher wurden Schaltnetze mit idealen Verknüpfungsgliedern betrachtet
  - ⇨ die Verknüpfungsglieder besaßen keine Signallaufzeit
- Bei realen Verknüpfungsgliedern dürfen Signallaufzeiten nicht vernachlässigt werden
  - ⇨ Schaltvariablen können Werte annehmen, die theoretisch oder bei idealen Verknüpfungsgliedern nie auftreten könnten
- Solche Störimpulse nennt man Hazards
  - ⇨ sie treten als Antwort auf die Änderung der Werte der Eingangsvariablen auf

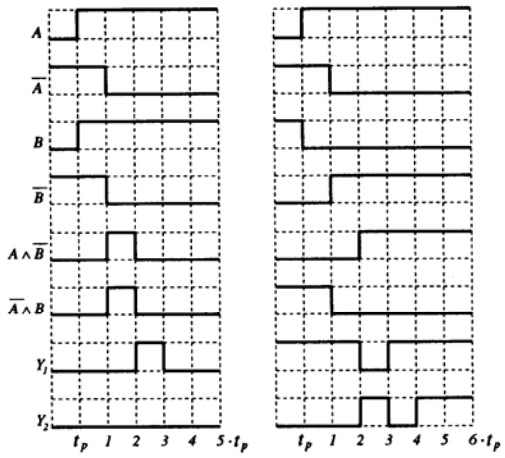
Entstehung von Hazards



a) Schaltnetz

B	A	Y_1	Y_2
0	0	0	0
1	1	0	0
1	0	1	0
0	1	1	1

b) Funktionstabelle



a) Impulsdiagramm

Statische Hazards

- **Statische Hazards sind Störimpulse aus einer Verknüpfung, die theoretisch konstant Null oder Eins liefern müsste**

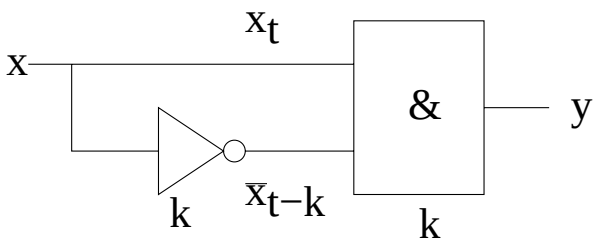
$X_t \wedge \overline{X}_{t-k}$ **müsste Null liefern**
statischer 1-Hazard bei einem Übergang von X: 0→1

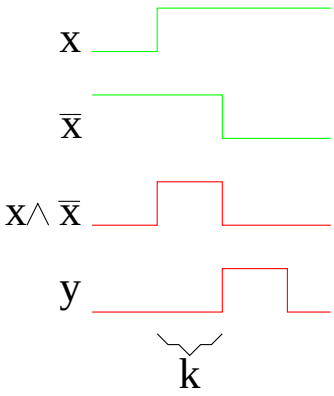
$X_t \vee \overline{X}_{t-k}$ **müsste Eins liefern**
statischer 0-Hazard bei einem Übergang von X: 1→0

Statische Hazards

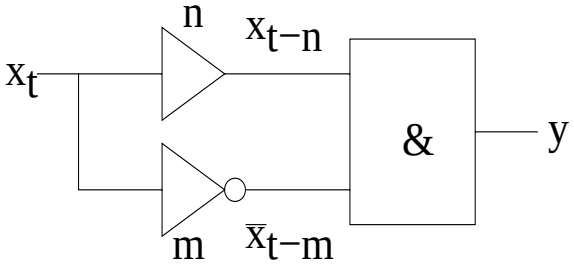
Statischer 1-Hazard

0 → 1





Allgemein:

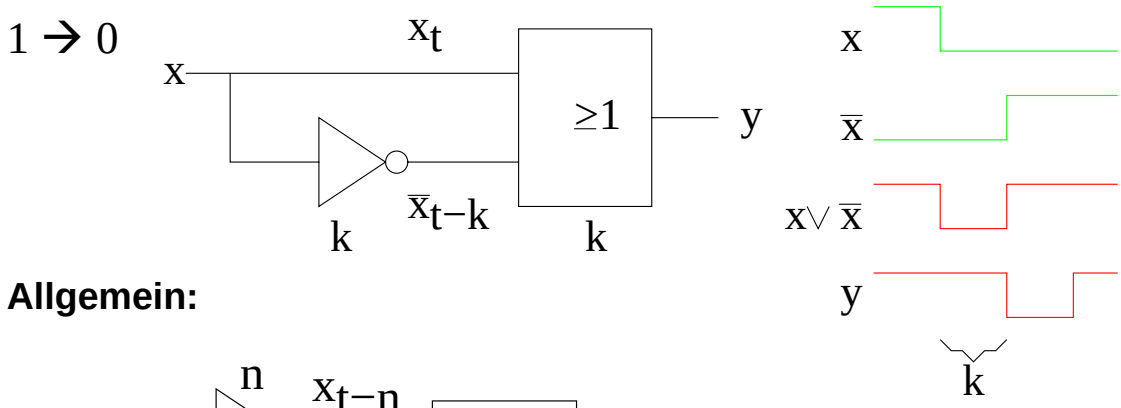


n < m: 0 → 1

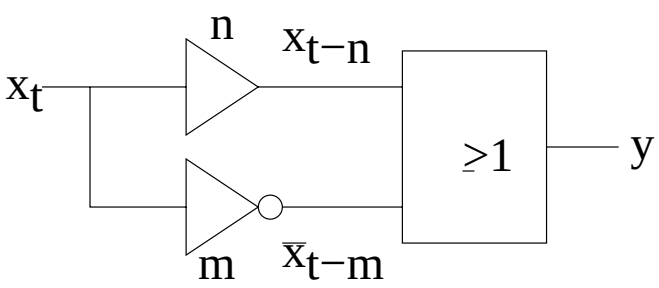
n > m: 1 → 0

Statische Hazards

Statischer 0-Hazard



Allgemein:



n < m: 1 → 0

n > m: 0 → 1

Dynamische Hazards

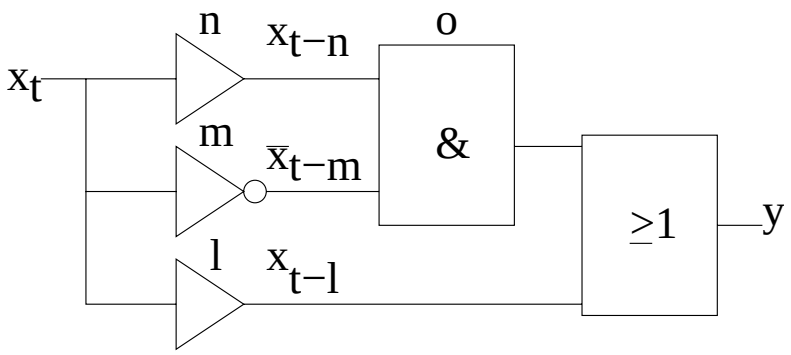
- **Dynamische Hazards entstehen als zusätzliche Übergänge beim Ausgang eines Schaltnetzes**

- $X_t \wedge \overline{X}_{t-k} \vee X_{t-l}$, mit $l > k$
  - ⇨ **bei einem Übergang von $X{=}0 \rightarrow X{=}1$ darf am Ausgang nur ein zu X_{t-l} synchroner 0 →1 Übergang auftreten**
  - ⇨ **durch den vorgeschalteten statischen Hazard kommt es aber zu einer zusätzlichen 0 →1 Flanke**

- $X_t \wedge (\overline{X}_{t-k} \vee X_{t-l})$, mit $l < k$
  - ⇨ **bei einem Übergang von $X{=}1 \rightarrow X{=}0$ darf am Ausgang nur ein zu X_t synchroner 1 →0 Übergang auftreten**
  - ⇨ **durch den vorgeschalteten statischen Hazard kommt es aber zu einer zusätzlichen 1 →0 Flanke**

Dynamische Hazards

Dynamischer Hazard

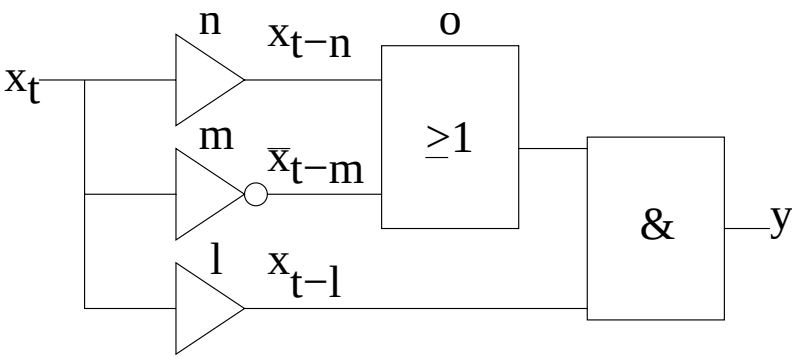


0 → 1: l > m+o > n+o

1 → 0: n+o > m+o > l

Dynamische Hazards

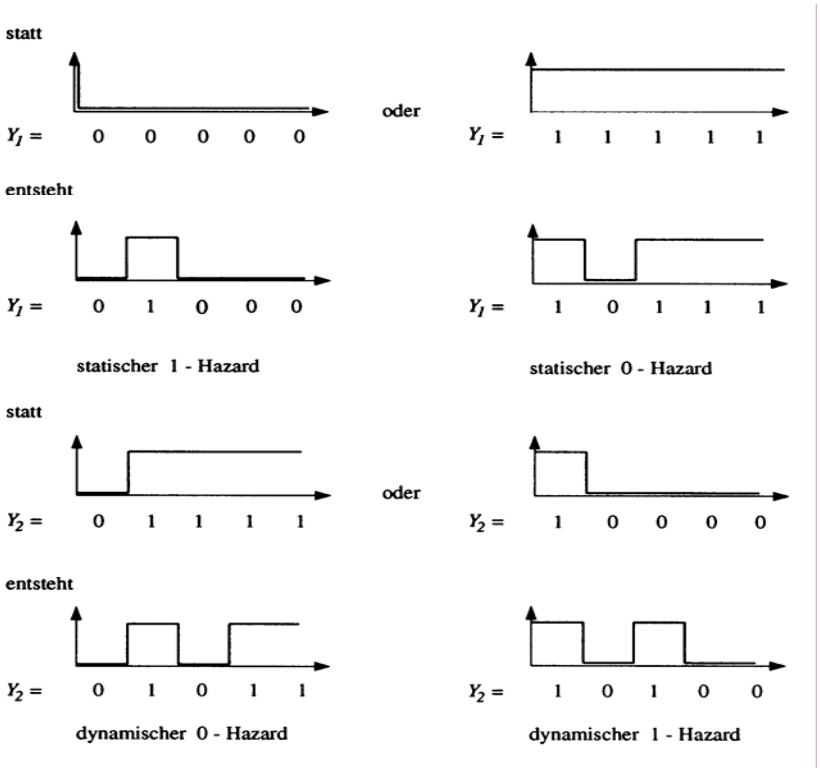
Dynamischer Hazard



1 → 0: l > m+o > n+o

0 → 1: n+o > m+o > l

Klassifikation von Hazards



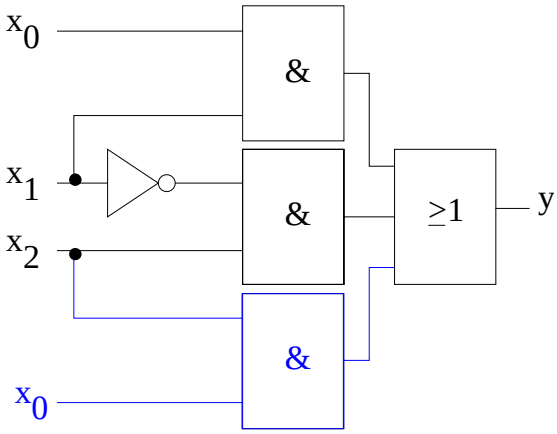
Behebung von Hazards

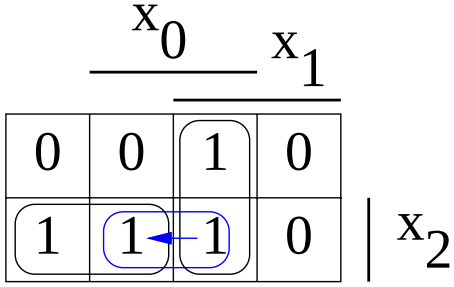
- Hazards können die Funktion von Schaltnetzen stören
  - ⇨ falsche Werte können an den Eingang eines Schaltnetzes zurückgekoppelt werden
- Zur Vermeidung solcher Fehler
  - ⇨ werden taktflankengetriggerte Speicherglieder in Rückkopplungen eingefügt und
  - ⇨ Signale erst übernommen, wenn die Hazards abgeklungen sind
    - nur stabile, gültige Werte werden übernommen
    - synchrone Schaltwerke: Steuerung durch einen zentralen Takt
- Hazards haben Einfluss auf die maximale Schaltgeschwindigkeit
  - ⇨ maximaler Takt
  - ⇨ Entfernung von Hazards führt zur Erhöhung der Funktionsgeschwindigkeit einer Schaltung

Behebung von Hazards

Realisierung zusätzlicher Primimplikanten:

Betrachte Wechsel 1 → 0 von x_1 bei konstanten $x_0=x_2=1$



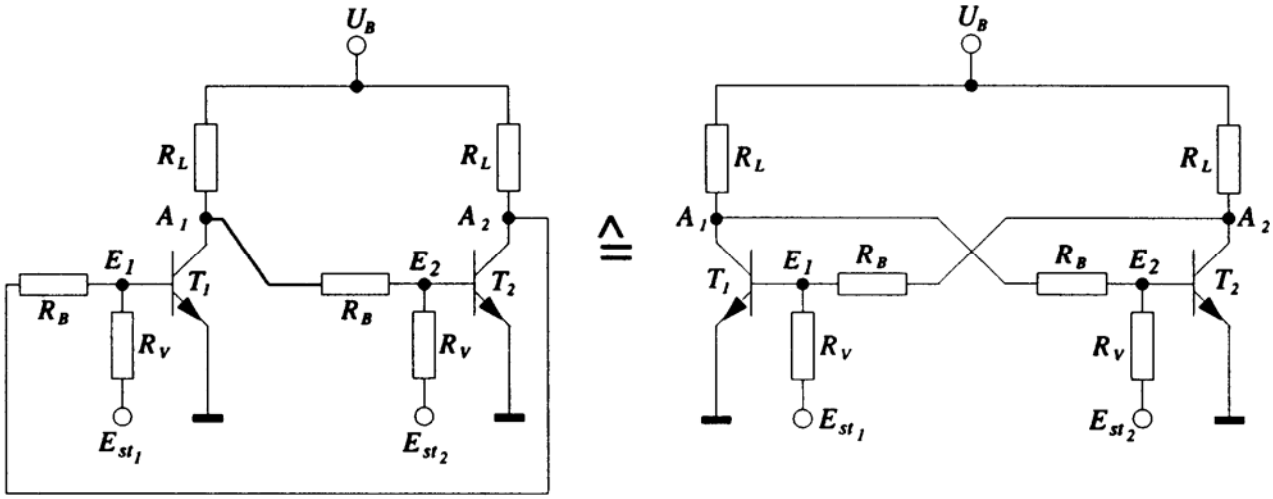


Ohne den zusätzlichen „blauen“ Primimplikanten würde ein 0-Hazard auftreten.

3 Speicherglieder

- **Speicherglieder**
  - ⇨ **dienen zur Aufnahme, Speicherung und Abgabe der Werte von Schaltvariablen**
  - ⇨ **sind bistabile Kippschaltungen (Flipflops)**
- **Zwei Zustände**
  - ⇨ **Zustand 1: Setzzustand**
  - ⇨ **Zustand 0: Rücksetzzustand**
- **Übernahme des Zustands kann erfolgen**
  - ⇨ **taktunabhängig (nicht taktgesteuert)**
  - ⇨ **taktabhängig (taktgesteuert)**
    - **taktzustandsgesteuert**
    - **taktflankengesteuert**
- **Unterschiedlichen Arten der Ansteuerungen führen zu unterschiedlichen Flipflop-Typen**

Funktionsprinzip



- **Rückkopplung**
  - **Wirkprinzip aller bistabilen Kippschaltungen**
  - **Ein Kippvorgang von einem stabilen Zustand in den anderen wird durch E_{st1} und E_{st2} ausgelöst**

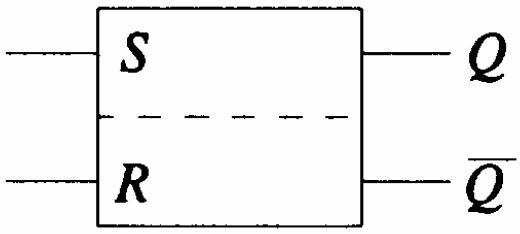
Funktionsprinzip

- **Nach dem Anlegen von U_B sei T_2 leitend, T_1 sperrt**
  - **A_1 besitzt H-Pegel und A_2 besitzt L-Pegel**
  - **dieser Zustand ist stabil**
- **Wird E_{st1} auf H-Pegel gesetzt, so**
  - **wird T_1 leitend, A_1 geht auf L-Pegel**
  - **T_2 sperrt und A_2 geht auf H-Pegel**
  - **dieser Zustand ist ebenfalls stabil**
- **Wird E_{st2} auf H-Pegel gesetzt, so**
  - **wird T_2 leitend, A_2 geht auf L-Pegel**
  - **T_1 sperrt und A_1 geht auf H-Pegel**
  - **dieser Zustand ist wiederum stabil**
- **Werden E_{st1} und E_{st2} auf H-Pegel gesetzt, so**
  - **leiten beide Transistoren, die Rückkopplung wird unwirksam**
  - **dieser Zustand ist nicht stabil**
  - **unzulässige Eingangsbelegung**

RS-Flipflop

- **Bistabile Kippschaltungen können gebaut werden aus rückgekoppelten**
  - ⇨ **Transistoren**
  - ⇨ **NOR-Gattern**
  - ⇨ **NAND-Gattern**

- **RS-Flipflop**
  - ⇨ **wenn die Eingänge den Wert 0 haben, bleibt der vorherige Zustand stabil**
  - ⇨ **wird S =1, dann Q=1 und $\overline{Q}$=0**
  - ⇨ **wird R=1, dann Q=0 und $\overline{Q}$=1**
  - ⇨ **gleichzeitig S=1 und R=1 ist nicht zulässig**

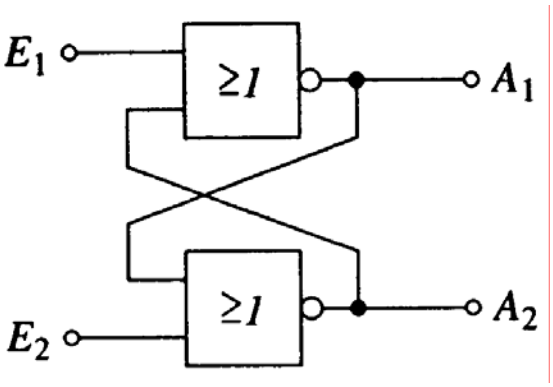


Schaltzeichen für ein RS-Flipflop nach DIN

RS-Flipflop aus NOR-Gattern

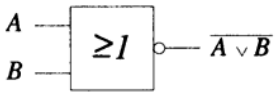
- **Liegt an einem Eingang eines NOR-Gatters eine 1 an, so geht der entsprechende Ausgang auf 0**

- **Liegt an beiden Eingängen eine 0 an, so bleiben die Ausgänge erhalten**



Funktionstabelle der Ausgänge A_1 und A_2

E_1	E_2	A_1	A_2
0	0	(wie vorher) speichern	
0	1	1	0
1	0	0	1
1	1	(0	0) unzulässig



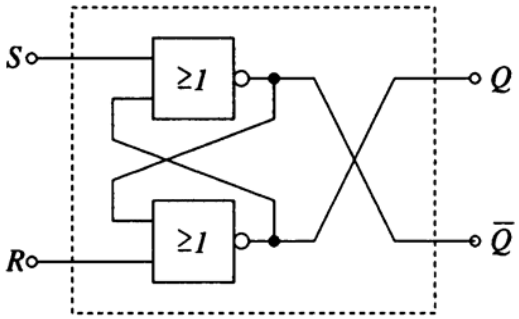
B	A	$\overline{A \vee B}$
0	0	1
0	1	0
1	0	0
1	1	0

RS-Flipflop aus NOR-Gattern

- Ein RS-Flipflop entsteht durch Vertauschen der Ausgänge

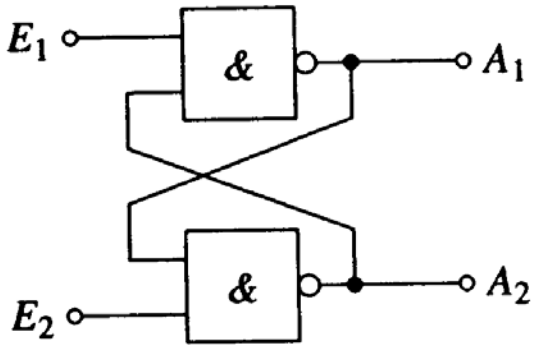
Funktionstabelle

S	R	Q	$\overline{Q}$	
0	0			(wie vorher) speichern
0	1	0	1	
1	0	1	0	
1	1	(0	0)	unzulässig



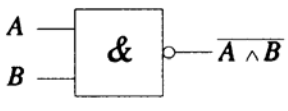
RS-Flipflop aus NAND-Gattern

- Liegt an beiden Eingängen eines NAND-Gatters 1 an, so geht der entsprechende Ausgang auf 0
- Liegt an beiden Eingängen der Schaltung 1 an, so bleiben die Ausgänge erhalten



Funktionstabelle der Ausgänge A_1 und A_2

E_1	E_2	A_1	A_2	
0	0	(1	1)	(unzulässig)
0	1	1	0	
1	0	0	1	
1	1			(wie vorher) speichern



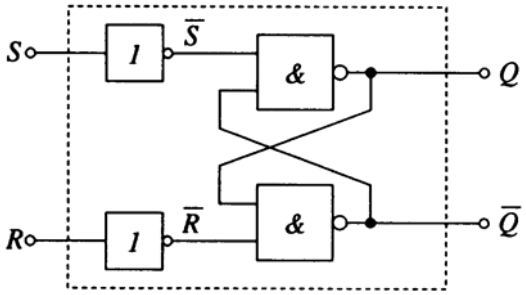
B	A	$\overline{A \wedge B}$
0	0	1
0	1	1
1	0	1
1	1	0

RS-Flipflop aus NAND-Gattern

- Ein RS-Flipflop entsteht durch Negation der Eingänge

Funktionstabelle

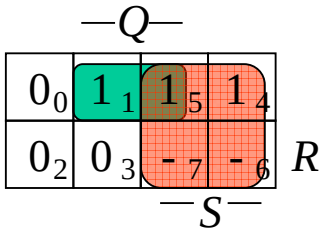
S	R	$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$	
0	0	1	1	(wie vorher) speichern		
0	1	1	0	0	1	
1	0	0	1	1	0	
1	1	0	0	(1	1)	unzulässig



Zustandsfolgetabelle

- Ausgangssignal ändert sich zeitversetzt nach der Signaländerung am Eingang
- Zeitverhalten wird in einer Zustandsfolge dargestellt
  - ⇨ Q_n ist der Wert vor der Signaländerung
  - ⇨ Q_{n+1} ist der Wert nach der Signaländerung

S	R	Q_n	Q_{n+1}	
0	0	0	0	speichern
0	0	1	1	speichern
0	1	0	0	rücksetzen
0	1	1	0	rücksetzen
1	0	0	1	setzen
1	0	1	1	setzen
1	1	0	-	unzulässig
1	1	1	-	unzulässig

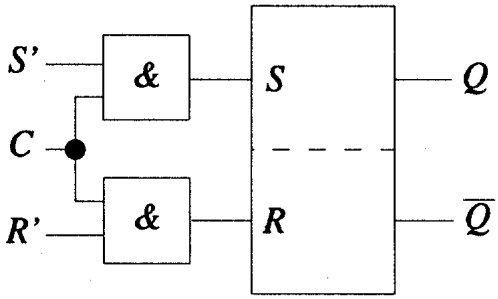


$$Q_{n+1} = S \vee (\overline{R} \wedge Q_n)$$

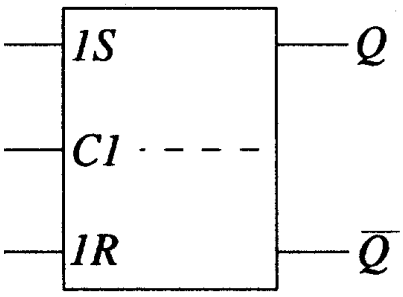
- Diese Gleichung heißt auch Funktionsgleichung oder Übergangsfunktion eines RS-Flipflops
  - ⇨ das Verhalten eines Flipflops kann durch eine Schaltfunktion beschrieben werden

RS-Flipflop mit Zustandssteuerung

- **Beim RS-Flipflop wird der Ausgang sofort nach Anlegen der Eingangssignale gesetzt**
  - ⇨ **zur Vermeidung von Hazards wird häufig gefordert, dass ein Flipflop seinen Wert nur zu bestimmten Zeitpunkten ändert**
  - ⇨ **Synchrone Schaltwerke**
  - ⇨ **Einführung eines Taktsignals**



Schaltung



Schaltzeichen

RS-Flipflop mit Zustandssteuerung

Funktionstabelle

C	S	R	Q_n	Q_{n+1}	
0	0	0	0	0	
0	0	0	1	1	keine Änderung
0	0	1	0	0	des
0	0	1	1	1	Ausgangs-
0	1	0	0	0	zustands
0	1	0	1	1	d.h.
0	1	1	0	0	Speichern
0	1	1	1	1	
1	0	0	0	0	speichern
1	0	0	1	1	speichern
1	0	1	0	0	rücksetzen
1	0	1	1	0	rücksetzen
1	1	0	0	1	setzen
1	1	0	1	1	setzen
1	1	1	0	-	unzulässig
1	1	1	1	-	unzulässig

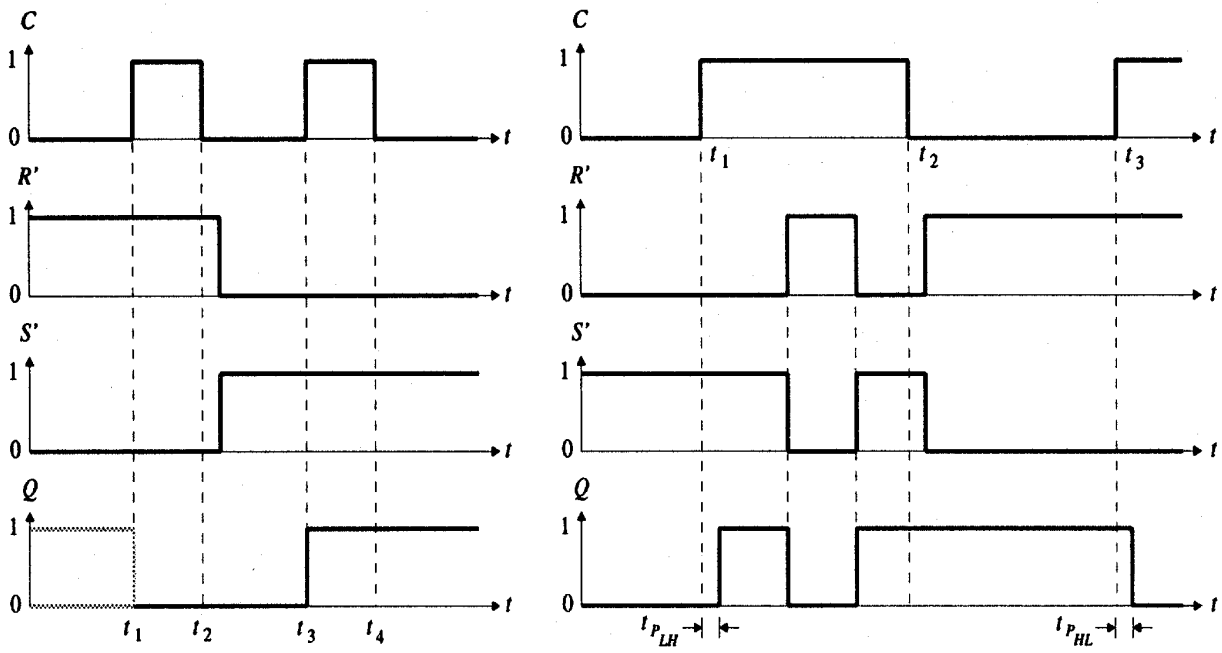
- **Aus der Übergangsfunktion des RS-Flipflops**

$$Q_{n+1} = S \vee (\overline{R} \wedge Q_n)$$

mit $S = (C \wedge S')$ **und** $R = (C \wedge R')$

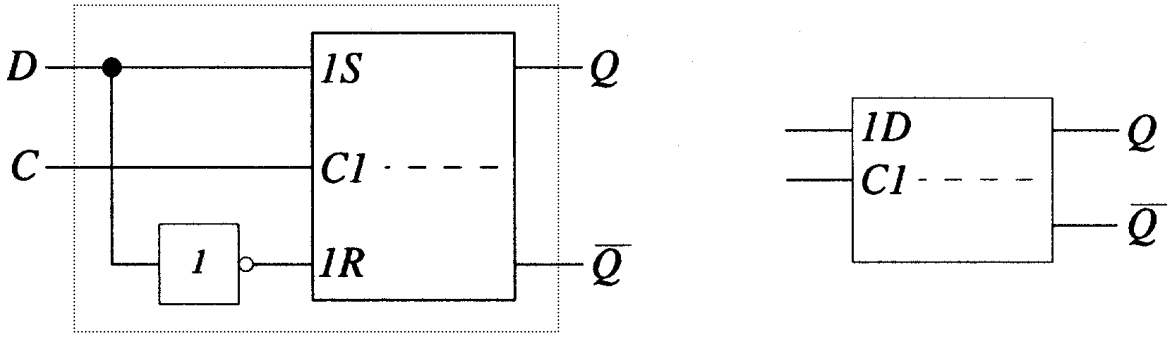
$$Q_{n+1} = (C \wedge S') \vee ((\overline{C \wedge R'}) \wedge Q_n)$$

Impulsdiagramm für Taktzustandssteuerung



D-Flipflop mit Zustandssteuerung

- Das D-Flipflop entsteht aus einem RS-Flipflop mit Zustandssteuerung, durch Negation des Setzsignals S

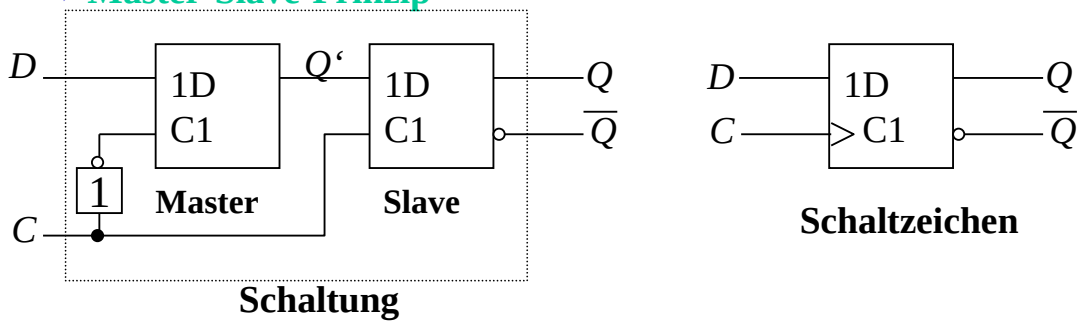


C	D	Q_n	Q_{n+1}	
0	-	0	0	speichern
0	-	1	1	speichern
1	0	-	0	rücksetzen
1	1	-	1	setzen

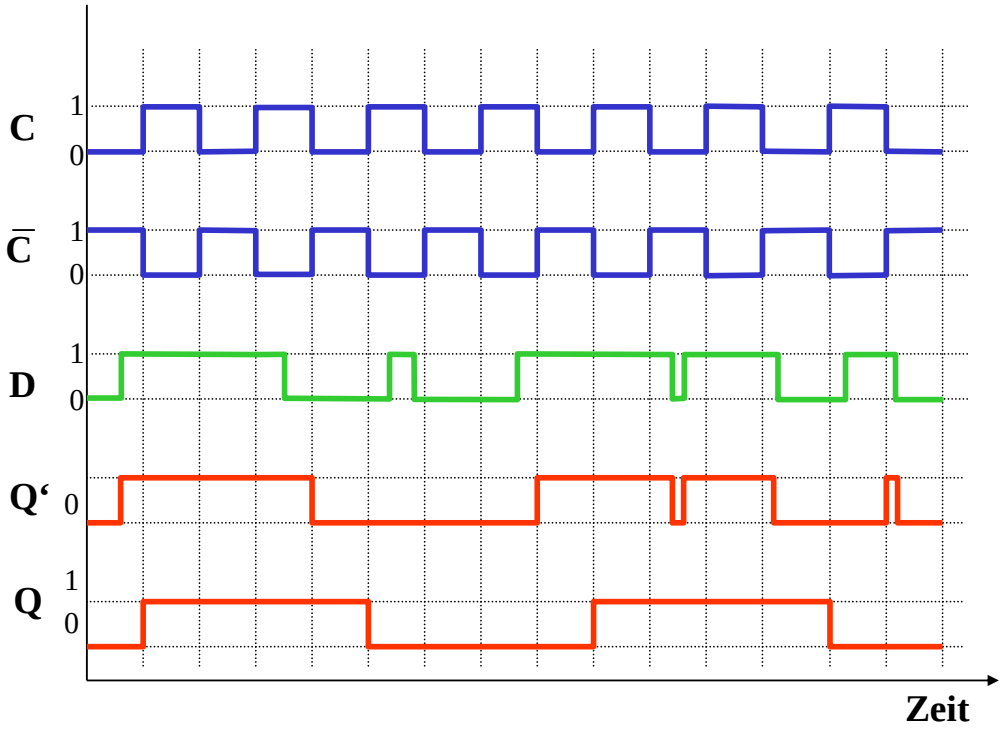
Master-Slave D- Flipflop

- Verketten von Flipflops
  - ⇨ Anwendung: Schieberegister, Zähler
  - ⇨ Problem: bei C=1 „rutschen“ Eingänge bis zum Ausgang durch

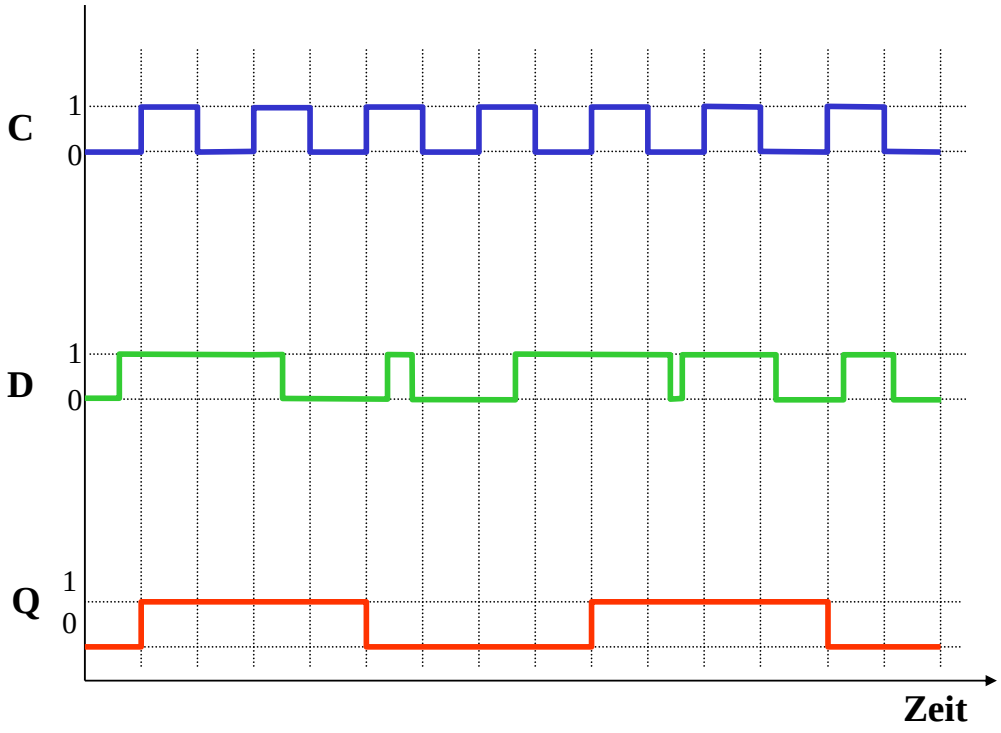
- Lösung: (positiv) flankengesteuertes Flipflop
  - ⇨ zwei D-Flipflops werden hintereinander geschaltet
  - ⇨ das erste Flipflop erhält den negierten Takt
  - ⇨ Master-Slave-Prinzip



Impulsdiagramm des Master-Slave D-Flipflops

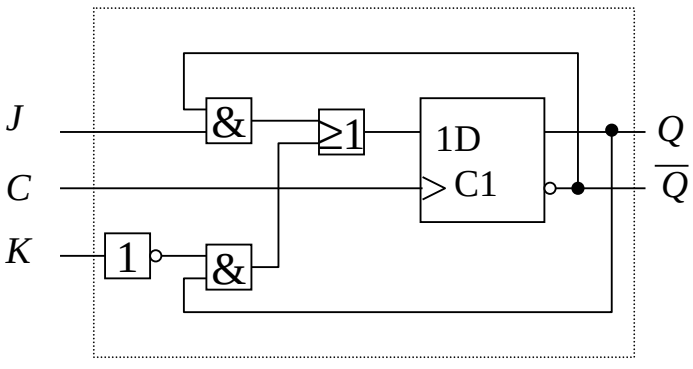


Impulsdiagramm des Master-Slave D-Flipflops

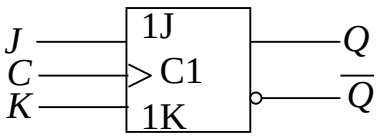


JK-Flipflop

- **Es ist sinnvoll, zu den Funktionen Speichern, Setzen und Rücksetzen eine weitere Funktion Wechseln für die bisher undefinierte Belegung R=S=1 zu definieren**
  - ⇨ **möglich durch Rückführung der Ausgänge an die Eingänge**



Schaltung



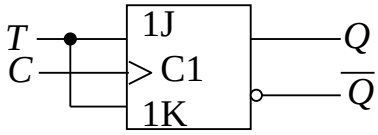
Schaltzeichen

J	K	Q_{n+1}	
0	0	Q_n	speichern
0	1	0	rücksetzen
1	0	1	setzen
1	1	$\overline{Q}_n$	wechseln

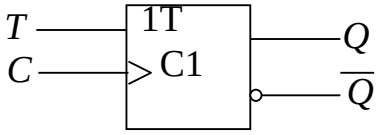
Funktionstabelle

Master-Slave T-Flipflop

- **T-Flipflop** besitzt wie das D-Flipflop nur einen Eingang
  - ⇨ ist dieser gleich 1, wechselt das Flipflop seinen Wert
  - ⇨ T steht für **toggle**



Schaltung



Schaltzeichen

T	Q_{n+1}	
0	Q_n	speichern
1	$\overline{Q}_n$	wechseln

Funktionstabelle

3 Schaltwerke

3.1 Formale Grundlagen

- **Schaltnetze**
  - ⇨ die Ausgabe einer Schaltung hängt nur von den Werten der Eingabe zum gleichen Zeitpunkt ab
  - ⇨ man nennt sie auch **kombinatorische Schaltungen**

- **Schaltwerke**
  - ⇨ die Ausgabe einer Schaltung kann von den Werten der Eingabe zu vergangenen Zeitpunkten abhängen
  - ⇨ alle Abhängigkeiten von Werten der Vergangenheit werden in einem **Zustand** zusammengefaßt
  - ⇨ sind Implementierungen von deterministischen endlichen Automaten

Beschreibung endlicher Automaten

- Andere Namen für endliche Automaten und deren Realisierungen:
  - ⇨ Finite State Machine, FSM
  - ⇨ sequentielle Schaltungen
  - ⇨ Schaltungen mit Speicherverhalten

- Aus der Automatentheorie bekannt:
  Ein endlicher Automat (mit Ausgabe) ist ein Sechstupel $A=(X,Y,S,\delta,\lambda,s_0)$
  - ⇨ endliche Menge von Eingangsbelegungen: X
  - ⇨ endliche Menge von Ausgangsbelegungen: Y
  - ⇨ endliche Menge von Zuständen: S
  - ⇨ Zustandsübergangsfunktion: $\delta : X \times S \rightarrow S$
  - ⇨ Ausgabefunktion: $\lambda : X \times S \rightarrow Y$ (Mealy Verhalten)
    $\lambda : S \rightarrow Y$ (Moore Verhalten)
  - ⇨ Startzustand: s_0

Mealy- und Moore-Automaten

- Die Zustände eines endlichen Automaten werden in Flipflops gespeichert
  - ⇨ möglich sind D-, T-, JK-, RS-Flipflops

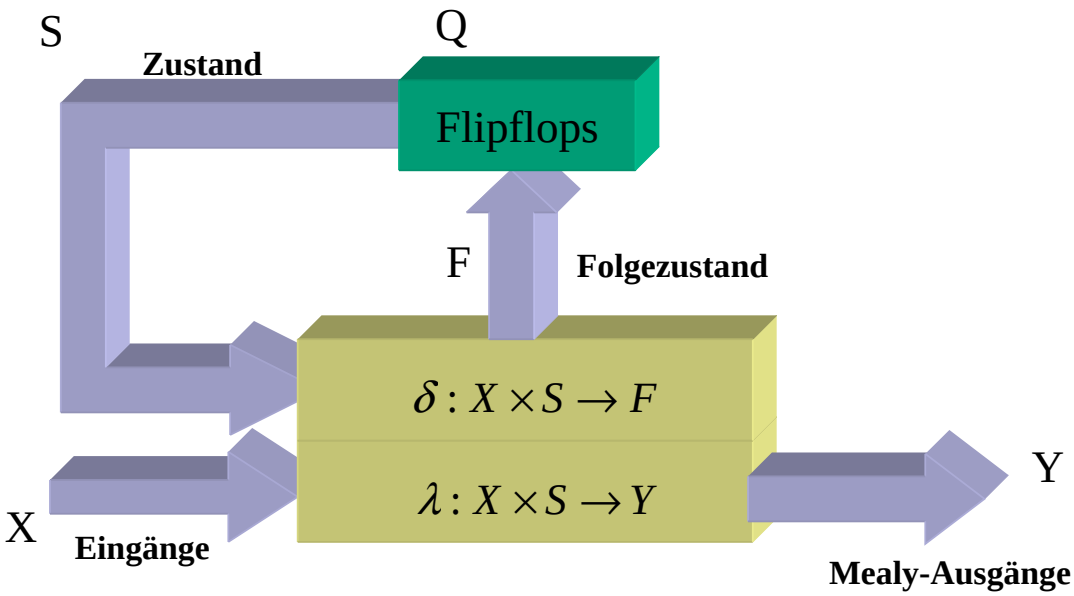
- Aktueller Zustand wird an die Eingänge der Schaltung rückgekoppelt.

- Man unterscheidet:
  Mealy-Automat:
  - ⇨ Ausgangsleitungen können sich ändern, auch wenn keine Taktflanke aufgetreten ist

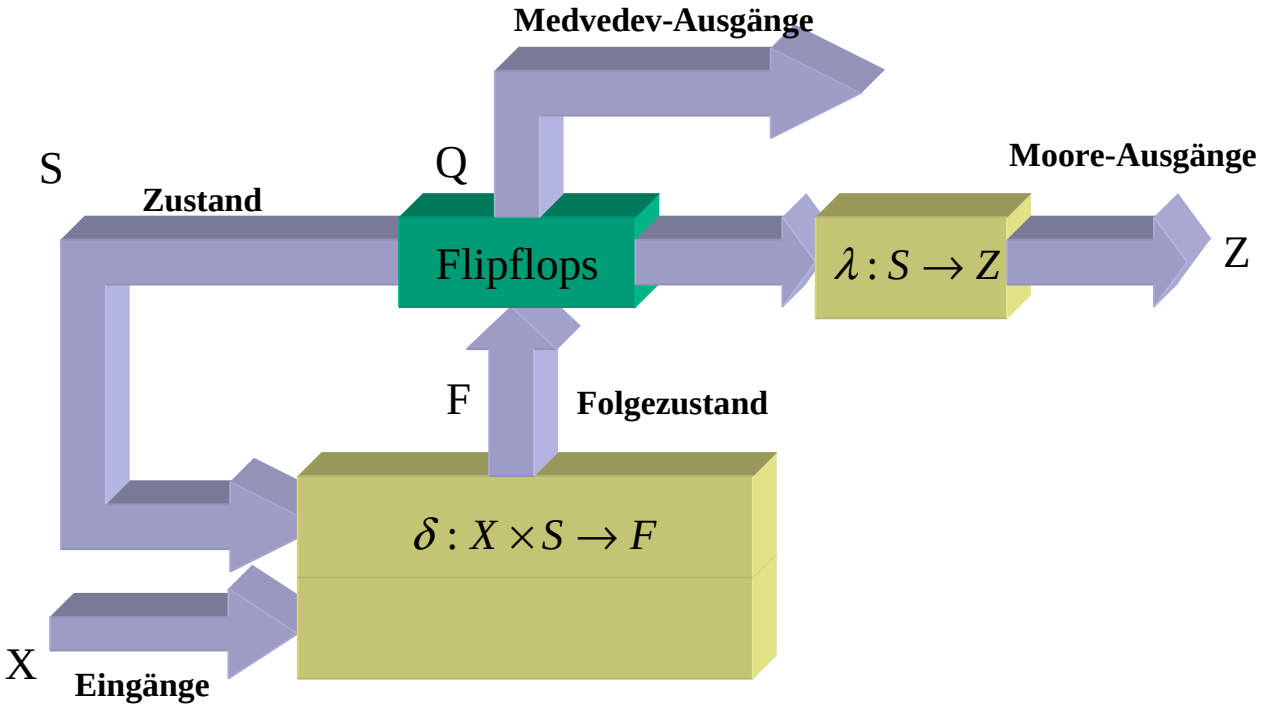
  Moore-Automat:
  - ⇨ Änderung von Ausgangsleitungen nur mit Änderung eines Taktimpulses möglich

  Medvedev-Automat:
  - ⇨ Spezialfall des Moore-Automaten
  - ⇨ die Ausgänge sind die Zustandsbits der Flipflops

Struktur eines Mealy-Automaten



Struktur eines Moore-Automaten



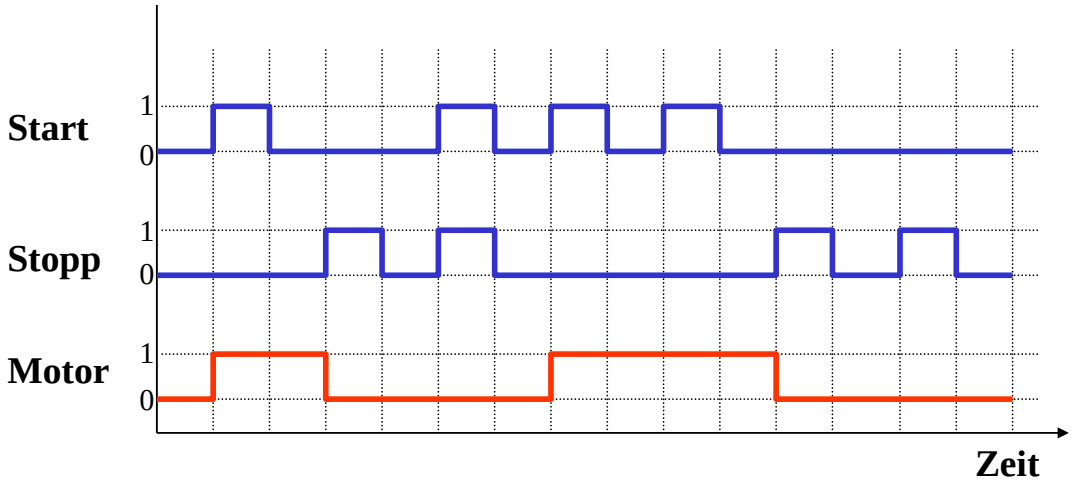
3.2 Darstellung endlicher Automaten

- Aufgabenstellung der Praxis, die durch endlichen Automaten gelöst werden soll, liegt meist in nichtformalisierter Form vor.
- Um beim Entwurf von Schaltwerken systematische und möglichst auch rechnergestützte Entwurfsverfahren einsetzen zu können, wird eine formalisierte Beschreibung benötigt.
- Häufig verwendete Darstellungsformen sind:
  - Zeitdiagramm
  - Automatengraph
  - Ablauftabelle
  - Schaltfunktionen
  - Automatentabelle

Beispiel: Selbsthalteschaltung

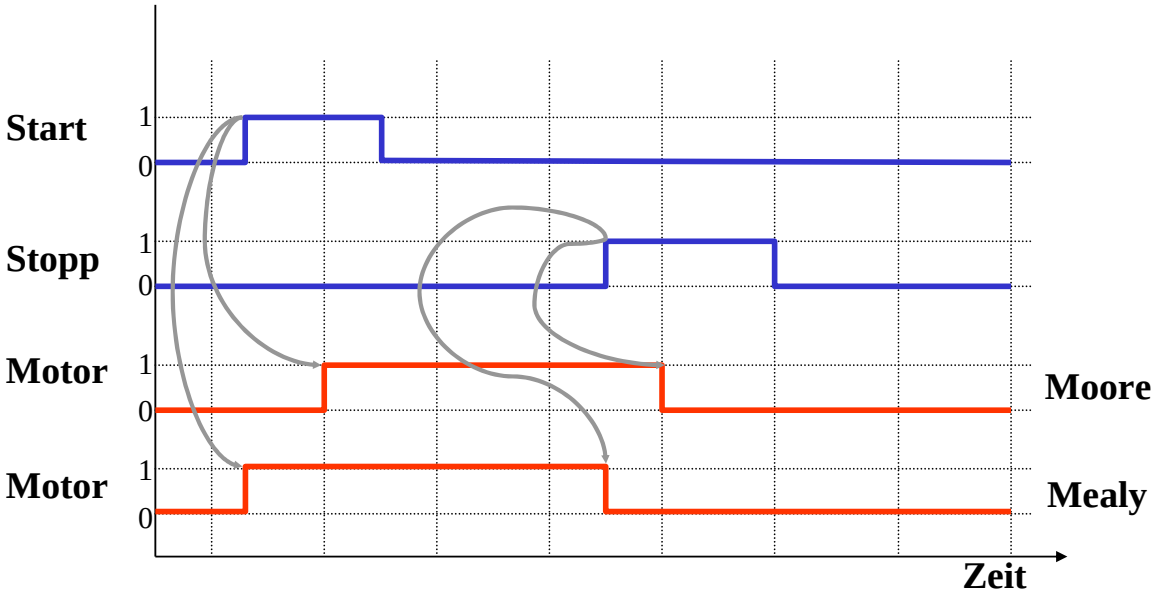
- Funktionsbeschreibung:
  - an den Eingängen befinden sich zwei Tasten : (Start und Stopp)
  - Schaltung liefert ein Ausgangssignal, mit dem ein Gerät ein- oder ausgeschaltet werden kann
  - wird die Starttaste gedrückt, soll das Gerät eingeschaltet werden
  - Gerät soll eingeschaltet bleiben, wenn die Starttaste losgelassen wird
  - Gerät soll ausgeschaltet werden, sobald die Stopptaste betätigt wird
- Zu klären:
  - Was passiert, wenn beide Tasten gleichzeitig betätigt werden?
  - Was passiert, wenn die Starttaste gedrückt wird, obwohl das Gerät eingeschaltet ist?
  - Was passiert, wenn das Gerät ausgeschaltet ist und die Stopptaste gedrückt wird?

Zeitdiagramm

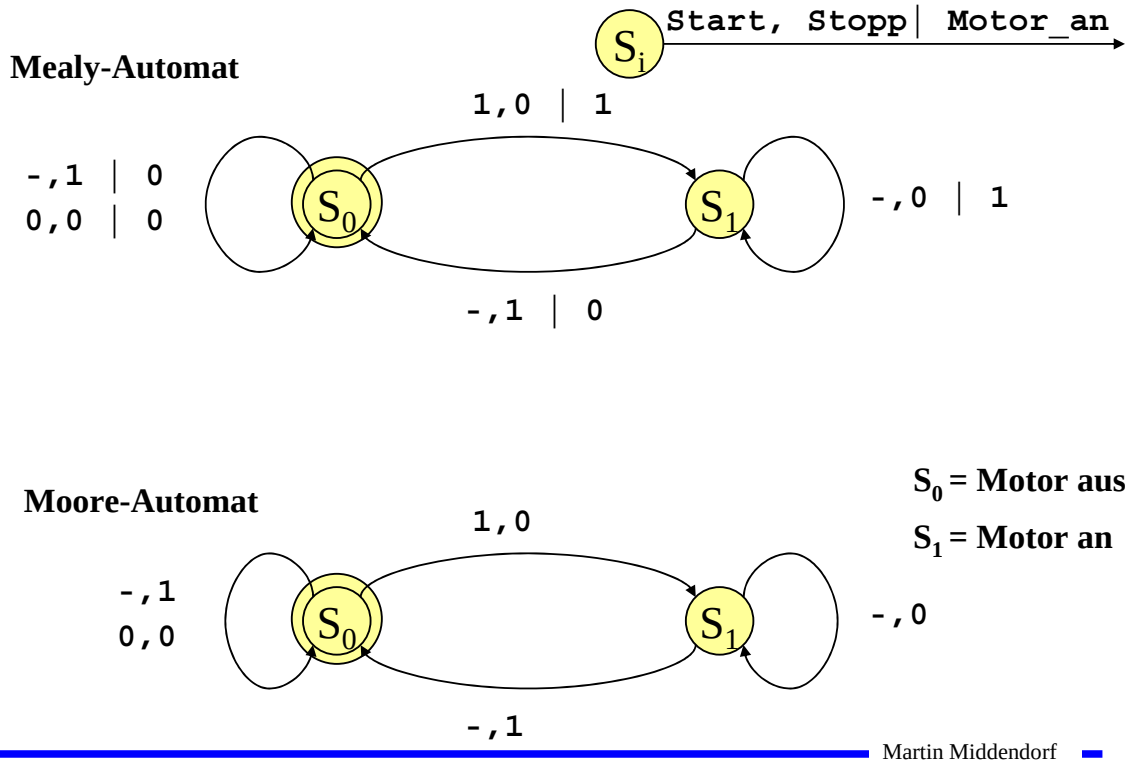


- **Damit lassen sich 2 Zustände festlegen:**
  - ⇨ **Zustand s_0: Ausgabe von** Motor=0 **und warten auf** Start=1 **und** Stopp=0
  - ⇨ **Zustand s_1: Ausgabe von** Motor=1 **und warten auf** Stopp=1

Mealy und Moore Verhalten



Automatengraph



Ablauftabelle

Mealy-Ablauftabelle

Eingang	Zustand	Folgezustand	Ausgang
-, 1	S0	S0	0
0, 0	S0	S0	0
1, 0	S0	S1	1
-, 0	S1	S1	1
-, 1	S1	S0	0

Moore-Ablauftabelle

Eingang	Zustand / Ausgang	Folgezustand
-, 1	S0/ 0	S0
0, 0	S0/ 0	S0
1, 0	S0/ 0	S1
-, 0	S1/ 1	S1
-, 1	S1/ 1	S0

Interpretation der Ablauftabelle

Wenn	**wir im Zustand 0 sind**
und	**zusätzlich Start = 1 und Stop = 0 gilt,**
dann	**wird Motor_an zu 1**
und	**wir gehen mit dem nächsten Takt in den Zustand 1**

Schaltfunktionen

- **Aus der Ablauftabelle lassen sich die die Ausgabe- und die Zustandsübergangsfunktion ablesen:**

x_1, x_2	Zustand S	Folgezustand S^+	Ausgang y
-, 1	S0	S0	0
0, 0	S0	S0	0
1, 0	S0	S1	1
-, 0	S1	S1	1
-, 1	S1	S0	0

- **Übergangsfunktion:**

$$s_0^+ = (x_2 \wedge s_0) \vee (\bar{x}_1 \wedge \bar{x}_2 \wedge s_0) \vee (x_2 \wedge s_1)$$

$$s_1^+ = (x_1 \wedge \bar{x}_2 \wedge s_0) \vee (\bar{x}_2 \wedge s_1)$$

- **Ausgabefunktion:**

$$y = (x_1 \wedge \bar{x}_2 \wedge s_0) \vee (\bar{x}_2 \wedge s_1)$$ **Mealy-Automat**

$$y = s_1$$ **Moore-Automat**

Automatentabelle

Zustand	Folgezustand Start/Stopp 0/0	0/1	1/0	1/1	Ausgang
s_0	s_0	s_0	s_1	s_0	0
s_1	s_1	s_0	s_1	s_0	1

Moore-Automat

Zustand	Folgezustand/ Ausgang Start/Stopp 0/0	0/1	1/0	1/1
s_0	$s_0/0$	$s_0/0$	$s_1/1$	$s_0/0$
s_1	$s_1/1$	$s_0/0$	$s_1/1$	$s_0/0$

Mealy-Automat

- In der Automatentabelle werden die Zustände senkrecht und alle möglichen Eingangsbelegungen waagerecht dargestellt
  - ⇨ an Schnittpunkten werden die Folgezustände eingetragen
  - ⇨ Moore-Automat: Ausgabe wird dem Zustand zugeordnet
  - ⇨ Mealy-Automat: Ausgabe wird dem Folgezustand zugeordnet

Medvedev- und Moore-Automaten

- Auch Moore-Automaten können während des Übergangs Fehlimpulse (Glitches, Hazards) auslösen
  - ⇨ unterschiedliche Laufzeiten in der Schaltung
  - ⇨ 01 nach 10 Übergänge der Zustandsübergangsfunktion ohne Änderung des Ausgangswertes
- Medvedev-Automaten besitzen am Ausgang ein Flipflop
  - ⇨ keine Fehlimpulse

3.3 Analyse und Entwurf von Schaltwerken

Grundlegende Realisierung von Automaten

- **Asynchrone Realisierung**
  - ⇨ **Zustandsspeicher durch Rückkopplung**
  - ⇨ **kein zentraler Takt**
  - ⇨ **Koordination durch zusätzliche Bereitstellungs- und Quittungssignale**
  - ⇨ **die Zustandsspeicher (Flipflops) können zu jedem Zeitpunkt ihren Wert ändern**
  - ⇨ **self-timed**

- **Synchrone Realisierung**
  - ⇨ **Rückkopplung nur durch flanken- oder pegelgetriggerte Flipflops**
  - ⇨ **die Taktleitungen aller Flipflops sind miteinander verbunden (oder hängen nach einem festen Zeitschema voneinander ab)**

- **Trotz gewisser praktischer Bedeutung asynchroner Realisierungen werden hier nur synchrone Realisierungen betrachtet.**

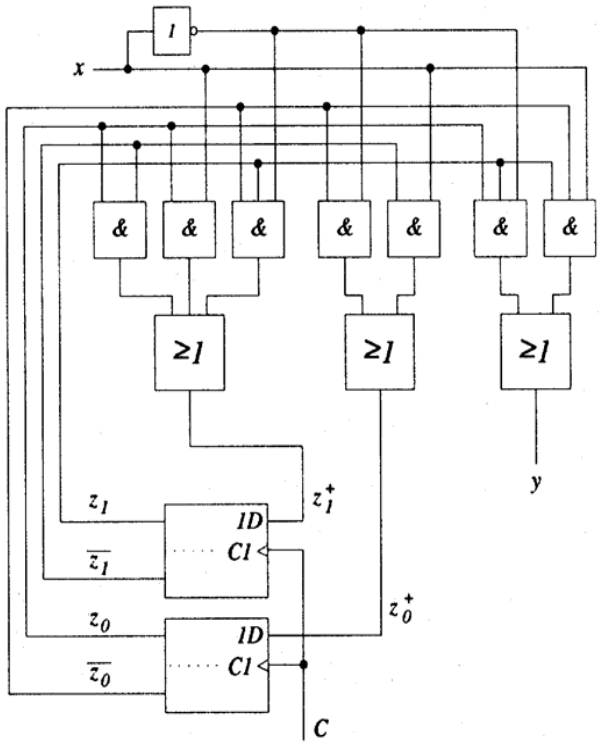
3.3.1 Analyse von Schaltwerken

- **Schaltwerkanalyse: Darstellung des Schaltverhalten durch**
  - ⇨ **Zustandstabelle**
  - ⇨ **Schaltfunktion**
  - ⇨ **Zustandsgraph**

- **Prinzipielles Vorgehen:**
  - ⇨ **von gegebenem Schaltplan werden zunächst die Ausgabe und Übergangsfunktion abgeleitet**
  - ⇨ **ein Anfangszustand wird angenommen**
  - ⇨ **mit den Werten der Eingangsvariablen werden die Folgezustände abgeleitet**
  - ⇨ **auf diese Weise entstehen die Ablauftabellen**
  - ⇨ **aus den Ablauftabellen kann der Automatengraph abgeleitet werden**

Beispiel: Ausgangspunkt - der Schaltplan

- Grundlegende Charakterisierungen
  - ⇨ synchrones Schaltwerk
  - ⇨ Eingang *x* und Ausgang *y* bestehen je aus einer Variablen
  - ⇨ das Schaltwerk enthält 2 D-Flipflops
  - ⇨ es kann maximal 4 Zustände besitzen
  - ⇨ Schaltwerk ist ein Mealy-Automat



Die Schaltfunktion

- Aus dem Schaltplan läßt sich ablesen:
  - ⇨ für die Übergangsfunktion

$$z_0^+ = (\overline{z}_0 \wedge \overline{x}) \vee (\overline{z}_1 \wedge x)$$

$$z_1^+ = (z_0 \wedge \overline{z}_1) \vee (z_0 \wedge x) \vee (\overline{z}_0 \wedge z_1 \wedge \overline{x})$$

  - ⇨ für die Ausgabefunktion

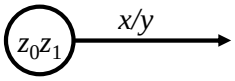
$$y = (z_0 \wedge z_1 \wedge \overline{x}) \vee (\overline{z}_0 \wedge z_1 \wedge x)$$

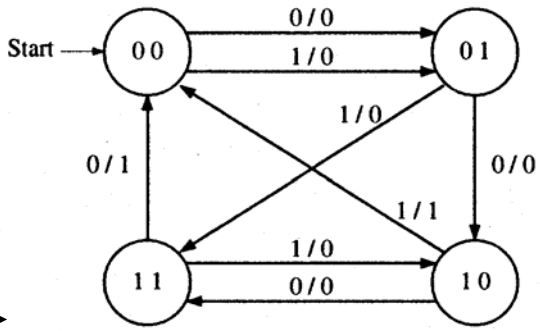
Die Ablauftabelle und der Automatengraph

- Aufstellen der Ablauftabelle über die Auswertung der Funktionen für z_0, z_1 und y
  - ⇨ alle Belegungen der Eingangsvariablen
  - ⇨ alle Belegungen der Zustandsvariablen

z_1	z_0	x	z_1^+	z_0^+	y
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	1	0
1	0	1	0	0	1
1	1	0	0	0	1
1	1	1	1	0	0

- Aufstellen des Automatengraphen über die Auswertung der Ablauftabelle
  - ⇨ Beschriftung der Zustände und Übergänge nicht vergessen!



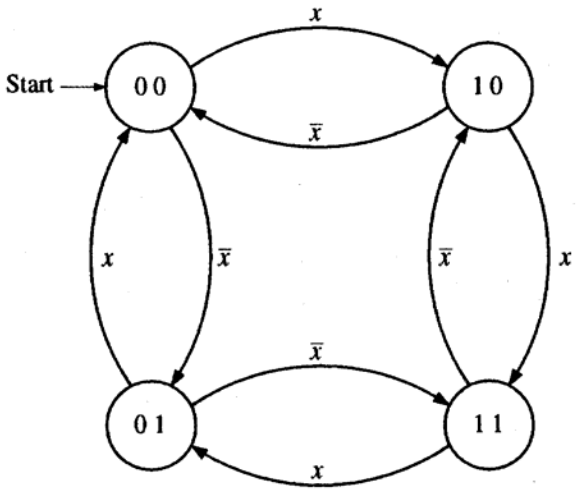


3.3.2 Entwurf von Schaltwerken

- Prinzipielles Vorgehen:
  - ⇨ Festlegung der Zustandsmenge
    - daraus ergibt sich die Anzahl der erforderlichen Speicherglieder
  - ⇨ Festlegung des Anfangszustands
  - ⇨ Definition der Ein- und Ausgangsvariablen
  - ⇨ Darstellung der zeitlichen Zustandsfolge in Form eines Zustandsgraphen
  - ⇨ Aufstellung der Ablauftabelle
  - ⇨ Herleitung der Übergangs- und Ausgabefunktionen
  - ⇨ Darstellung der Übergangs- und Ausgabefunktionen in einem KV-Diagramm und Minimierung
  - ⇨ Darstellung des Schaltwerks in einem Schaltplan

Beispiel: ein umschaltbarer Zähler

- Es soll ein zweistelliger Gray-Code-Zähler entworfen werden, der vorwärts und rückwärts zählen kann
- Umschaltung der Zählrichtung erfolgt über Eingangsvariable x
  - ⇨ für x=0 ist die Zählfolge
    `00 - 01 - 11 - 10`
  - ⇨ für x=1 ist die Zählfolge
    `00 - 10 - 11 – 01`
- Die Ausgangsvariablen sind identisch mit den Zustandsvariablen, da der Zählerstand angezeigt werden soll
  - ⇨ Moore-Automat



Automatengraph

Ablauftabelle und die Übergangsfunktionen

- Die Ablauftabelle kann direkt aus dem Automatengraph abgeleitet werden
  - ⇨ die linke Seite enthält alle Wertekombinationen, die z_0, z_1 und x einnehmen können
  - ⇨ die rechte Seite enthält die Werte der Folgezustände

z_1	z_0	x	z_1^+	z_0^+
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	0
1	1	1	0	1

- Aus der Ablauftabelle können die KV-Diagramme für z_0 und z_1 aufgestellt werden
- Aus den KV-Diagrammen lassen sich die minimierten Übergangsfunktionen ablesen

z_1^+ — x —

0_0	1_1	1_5	0_4
1_2	0_3	0_7	1_6

z_0 — z_1 —

z_0^+ — x —

1_0	0_1	1_5	0_4
1_2	0_3	1_7	0_6

z_0 — z_1 —

$$z_1^+ = (\bar{z}_0 \wedge x) \vee (z_0 \wedge \bar{x})$$

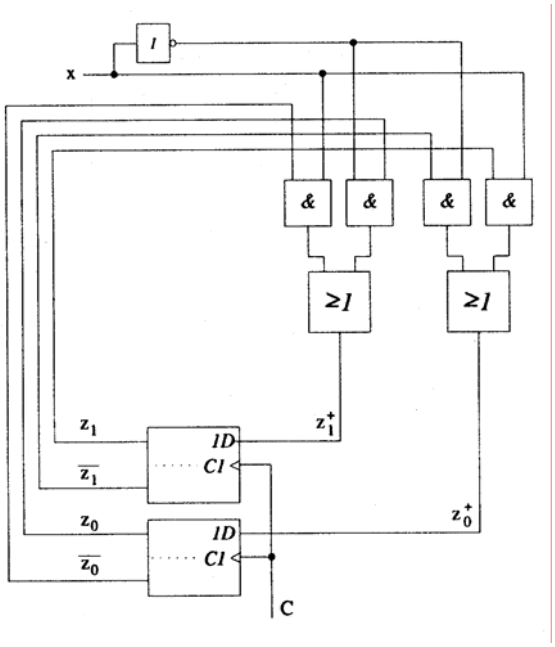
$$z_0^+ = (\bar{z}_1 \wedge \bar{x}) \vee (z_1 \wedge x)$$

Das Schaltwerk

- Die minimierten Übergangsfunktionen können schließlich in einem Schaltplan gezeichnet werden

$$z_0^+ = (\bar{z}_0 \wedge x) \vee (z_0 \wedge \bar{x})$$

$$z_1^+ = (\bar{z}_1 \wedge \bar{x}) \vee (z_1 \wedge x)$$



3.4 Technische Realisierung von Schaltwerken

- Realisierung mit diskreten Bauelementen
  - ⇨ Verknüpfungsglieder
  - ⇨ Speicherglieder

Die Bauelemente werden durch eine feste Verdrahtung miteinander verbunden.

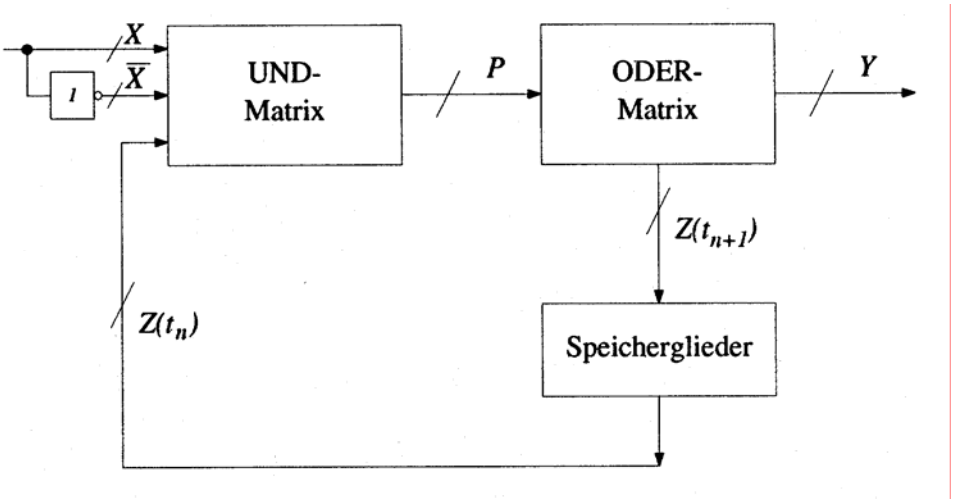
Solche Schaltwerksrealisierungen können nur eine feste Aufgabe erfüllen

- ⇨ das Schaltwerk ist nicht flexibel
- ⇨ bei einem Fehler in der Verdrahtung kann keine Korrektur vorgenommen werden

Die Bauelemente stehen als integrierte Schaltkreise zur Verfügung.

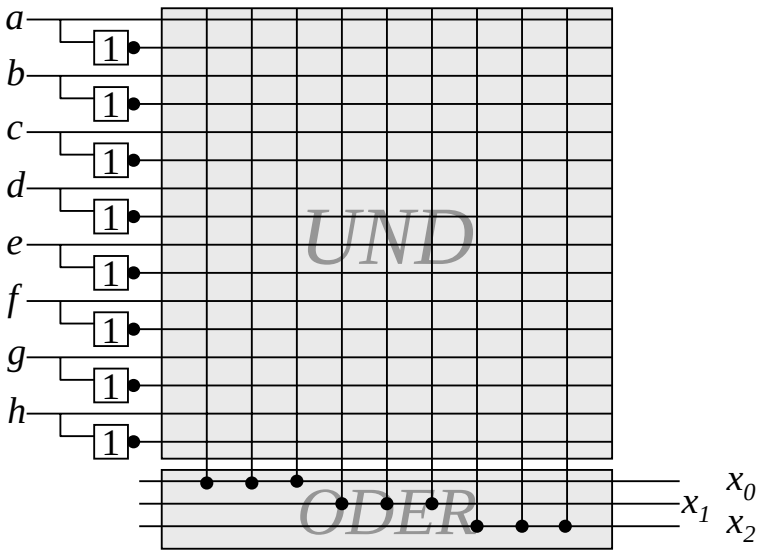
Realisierung mit einem PLA

- Programmable Logic Array (PLA)
  - ⇨ technische Realisierung der DMF
  - ⇨ UND- und ODER-Matrix sind frei programmierbar



Realisierung mit einem PAL

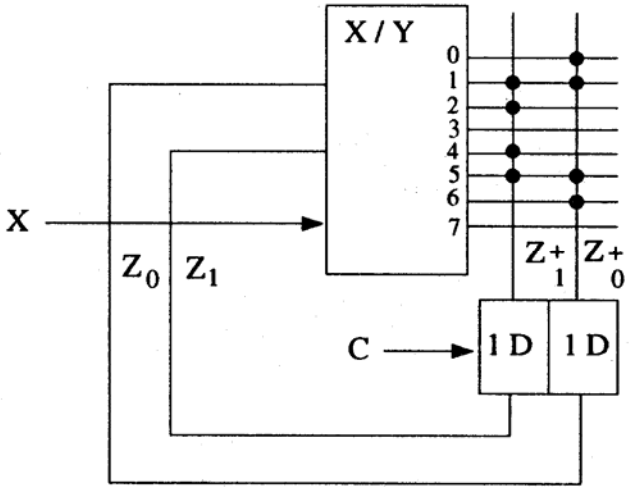
- Programmable Array Logic (PAL)
  - ⇨ die ODER-Matrix ist vorgegeben
  - ⇨ es steht eine feste Anzahl von Implikanten pro Ausgang zur Verfügung
  - ⇨ die UND-Matrix ist programmierbar



Realisierung mit einem ROM

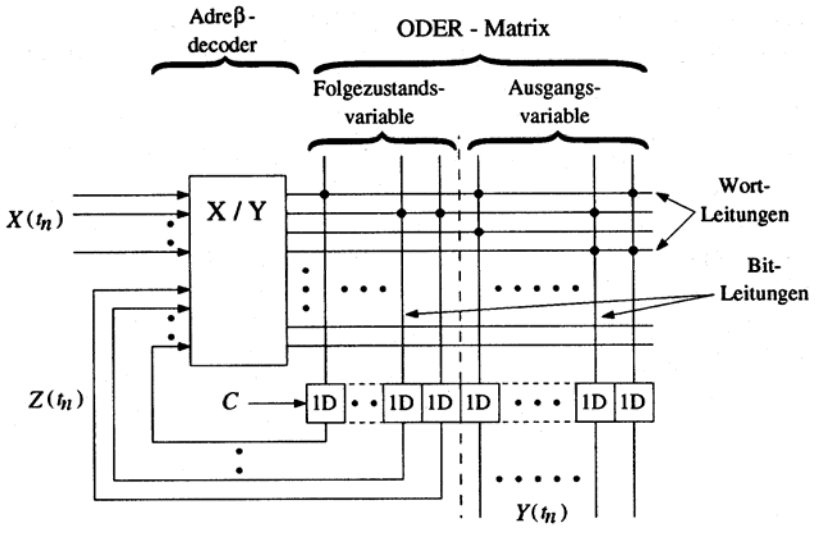
- Technische Realisierung durch ein PROM, EPROM, EEPROM
  - ⇨ Die UND-Matrix ist durch den Adressdekodierer vorgegeben
  - ⇨ alle Minterme sind implementiert
  - ⇨ direkte Implementierung der Funktionstabelle

X	Z_1	Z_0	Z_1^+	Z_0^+
0	0	0	0	1
0	0	1	1	1
0	1	1	1	0
0	1	0	0	0
1	0	0	1	0
1	1	0	1	1
1	1	1	0	1
1	0	1	0	0



Realisierung mit einem ROM

- Auch die Ausgabefunktion kann mit einem ROM realisiert werden
  - ⇨ Wortorientierung des ROMs wird ausgenutzt
  - ⇨ Mikroprogramm
  - ⇨ mögliche Implementierung des Steuerwerks in Mikroprozessoren



4. Spezielle Schaltnetze und Schaltwerke

- Für die Implementierung komplexer Schaltungen werden bestimmte Bausteintypen häufig verwendet:

Schaltnetze

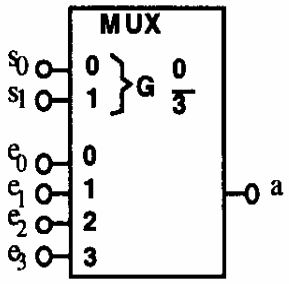
- ⇨ Multiplexer/Demultiplexer
- ⇨ Vergleicher
- ⇨ Addierer
- ⇨ Multiplizierer

Schaltwerke

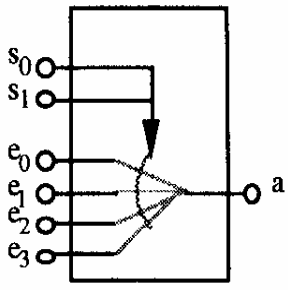
- ⇨ Register
- ⇨ Schieberegister
- ⇨ Zähler

Multiplexer

- Von mehreren Eingängen wird einer als Ausgang durchgeschaltet
- Die Auswahl aus 2^n Eingängen geschieht über n Steuerleitungen



Steuerleitungen s_1	Steuerleitungen s_0	Ausgang a
0	0	e_0
0	1	e_1
1	0	e_2
1	1	e_3

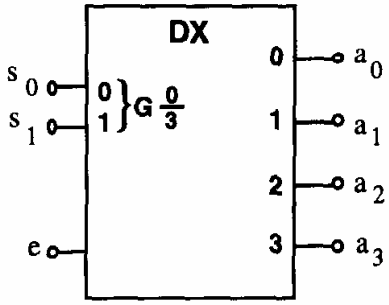


Schaltbild und logisches Verhalten eines 1-aus-4-Multiplexers

Bezeichnung: 4:1-MUX, allgemein 2^n:1-MUX

Demultiplexer

- Ein Eingang wird auf einen von 2^n Ausgängen durchgeschaltet



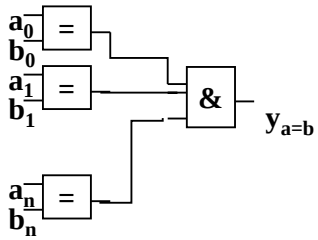
s_1	s_0	a_0	a_1	a_2	a_3
0	0	e	0	0	0
0	1	0	e	0	0
1	0	0	0	e	0
1	1	0	0	0	e

Schaltbild und logisches Verhalten eines 1-auf-4-Demultiplexers

Bezeichnung: 1:4-DX, allgemein 1:2^n-DX

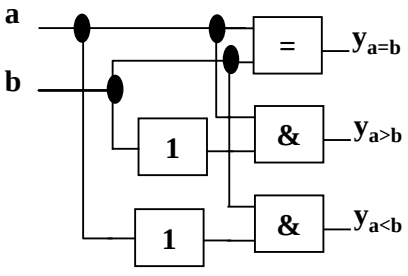
Vergleicher (Komparatoren)

- Vergleich zweier Zahlen
  - ⇨ A=B, A<B, A>B
- Gleichheit bedeutet, dass alle Bits übereinstimmen



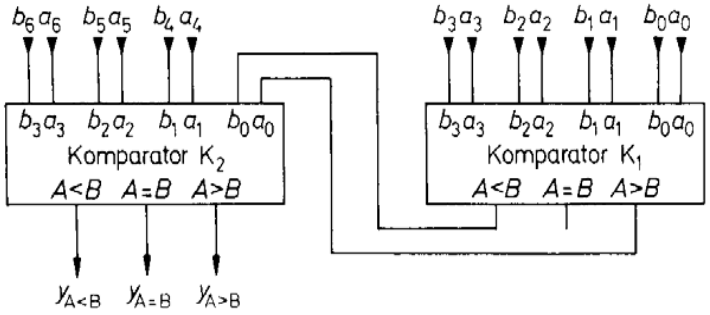
- 1-Bit Komparator mit Größenvergleich

a	b	$y_{a>b}$	$y_{a=b}$	$y_{a<b}$
0	0	0	1	0
0	1	0	0	1
1	0	1	0	0
1	1	0	1	0

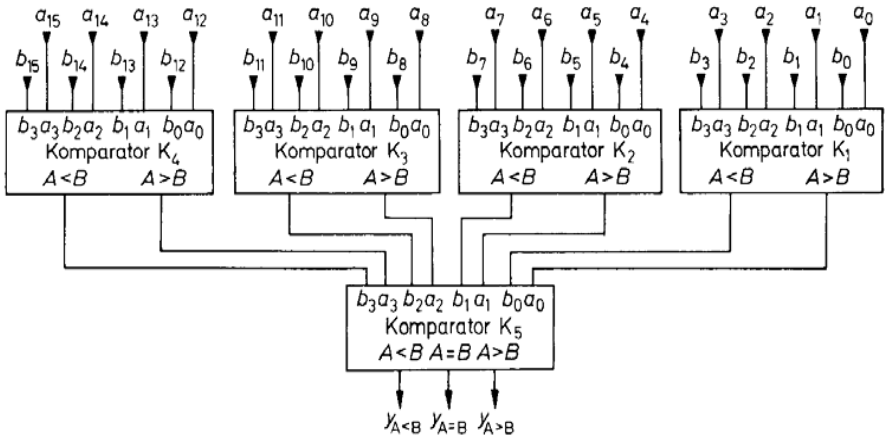


Komparatoren

- **Serielle Erweiterung**



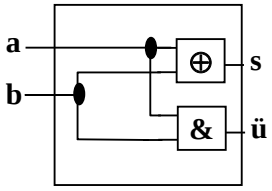
- **Parallele Erweiterung**



Addierer

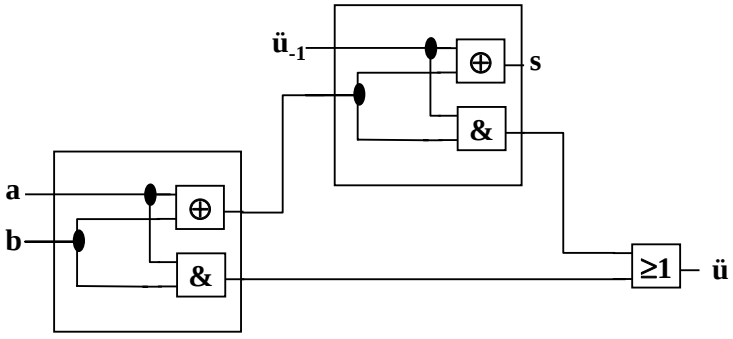
- **Halbaddierer**

a	b	s	ü
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



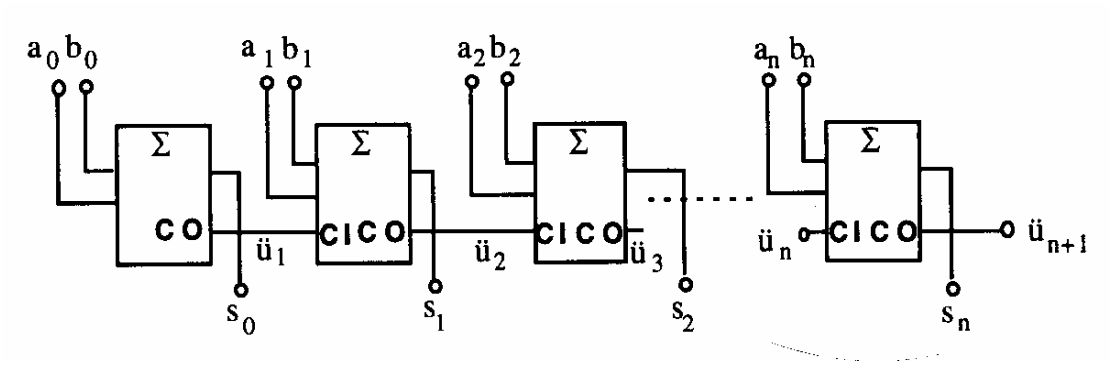
- **Volladdierer**

a	b	$ü_{-1}$	s	ü
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



Addition mit seriellem Übertrag

- Der Übertrag des Volladdierers $ü_i$ wird mit c_{i+1} verbunden

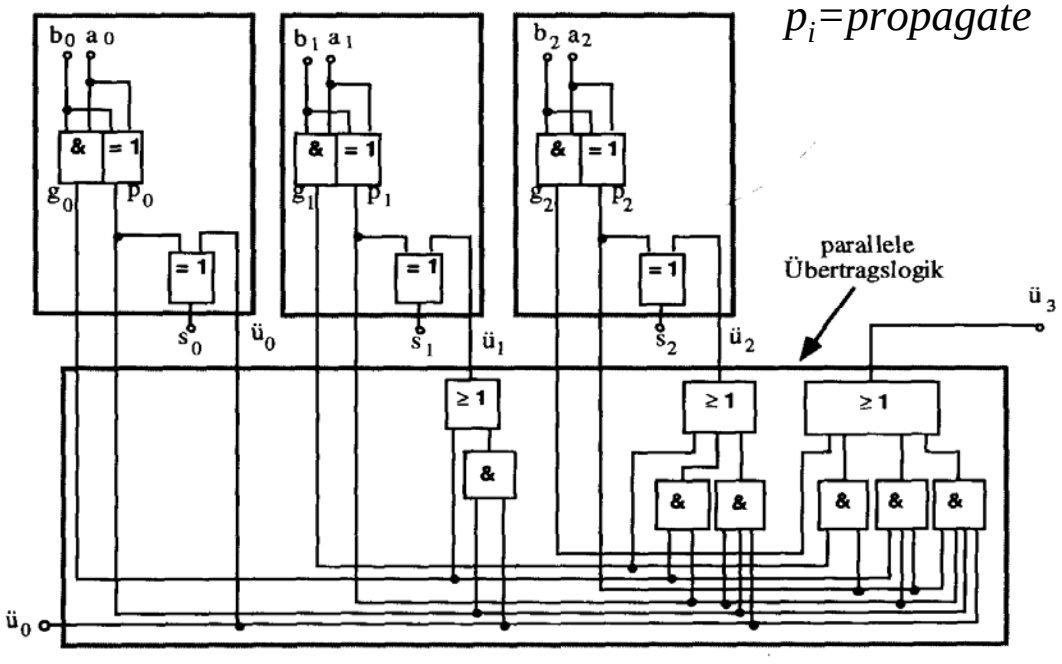


CO=Carry out CI=Carry in

Addierer mit paralleler Übertragslogik

- **Allgemein:** $c_i = a_i b_i \vee (a_i \oplus b_i) c_{i-1}$

g_i=*generate*
p_i=*propagate*

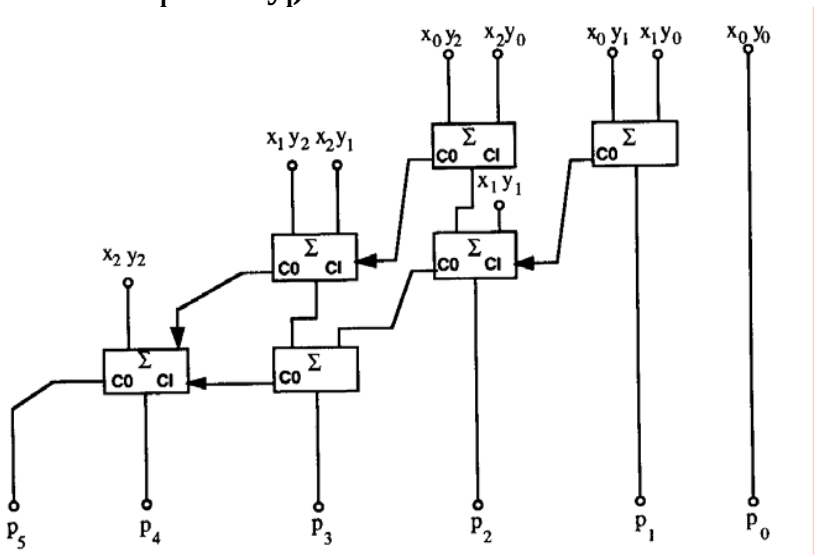


Multiplizierer

- **Parallele Multiplikation durch Addierwerk**

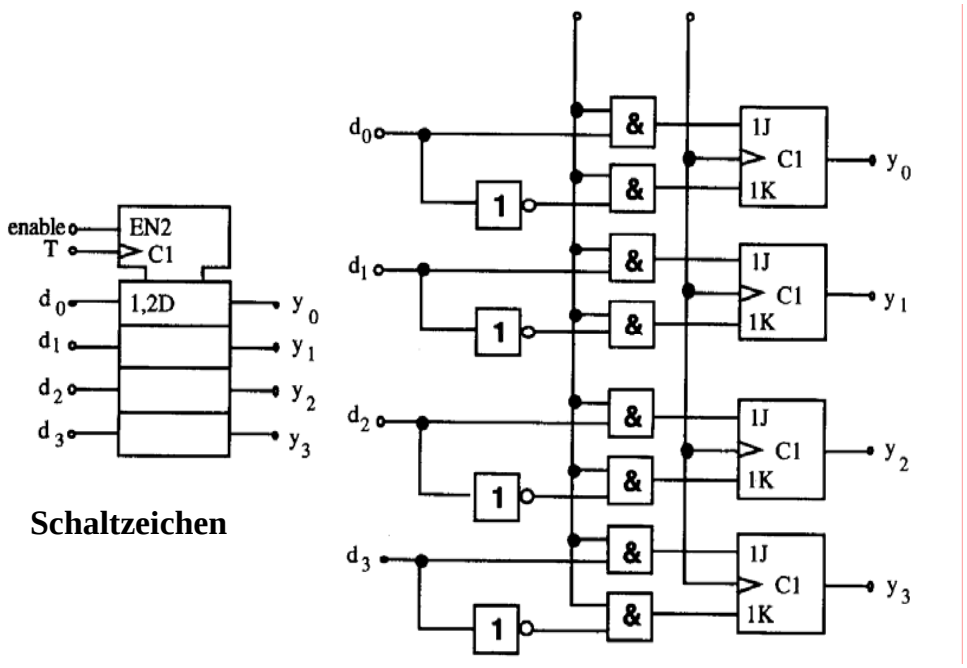
$$p = x \cdot y = \left(\sum_{i=0}^{n-1} x_i \cdot 2^i \right) \cdot \left(\sum_{j=0}^{n-1} y_j \cdot 2^j \right) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \cdot 2^{i+j} x_i y_j$$

- **für n= 3 ($x_i y_j$ steht für x_i UND y_i):**



Register

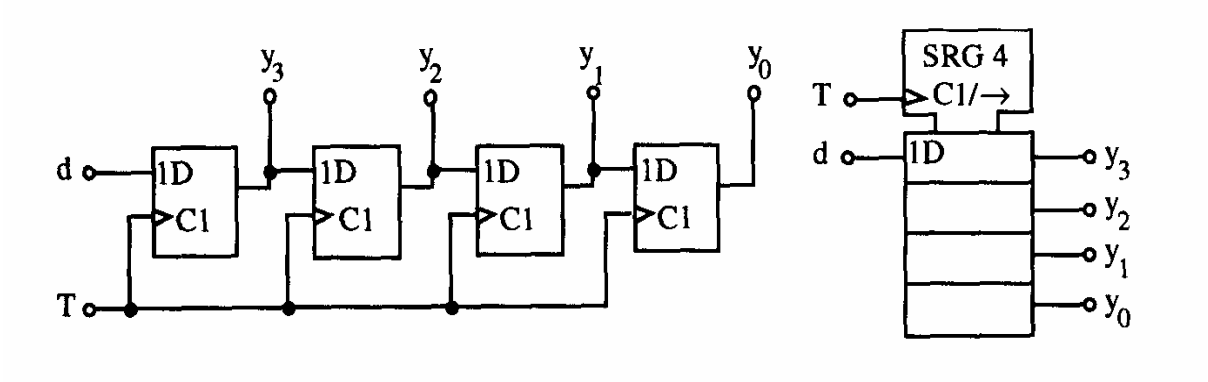
- **Speicherung einer n-stelligen Zahl durch n Flipflops**



Schaltzeichen

Schieberegister

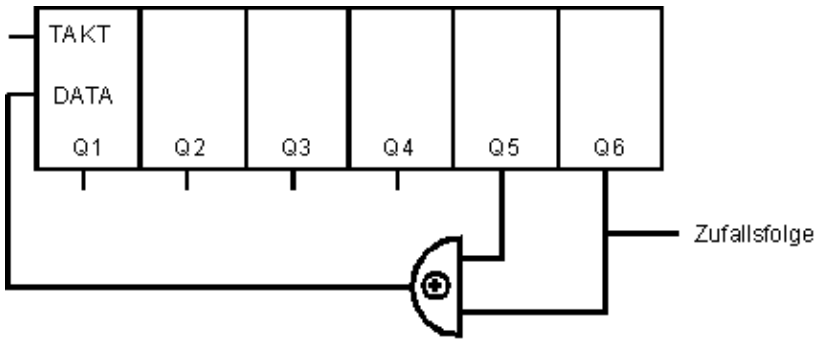
- Kette von Flipflops
- Anwendungen:
  - ⇨ Serien-Parallel-Wandlung und Parallel-Serien-Wandlung
  - ⇨ FIFO oder Stapel-Speicher
  - ⇨ Multiplikation mit 2 oder Division durch 2



Schieberegister

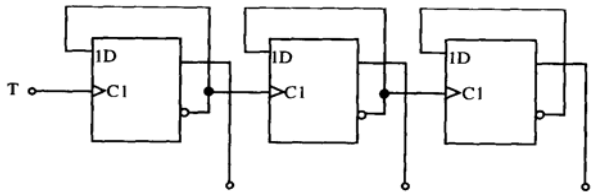
- Rückkopplung zur Erzeugung komplexer Signalfolgen (Sequenzer).

Beispiel: Pseudozufallszahlengenerator

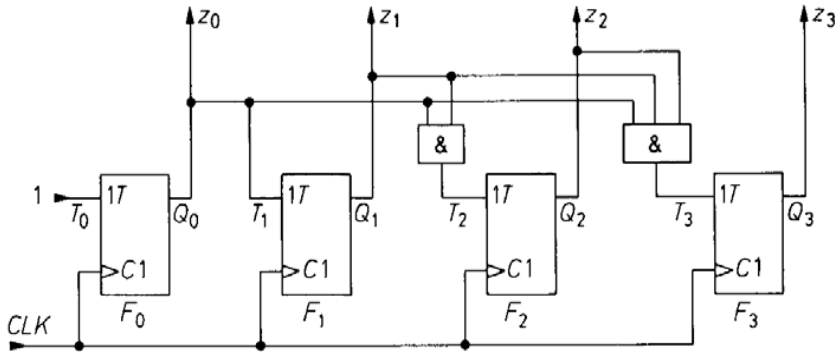


Zähler

- Einfacher Dualzähler durch Rückkopplung
- Asynchroner Ripple Carry Zähler

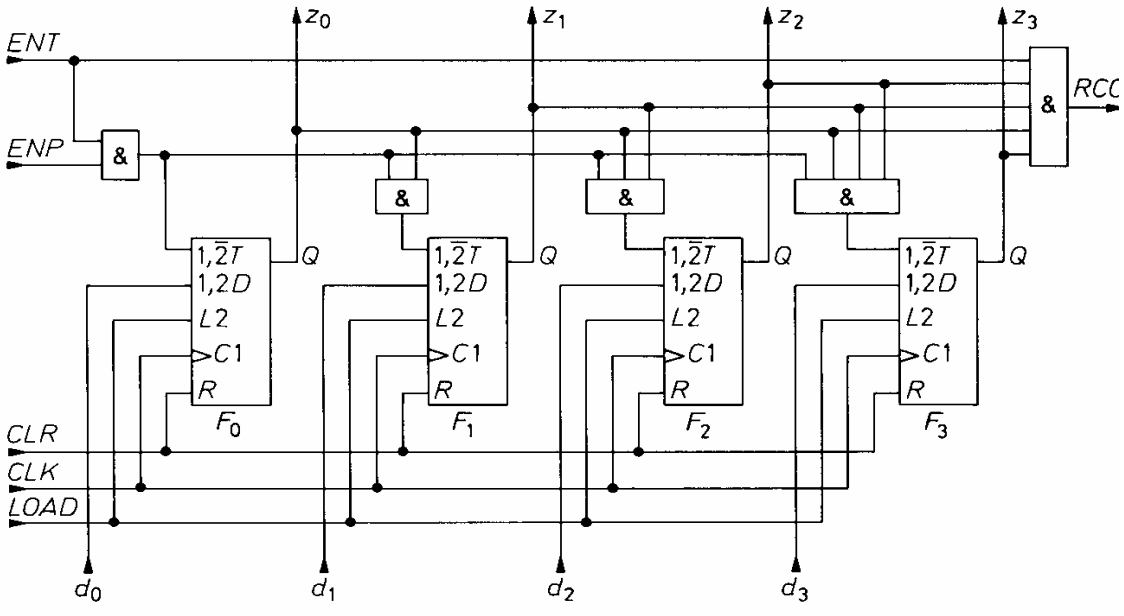


- Synchroner Dualzähler durch Carry-Look-Ahead-Logik



Zähler

- Praktische Ausführung eines Zählers



Zähler

- **Bezeichnungen:**

ENP: Pause Zähler

ENT: Freigabe Zähler und Übertrag

LOAD: Laden, d.h. Einlesen der Werte an d_i

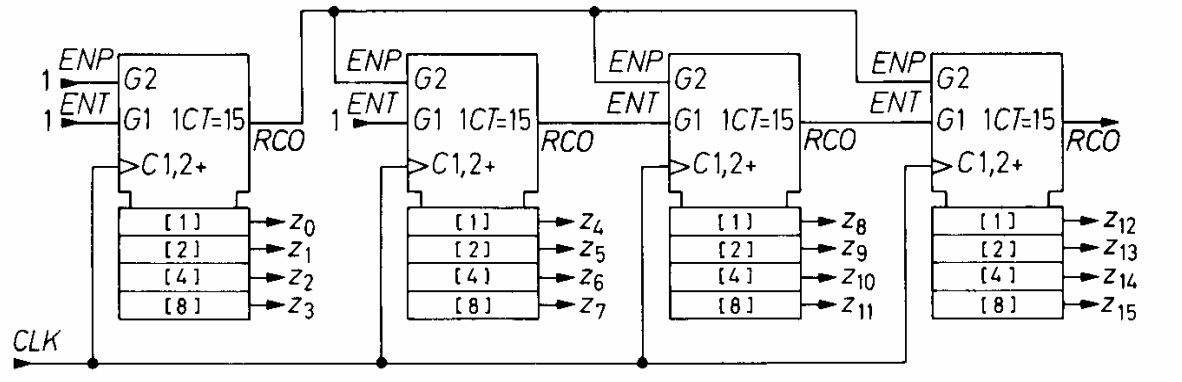
CLR: Nullsetzen

1,2D besagt, dass dieser Eingang abhängig von den Eingängen C1 und L2 ist.

R besagt das Rücksetzen asynchron also unabhängig von C1 erfolgt.

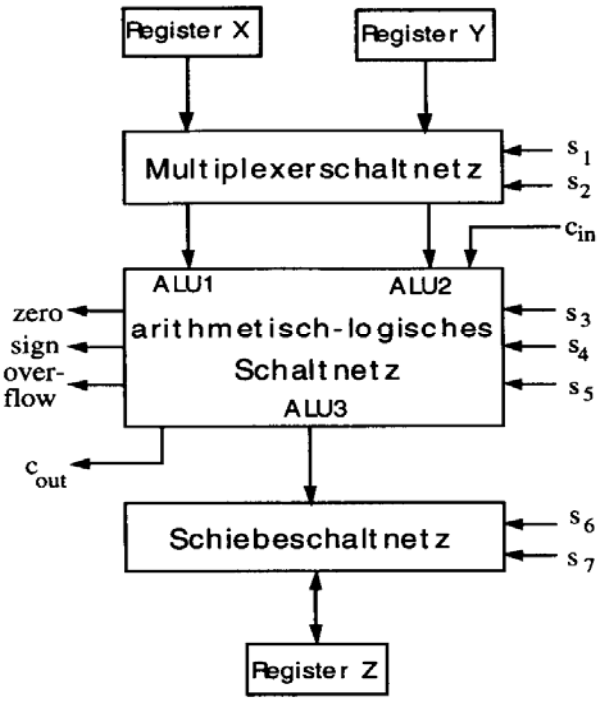
Zähler

- **Kaskadierung eines Zählers**



Bezeichnung: 1CT=15 ist 1, wenn Eingang G1auf 1 ist und Zählerstand 15 ist
RCO=Ripple Carry Out

Aufbau einer ALU



s_1	s_2	ALU1	ALU2
0	0	X	Y
0	1	X	0
1	0	Y	0
1	1	Y	X

s_3	s_4	s_5	ALU3
0	0	0	ALU1+ALU2+c_{in}
0	0	1	ALU1-ALU2-$\overline{c_{in}}$
0	1	0	ALU2-ALU1-$\overline{c_{in}}$
0	1	1	ALU1 $\vee$ ALU2
1	0	0	ALU1 $\wedge$ ALU2
1	0	1	$\overline{\text{ALU1}}$ $\wedge$ ALU2
1	1	0	ALU1 $\not\leftrightarrow$ ALU2
1	1	1	ALU1 $\leftrightarrow$ ALU2

s_6	s_7	Z
0	0	ALU3
0	1	ALU3 / 2
1	0	ALU3 * 2
1	1	Z speichern

Bauelemente eines Rechnersystems

- Multiplexer und Demultiplexer zur Steuerung des Datenflusses
- Zähler für die Programmsteuerung
- ALU
  - ⇨ Register
  - ⇨ Addierer
  - ⇨ Multiplizierer
  - ⇨ Schieberegister
- Speicherzellen
  - ⇨ RAM
  - ⇨ ROM

5 Rechnerarithmetik

- Die Rechnerarithmetik behandelt
  - ⇨ die Darstellung von Zahlen
  - ⇨ Verfahren zur Berechnung der vier Grundrechenarten
  - ⇨ Schaltungen, die diese Verfahren implementieren

5.1 Formale Grundlagen

- Menschen rechnen und denken im Dezimalsystem
- Die meisten Rechner verwenden das Dualsystem
  - ⇨ man benötigt Verfahren der Konvertierung, die sich algorithmisch umsetzen lassen

7.1.1 Zahlensysteme

- Stellenwertsysteme
  - ⇨ jeder Position i der Ziffernreihe ist ein Stellenwert zugeordnet welcher der Potenz b^i der Basis b eines Zahlensystems entspricht $z_n z_{n-1} \ldots z_1 z_0 . z_{-1} z_{-2} z_{-m}$
  - ⇨ der Wert X_b ergibt sich aus der Summe der Werte aller Einzelstellen

$$X_b = z_n b^n + z_{n-1} b^{n-1} + \ldots + z_1 b + z_0 + z_{-1} b^{-1} + z_{-2} b^{-2} + z_{-m} b^{-m} = \sum_{i=-m}^{n} z_i b^i$$

Die wichtigsten Zahlensysteme

b	Zahlensystem	Ziffern
2	Dualsystem	0, 1
8	Oktalsystem	0, 1, 2, 3, 4, 5, 6, 7
10	Dezimalsystem	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
16	Hexadezimalsystem	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

- Dualsystem kann direkt auf 2-wertige Logik umgewandelt werden
- Oktal- und Hexadezimalsystem sind Kurzschreibweisen der Zahlen im Dualsystem
  - ⇨ sie lassen sich leicht in Zahlen des Dualsystems umwandeln

Umwandlung vom Dezimalsystem in ein Zahlensystem zur Basis b

- Euklidischer Algorithmus
  - ⇨ die einzelnen Ziffern werden sukzessive berechnet

$$Z = z_n 10^n + z_{n-1}10^{n-1} + ... + z_1 10 + z_0 + z_{-1}10^{-1} + z_{-2}10^{-2} + z_{-m}10^{-m}$$
$$= y_p b^p + y_{p-1}b^{p-1} + ... + y_1 b + y_0 + y_{-1}b^{-1} + y_{-2}b^{-2} + y_{-q}b^{-q}$$

  - ⇨ Algorithmus
    1. Berechne P gemäß der Ungleichung $b^{p-1} \le Z < b^p$
    2. Ermittle y_p und den Rest R_p durch Division von Z durch b^p
       $y_p = Z \; div \; b^p$; $R_p = Z \; mod \; b^p$; $y_p = \{0, 1, ..., b\text{-}1\}$
    3. Wiederhole 2. für $i = p\text{-}1$ und ersetze dabei nach jedem Schritt Z durch R_i, bis $R_i = 0$ oder bis b_i klein genug ist

Beispiel

- **Umwandlung von** $15741{,}233_{10}$ **ins Hexadezimalsystem**

1. Schritt	$16^3 \le 15741{,}233_{10} < 16^4$	**höchste Potenz** 16^3
2. Schritt	$15741{,}233_{10} : 16^3 = 3$	**Rest** 3453,233
3. Schritt	$3453{,}233 : 16^2 = D$	**Rest** 125,233
4. Schritt	$125{,}233 : 16 = 7$	**Rest** 13,233
5. Schritt	$13{,}233 : 1 = D$	**Rest** 0,233
6. Schritt	$0{,}233 : 16^{-1} = 3$	**Rest** 0,0455
7. Schritt	$0{,}0455 : 16^{-2} = B$	**Rest** 0,00253
8. Schritt	$0{,}00253 : 16^{-3} = A$	**Rest** 0,000088593
9. Schritt	$0{,}000088593 : 16^{-4} = 5$	**Rest** 0,000012299

Fehler

Ergebnis: $15741{,}233_{10} = 3D7D{,}3BA5_{16}$

Umwandlung vom Dezimalsystem in eine Zahl zur Basis *b*

- **Horner-Schema**
  - ⇨ **Eine ganze Zahl** X_b **kann auch in der folgenden Form dargestellt werden:**

$$X_b = ((...(((y_n b + y_{n-1})b + y_{n-2})b + y_{n-3})b...)b + y_1)b + y_0$$

- **Die gegebene Dezimalzahl wird sukzessive durch die Basis** *b* **dividiert**
  - ⇨ **Die jeweiligen ganzzahligen Reste ergeben die Ziffern der Zahl** X_b
  - ⇨ **Reihenfolge: niederwertige zur höchstwertige Stelle**
- **Beispiel: Umwandlung von** 15741_{10} **ins Hexadezimalsystem**

$15741_{10} : 16 = 983$	**Rest** 13	(D_{16})
$983_{10} : 16 = 61$	**Rest** 7	(7_{16})
$61_{10} : 16 = 3$	**Rest** 13	(D_{16})
$3_{10} : 16 = 0$	**Rest** 3	(3_{16})

Ergebnis: $15741_{10} = 3D7D_{16}$

Umwandlung des Nachkommateils

- **Der Nachkommateil einer Zahl X_b kann in der folgenden Form dargestellt werden**

$$Y_b = ((...((y_{-m}b^{-1} + y_{-m+1})b^{-1} + y_{-m+2})b^{-1} + ... + y_{-2})b^{-1} + y_{-1})b^{-1}$$

- **sukzessive Multiplikation des Nachkommateils der Dezimalzahl mit der Basis b des Zielsystems ergibt nacheinander die y_{-i}**
- **Beispiel: Umwandlung von $0{,}233_{10}$ ins Hexadezimalsystem**

0,233 * 16	= 3,728	$z_{-1} = 3$
0,728 * 16	= 11,648	$z_{-2} = B$
0,648 * 16	= 10,368	$z_{-3} = A$
0,368 * 16	= 5,888	$z_{-4} = 5$

Ergebnis: $0{,}233_{10} = 0{,}3BA5_{16}$

Umwandlung einer Zahl zur Basis b ins Dezimalsystem

- **Werte der einzelnen Stellen werden mit deren Wertigkeit multipliziert und aufsummiert**
- **Beispiel: Umwandlung von $101101{,}1101$ ins Dezimalsystem**

101101,1101

$1 * 2^{-4}$	=	0,0625
$0 * 2^{-3}$	=	0
$1 * 2^{-2}$	=	0,25
$1 * 2^{-1}$	=	0,5
$1 * 2^{-0}$	=	1
$0 * 2^{1}$	=	0
$1 * 2^{2}$	=	4
$1 * 2^{3}$	=	8
$0 * 2^{4}$	=	0
$1 * 2^{5}$	=	32
		$45{,}8125_{10}$

Weitere Umwandlungen

- **Umwandlung zwischen zwei beliebigen Zahlensystemen**
  - ⇨ **zwei Schritte: Umwandlung ins Dezimalsystem und danach vom Dezimalsystem ins Zielsystem**
- **Spezialfall: Eine Basis eine Potenz der anderen Basis**
  - ⇨ **Umwandlung erfolgt durch Zusammenfassen der Stellen**
  - ⇨ **Beispiel: Umwandlung von** $0110100{,}110101_2$ **ins Hexadezimalsystem**

0011	0100,	1101	0100
3	4,	D	4

5.1.2 Kodierung zur Zahlen- und Zeichendarstellung

- **Die Dezimalzahlen können auch ziffernweise in eine Binärdarstellung überführt werden**
  - ⇨ **um die 10 Ziffern 0 bis 9 darstellen zu können, benötigt man 4 Bit**
  - ⇨ **eine solche 4er-Gruppe wird Tetrade genannt**
  - ⇨ **Pseudotetraden: 6 der 16 Kodierungen stellen keine gültigen Ziffern dar**
- **BCD**
  - ⇨ **Binary Coded Decimals**
  - ⇨ **man verwendet das Dualäquivalent der ersten 10 Dualzahlen**
  - ⇨ **Beispiel:**

$$8127_{10}=1000\ 0001\ 0010\ 0111_{BCD}=1111110111111_2$$

  - ⇨ **Nachteile der BCD-Kodierung**
    - **höherer Platzbedarf**
    - **aufwändige Implementierung der Rechenoperationen**

Gray-Kodierung

- Einschrittige Kodierung
  - ⇨ bei benachbarten Zahlen ändert sich nur ein Binärzeichen
- Vorteil
  - ⇨ keine Hazards bei der Analog/Digitalwandlung und bei Abtastern
- Nachteil
  - ⇨ keine Stellenwertigkeit
  - ⇨ aufwändige Rechenoperationen

Dezimalzahl	Gray-Codierung
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

Kodierung von Zeichen

- American Standard Code for Information Interchange (ASCII)
  - ⇨ 7 Bit-Kodierung für 128 Zeichen
  - ⇨ 2*26 Zeichen, 10 Ziffern und 32 Kommunikationssteuerzeichen
- Umlaute und Sonderzeichen sind nicht enthalten
  - ⇨ 8-Bit Erweiterungen unterschiedlicher Computerhersteller
  - ⇨ Andere Verwendung des 8. Bits: Paritätsprüfung

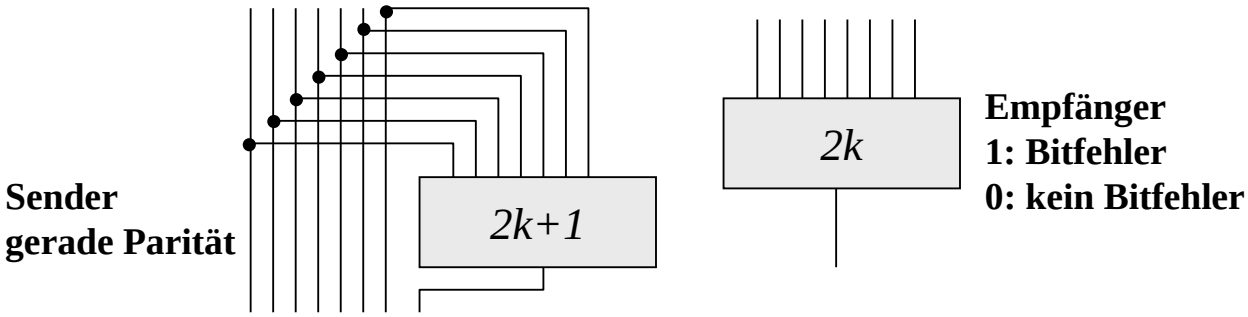
ASCII-Tabelle

	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	'	p
0001	SOH	DC 1	!	1	A	Q	a	q
0010	STX	DC 2	"	2	B	R	b	r
0011	ETX	DC 3	#	3	C	S	c	s
0100	EOT	DC 4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	´	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

Die höchstwertigen Bits der Kodierung eines Zeichens sind in der Kopfzeile abzulesen, die niederwertigen Bits in der ersten Spalte (Beispiel: A $\rightarrow 100\,0001_2$).

Paritätsprüfung

- **Problem:**
  - ⇨ **Erkennung von Übertragungsfehlern**
- **Prinzip:**
  - ⇨ **die 7-Bit Kodierung wird beim Sender so auf 8 Bit ergänzt, dass stets eine gerade (ungerade) Anzahl von Einsen ergänzt**
    - **gerade (ungerade) Parität**
  - ⇨ **beim Empfänger wird diese Eigenschaft überprüft**
    - **falls bei der Übertragung ein Bitfehler auftritt, wird dieser erkannt**



5.1.3 Darstellung negativer Zahlen

- Für die Darstellung von Zahlen in Rechnern werden vier verschiedene Formate benutzt
  - Darstellung mit Betrag und Vorzeichen
  - Stellenkomplement (Einerkomplement)
  - Zweierkomplement
  - Offset-Dual-Darstellung (Charakteristik)

Darstellung mit Betrag und Vorzeichen

- Die erste Stelle der Zahl wird als Vorzeichen benutzt
  - 0: Die Zahl ist positiv
  - 1: Die Zahl ist negativ
- Beispiel:
  - `0001 0011 = + 19`
  - `1001 0011 = - 19`
- Nachteile dieser Darstellung
  - bei Addition und Subtraktion müssen die Vorzeichen getrennt betrachtet werden
  - es gibt 2 Repräsentanten der Zahl 0
    - positives und negatives Vorzeichen

Einerkomplement

- Jede Ziffer der Binärzahl wird negiert
  - ⇨ negative Zahlen werden ebenfalls durch eine 1 an der 1. Stelle gekennzeichnet
- Vorteil:
  - ⇨ die 1. Stelle muss bei Addition und Subtraktion nicht gesondert betrachtet werden
- Beispiel:

```
        2        0010
+      -3      + 1100        (Komplement: 0011)
-------------------------
=      -1      = 1110        (Komplement: 0001)
```

- Nachteil:
  - ⇨ es gibt 2 Repräsentanten der Zahl 0:
    - 0000 und 1111

Zweierkomplement

- Addiert man zum Einerkomplement noch 1 hinzu, dann fallen die beiden Darstellungen der Zahl 0 durch den Überlauf wieder aufeinander
  - ⇨ Die Zahl 0 `0000`
  - ⇨ Einerkomplement `1111`
  - ⇨ Zweierkomplement `1111 + 0001 = 0000`
- Vorteile
  - ⇨ das 1. Bit enthält das Vorzeichen
  - ⇨ direkte Umwandlung der Zahl Z über die Stellenwertigkeit
- Beispiel $Z = -z_n \cdot 2^n + z_{n-1} \cdot 2^{n-1} + \ldots + z_1 \cdot 2 + z_0$
  - ⇨ Die Zahl 54 = 00110110_2
  - ⇨ mit Vorzeichenbit -54_{10} = 10110110_2
  - ⇨ Einerkomplement = 11001001_2
  - ⇨ Zweierkomplement = 11001010_2

Addition im Zweierkomplement

- **Beispiel:**

```
    73          01001001
   -54          11001010
------------------------
 =  19       (1)00010011
```

- **Beispiel:**

```
    37          00100101
   -54          11001010
------------------------
 = -17          11101111     (00010001)
```


Charakteristik

- **Hauptsächlich in der Darstellung von Exponenten für Gleitkommazahlen**
  - ⇨ **der gesamte Zahlenbereich wird durch die Addition einer Konstanten so nach oben verschoben, dass die kleinste Zahl die Darstellung 0...0 erhält**

- **Übersicht der Zahlendarstellungen**

Dez.	Betrag mit Vorz.	Einerkomp.	Zweierkomp.	Charakteristik
-4	---	---	100	000
-3	111	100	101	001
-2	110	101	110	010
-1	101	110	111	011
0	100 oder 000	000 oder 111	000	100
1	001	001	001	101
2	010	010	010	110
3	011	011	011	111

5.1.4 Fest- und Gleitkommazahlen

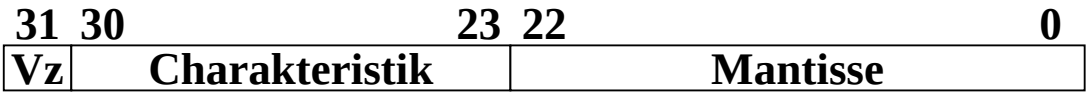
- Darstellung von Zahlen mit einem Komma
- Festkommadarstellung
  - ⇨ Festlegung der Stelle in einem Datenwort

0	1	0	1	1	0	0	1	0	,	1	1	0	0	0	0	0

  - ⇨ wird heute hardwareseitig nicht mehr eingesetzt
- Gleitkommadarstellung
  - ⇨ Angabe der Stelle des Kommas in der Zahlendarstellung

$$Z = \pm \text{Mantisse} \cdot b^{\text{Exponent}}, b \in \{2,16\}$$

  - ⇨ negative Zahlen werden meist in Betrag und Vorzeichen dargestellt (kein Zweierkomplement)
  - ⇨ sowohl für die Mantisse als auch für die Charakteristik wird eine feste Anzahl von Speicherstellen vorgesehen



Normalisierte Gleitkommadarstellung

- Eine Gleitkommazahl heißt normalisiert, wenn die folgende Beziehung gilt:

$$\frac{1}{2} \leq \text{Mantisse} < 1$$

  - ⇨ bei allen Zahlen außer der 0 ist die erste Stelle hinter dem Komma immer 1
  - ⇨ legt man für die Zahl 0 ein festes Bitmuster fest, kann man die erste 1 nach dem Komma weglassen
- Beispiel: Die Zahl 7135_{10}
  - ⇨ Festkommazahl

    `0 000 0000 0000 0000 0001 1011 1101 1111`$_2$
  - ⇨ Gleitkommadarstellung, normiert

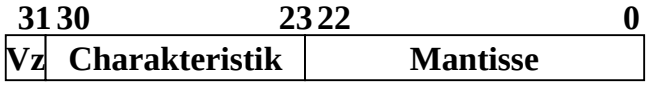
0	100 0110 1	110 1111 0111 1100 0000 0000

  - ⇨ Gleitkommadarstellung, normiert, implizite erste 1

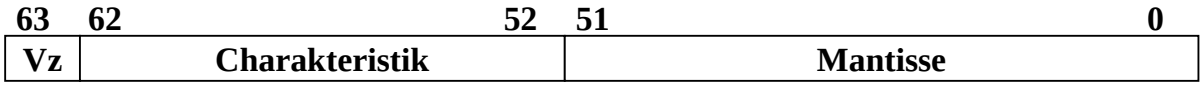
0	100 0110 1	101 1110 1111 1000 0000 0000

IEEE Gleitkommadarstellung

- Auch bei gleicher Wortbreite lassen sich unterschiedliche Gleitkommaformate definieren
  - ⇨ Normung durch IEEE
  - ⇨ einfache Genauigkeit (32 Bit)



  - ⇨ doppelte Genauigkeit (64 Bit)



- Eigenschaften
  - ⇨ Basis b ist gleich 2
  - ⇨ das erste Bit wird implizit zu 1 angenommen, wenn die Charakteristik nicht nur Nullen enthält
  - ⇨ Es wird so normalisiert, dass das erste Bit vor dem Komma steht

IEEE Gleitkommadarstellung

- Zusammenfassung des 32-bit IEEE-Formats:

Charakteristik	Zahlenwert
0	$(-1)^{Vz}$ 0,Mantisse*2^{-126}
1	$(-1)^{Vz}$ 1,Mantisse*2^{-126}
...	$(-1)^{Vz}$ 1,Mantisse*$2^{Charakteristik-127}$
254	$(-1)^{Vz}$ 1,Mantisse*2^{127}
255	Mantisse = 0: overflow, $(-1)^{Vz}\infty$
255	Mantisse ≠ 0: NaN (not a number)

- Um Rundungsfehler zu vermeiden, wird intern mit 80 Bit gerechnet

5.2 Addition und Subtraktion

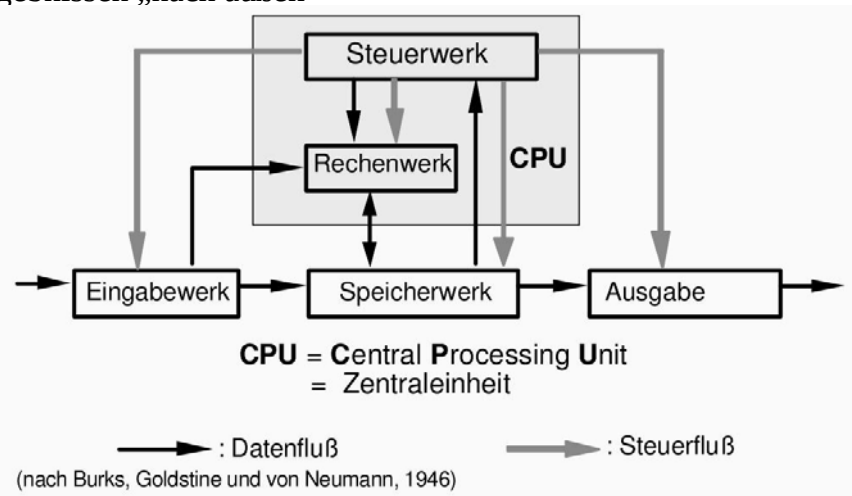
- Addition erfolgt Hilfe von Volladdierern wie im letzten Abschnitt beschrieben
  - ⇨ Ripple-Carry oder Carry-Look-Ahead Addierer
- Für die Subtraktion können ebenfalls Volladdierer verwendet werden
  - ⇨ `X -Y = X + (-Y)`
  - ⇨ Zweierkomplement berechnet sich über die Negation aller Bits mit einer 1 am ersten Übertrag des Addierers
- Bei Gleitkommazahlen müssen Mantisse und Exponent separat betrachtet werden
  - ⇨ Angleichen der Exponenten: Bilde die Differenz der Exponenten und verschiebe die Mantisse, die zum kleineren Exponenten gehört um die entsprechende Anzahl nach rechts
  - ⇨ Addition der Mantissen
  - ⇨ Normalisierung

5.3 Multiplikation und Division

- Prinzip der Multiplikation: Schieben und Addieren
- Multiplikation von Zahlen im Zweierkomplement:
  - ⇨ die Zahlen werden in eine Form mit Betrag und Vorzeichen konvertiert
  - ⇨ die Beträge werden Multipliziert (kaskadiertes Addierwerk)
  - ⇨ das neue Vorzeichen wird berechnet (Exklusiv-ODER-Verknüpfung)
- Prinzip der Division: Schieben und Subtrahieren
  - ⇨ zwei Sonderfälle:
    - Division durch 0 muss eine Ausnahme auslösen
    - Die Division muss abgebrochen werden, wenn die vorgegebene Bitzahl des Ergebnisregisters ausgeschöpft ist

6 Rechnerarchitektur und -organisation

- **Der von-Neumann-Rechner**
  - ⇨ **Speicher:** Speicherung von Progamm und Daten
  - ⇨ **Rechenwerk:** Ausführung arithmetischer/logischer Operationen
  - ⇨ **Leitwerk (Steuerwerk):** Steuerung des Programmablaufs
  - ⇨ **Ein- und Ausgabewerk:** Eingabe von Daten/Programmen, Ausgabe von Ergebnissen „nach außen“



Von-Neumann-Architektur

- **Klassischer Universalrechner** mit fest vorgegebener Struktur, die unabhängig vom bearbeiteten Problem ist.

- Zentrale Steuerung durch das Steuerwerk

- Programme werden von außen eingegeben und **im gleichen Speicher** wie die Daten abgelegt. Daten und Programm sind **binär codiert**.

  Interpretation eines Speicherinhalts hängt nur vom **aktuellen Kontext** des laufenden Programms ab.

- Speicher: in Einheiten (Speicherzellen) unterteilt und ansonsten unstrukturiert.

  Zugriff auf Speicherzellen: **direkt** über ihre Adresse.
  - ⇨ Befehle des Programms stehen in auf einander folgenden Speicherzellen

Von-Neumann-Architektur

⇨ Nächster Befehl wird durch Erhöhen der Befehlsadresse um Eins angesprochen - außer bei **Sprungbefehlen**:

- **Unbedingter Sprung**: Befehlsadresse wird auf `label` gesetzt
  `GO TO label`
- **Bedingter Sprung**: Befehlsadresse wird auf `label` gesetzt, falls `test`, sonst um Eins erhöht
  `IF test THEN label`

❍ **Befehlsabarbeitung nach 2-Phasen Konzept:**

⇨ **Interpretations-Phase**: Der entsprechend dem Stand des Befehlszählers geholte Inhalt einer Speicheradresse wird als Befehl interpretiert

⇨ **Ausführungs-Phase**: Aufgrund der im Befehl enthaltenen Adresse wird ein Speicherwort geholt und als Datenwert verarbeitet

Von-Neumann-Architektur

❍ **Die meisten heutigen Rechner beruhen auf dem von-Neumann-Prinzip besitzen jedoch Erweiterungen**

❍ **Funktionseinheiten heutiger Rechner:**

⇨ Arbeitsspeicher
→ normalerweise Speicherhierarchie
→ Im Befehl codierte Adressen werden modifiziert durch das Betriebssystem

⇨ Central Processing Unit (CPU): Steuerwerk + Rechenwerk
→ meist mehrere Rechen- und Steuerwerke
→ externe Signal können den Programmablauf unterbrechen

⇨ Ein- /Ausgabeeinheit

⇨ Datenwege zum Austausch von Informationen zwischen den Funktionseinheiten

Von-Neumann-Architektur

- **Vorteile der von-Neumann-Architektur**
  - ⇨ Geringer Speicheraufwand
  - ⇨ Geringer Hardwareaufwand

- **Nachteile**

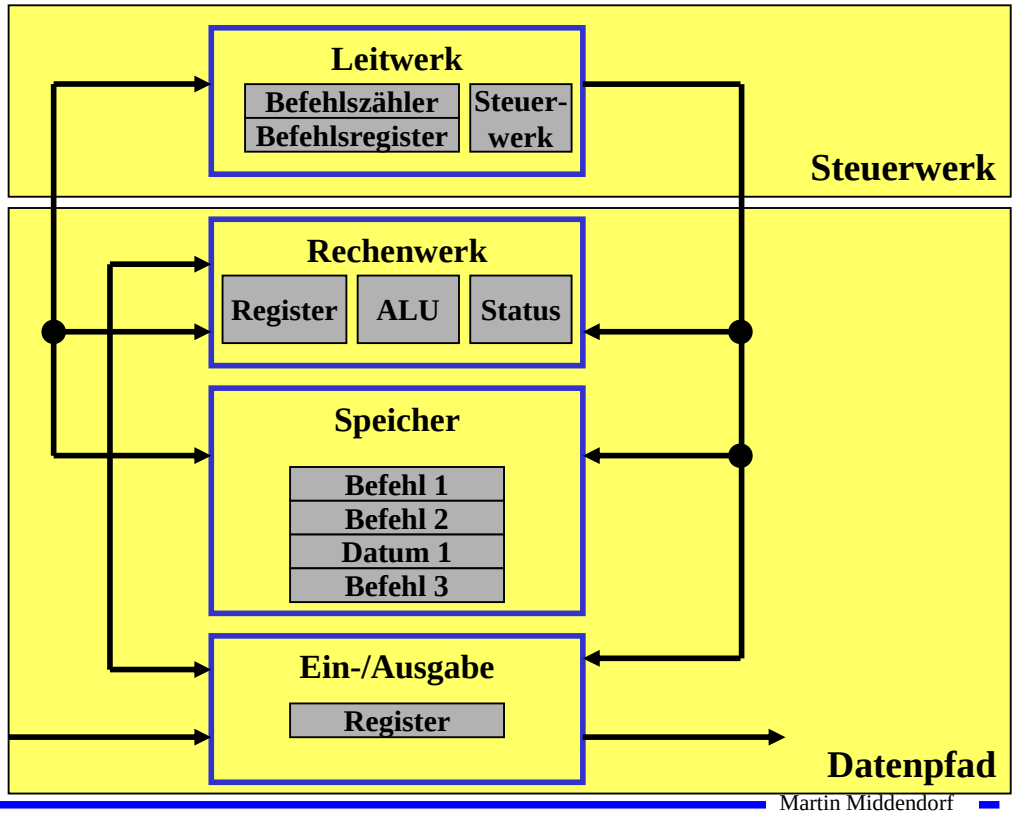
  - ⇨ Befehle werden nacheinander über die Verbindung zwischen Speicher und Steuerwerk geholt („von-Neumann-Flaschenhals“)

  - ⇨ Festlegung einer sequentiellen Bearbeitungsreihenfolge wird gefordert (intellektueller „von-Neumann-Flaschenhals“)

  - ⇨ Geringe Strukturierung der Daten

  - ⇨ Maschinenbefehl bestimmt den Operandentyp (semantische Lücke)

Von-Neumann-Architektur



Befehlsablauf im von-Neumann-Rechner

- **Lesen**
  - ⇨ **Einen neuen Programmzähler-Wert (PC) bestimmen**
  - ⇨ **Bestimmung der Speicheradresse des Quelloperanden**
  - ⇨ **Lesezugriff auf den Speicher**
  - ⇨ **Speichern des gelesenen Wertes im Zielregister**

- **Schreiben**
  - ⇨ **Einen neuen Programmzähler-Wert (PC) bestimmen**
  - ⇨ **Bestimmung der Speicheradresse des Zieloperanden**
  - ⇨ **Lesezugriff auf das Quellregister**
  - ⇨ **Schreibzugriff auf den Speicher**

Befehlsablauf im von-Neumann-Rechner

- **Verknüpfung von Operanden**
  - ⇨ **Einen neuen Programmzähler-Wert (PC) bestimmen**
  - ⇨ **Auslesen der Operanden aus dem Registerblock**
  - ⇨ **Verknüpfung der Operanden in der ALU**
  - ⇨ **Schreiben des Ergebnisses in den Registerblock**

- **Verzweigungen und Sprünge**
  - ⇨ **Einen neuen Programmzähler-Wert (PC) bestimmen**
  - ⇨ **Berechnung der Adresse des Sprungziels**
  - ⇨ **Prüfung der Sprungbedingung (bei Verzweigungen)**
  - ⇨ **Überschreiben des Befehlszählers, wenn der Sprung ausgeführt werden soll**

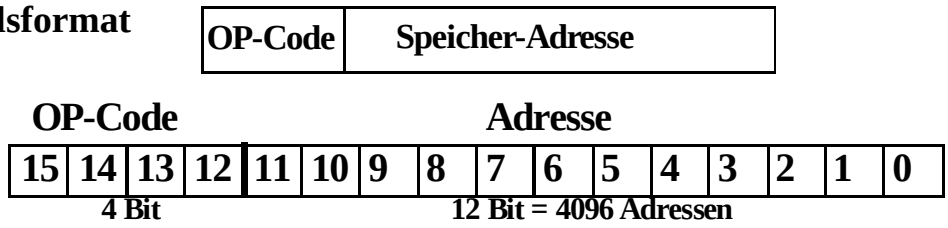
Der Toy-Rechner

- Implementierung einer einfachen von-Neumann-Architektur
  - ⇨ Quelle: Phil Kopmann, Microcoded versus Hard-Wired Logic
  - ⇨ Byte Januar 87, S. 235
  - ⇨ einfacher aber vollständiger Mikrorechner
  - ⇨ einfacher Aufbau mit Standardbausteinen

- RISC-Rechner
  - ⇨ alle Befehle in einem Takt (2 Phasen Takt)
  - ⇨ sehr einfacher Befehlssatz (12 Befehle)

Spezifikation des Toy-Rechners

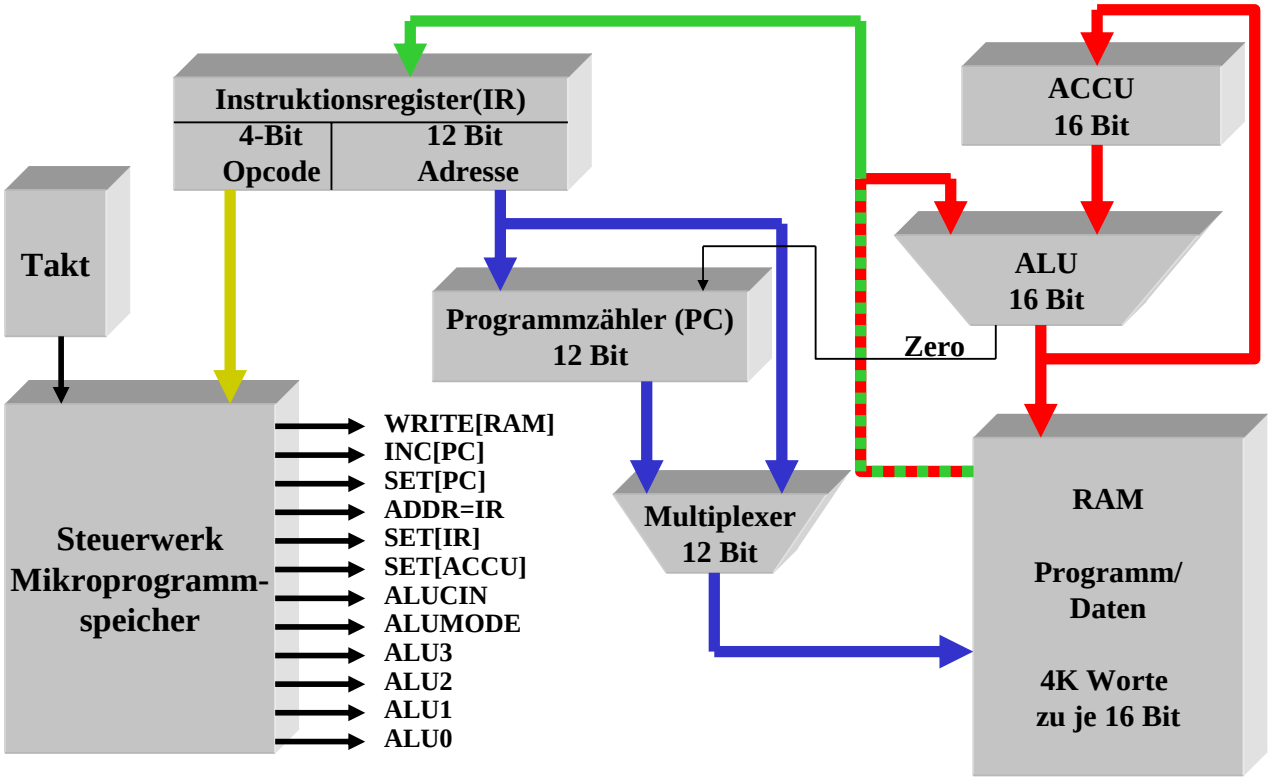
- 1-Adress-Maschine (nur ein Register)
  - ⇨ Ein Quelloperand kommt aus dem Speicher
  - ⇨ Der zweite Operand kommt aus dem Akkumulator (ACCU)
  - ⇨ Zielregister ist immer ACCU
- Befehlsformat



- Komponenten (Speicher CPU)

RAM:	4096 * 16 Bit
ALU:	4 * 74181 ALU-Baustein
ACCU:	Register
IR:	Instruktionsregister
PC:	Programmzähler
MUX:	Multiplexer

Blockschaltbild des Toy-Rechners



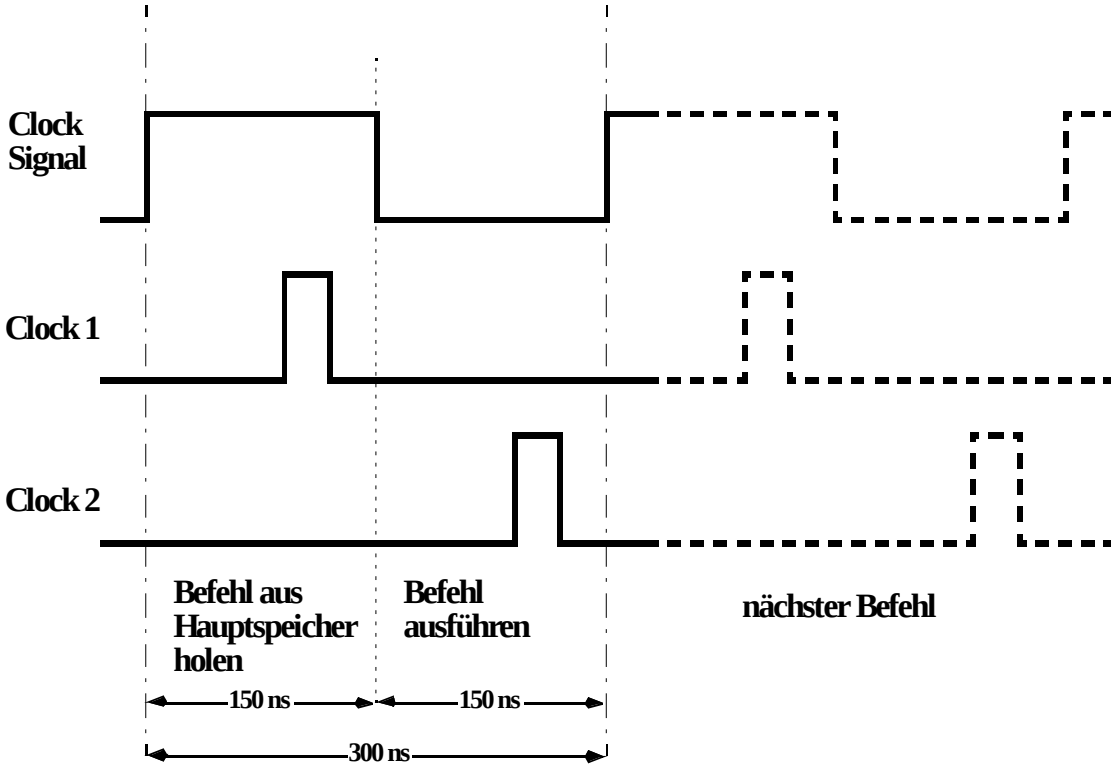
Befehlssatz

Opcode	Operation		Beschreibung
0	STO	<Adresse>	speichere Inhalt ACCU ins RAM an die Adresse
1	LDA	<Adresse>	lade ACCU mit dem Inhalt der Adresse
2	BRZ	<Adresse>	springe nach Adresse, wenn ACCU Null ist
3	ADD	<Adresse>	addiere den Inhalt der Adresse zum ACCU
4	SUB	<Adresse>	subtrahiere den Inhalt der Adresse vom ACCU
5	OR	<Adresse>	logisches ODER von ACCU und Inhalt der Adresse
6	AND	<Adresse>	logisches UND von ACCU und Inhalt der Adresse
7	XOR	<Adresse>	logisches ExODER von ACCU und Inhalt der Adresse
8	NOT		logisches NICHT der Bits im ACCU
9	INC		inkrementiere ACCU
10	DEC		dekrementiere ACCU
11	ZRO		setze ACCU auf NULL
12	NOP		nicht benutzt
13	NOP		nicht benutzt
14	NOP		nicht benutzt
15	NOP		nicht benutzt

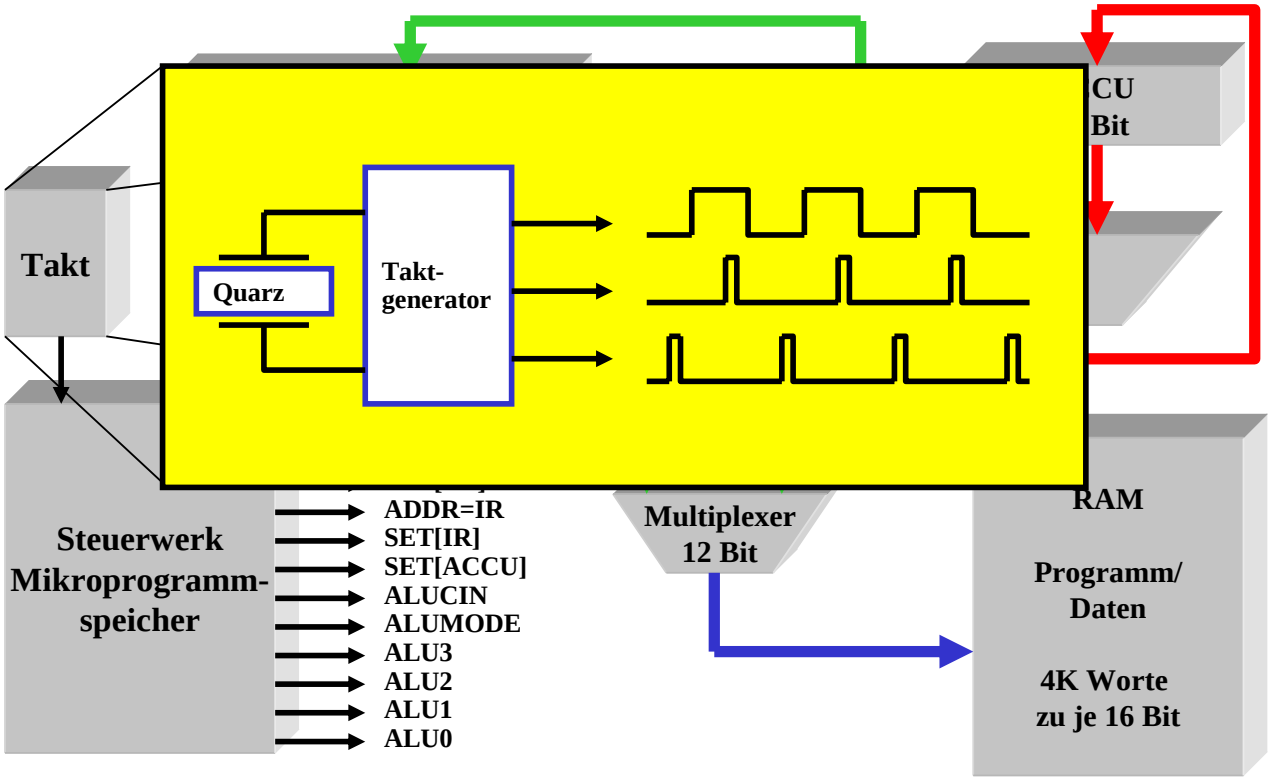
Spezifikation der Befehle

OpCode	Operation		Zyklus Beschreibung
0	STO	1	ADDR=IR; ALU=ACC; WRITE(RAM)
		2	ADDR=PC; SET(IR); INC(PC)
1	LDA	1	ADDR=IR; ALU=RAM; SET(ACC)
		2	ADDR=PC; SET(IR); INC(PC)
2	BRZ	1	SET[PC]
		2	ADDR=PC; SET(IR); INC(PC)
3	ADD	1	ADDR=IR; ALU=ACC+RAM; SET(ACC)
		2	ADDR=PC; SET(IR); INC(PC)
...			
9	INC	1	ALU=ACC+1; SET(ACC)
		2	ADDR=PC; SET(IR); INC(PC)
...			
12-15	NOP	1	ADDR=PC; SET(IR); INC(PC)
		2	

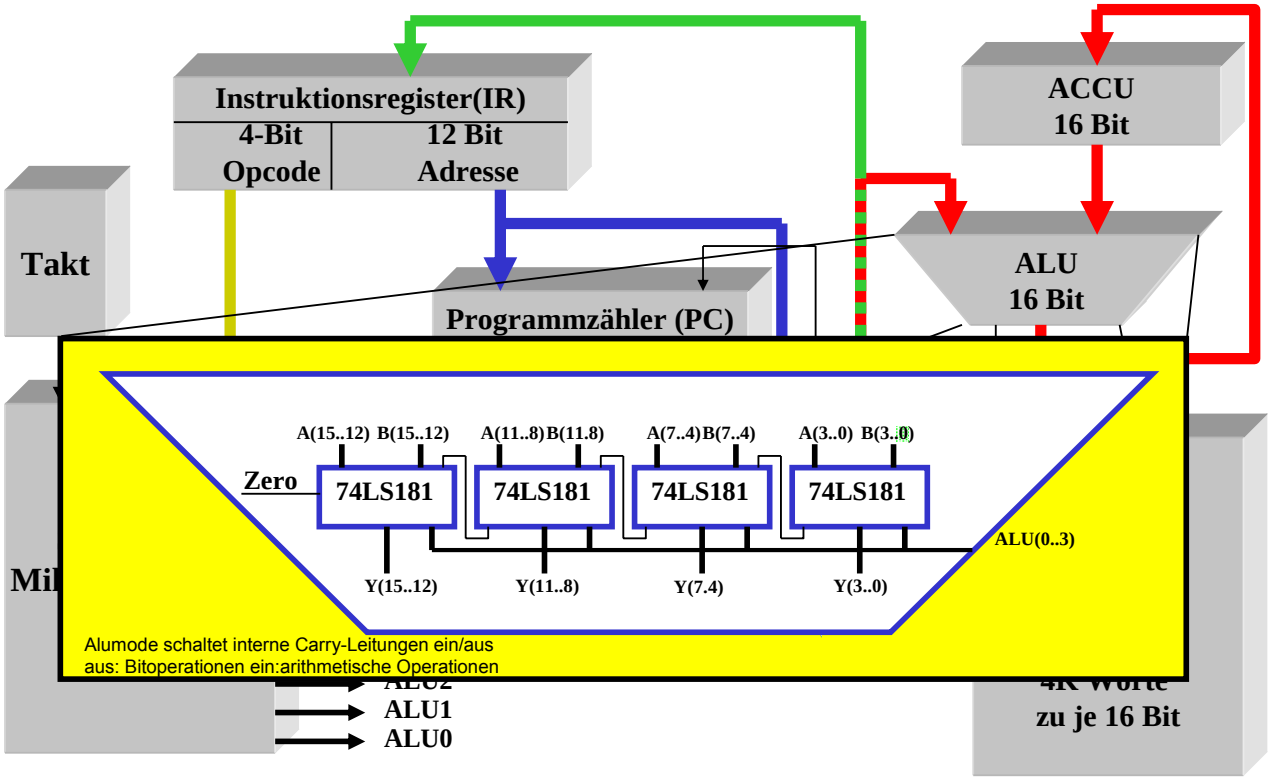
Ablaufsteuerung



Komponente 1: Der Taktgenerator



Komponente 2: Die ALU



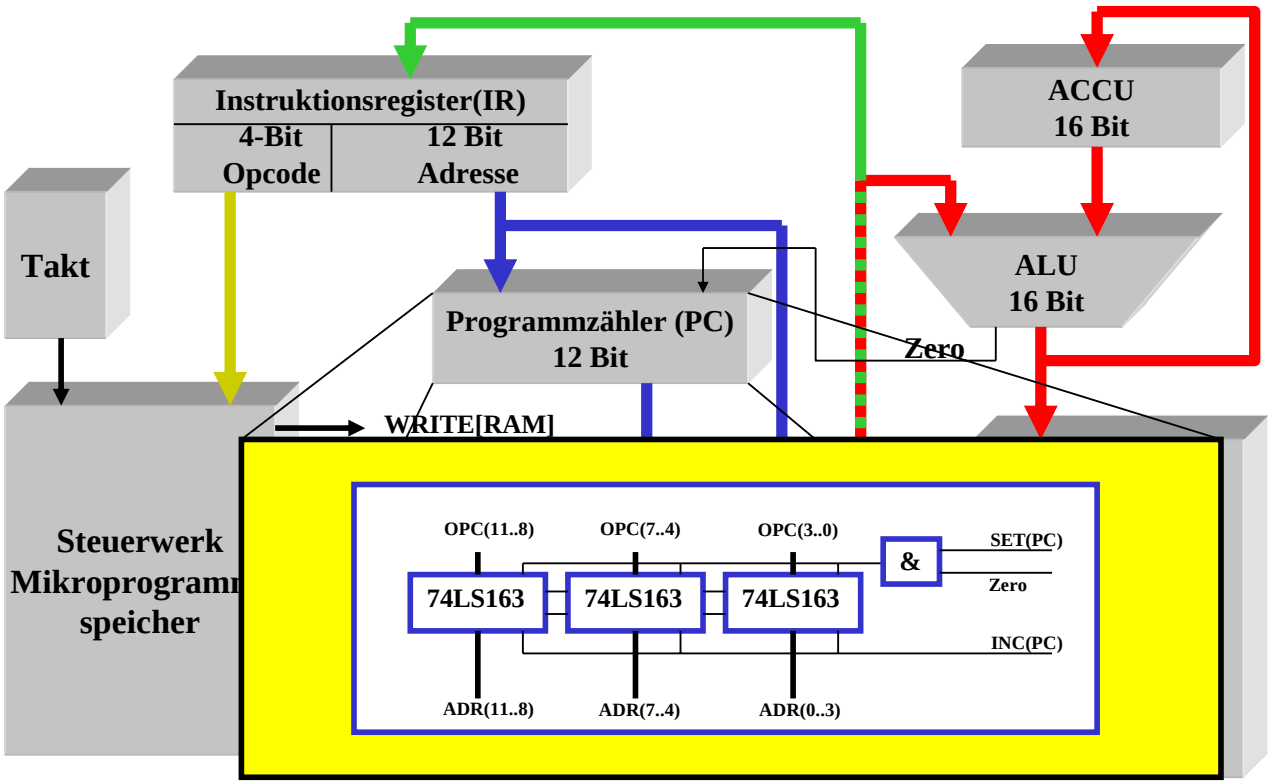
Ablauf eines Maschinenbefehls

- ALU (74LS181)

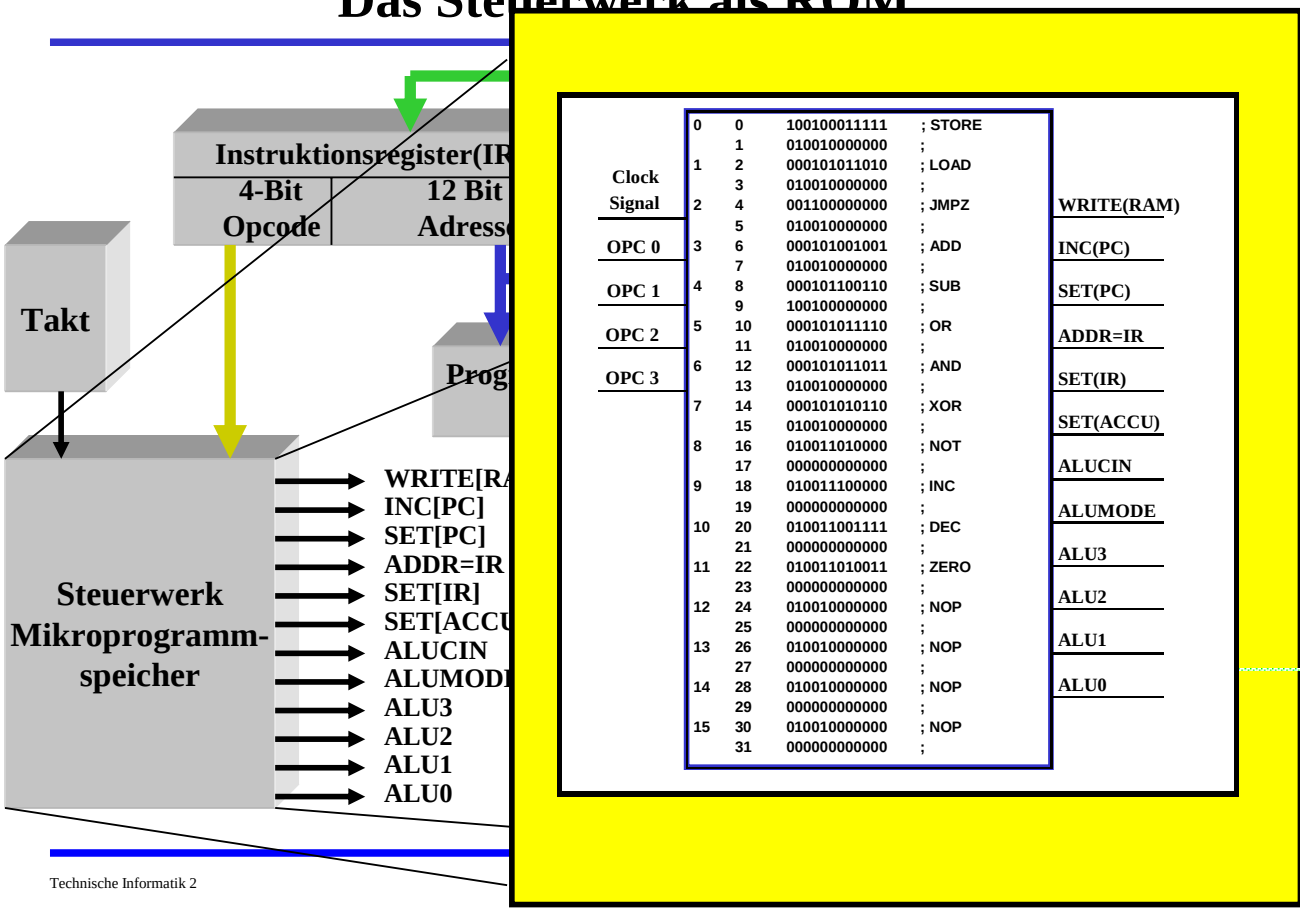
Function Table

Mode Select Inputs				Active LOW Operands & F_n Outputs	
				Logic	Arithmetic (Note 2)
S3	S2	S1	S0	(M = H)	(M = L) (C_n = L)
L	L	L	L	$\overline{A}$	A minus 1
L	L	L	H	$\overline{AB}$	AB minus 1
L	L	H	L	$\overline{A} + B$	$A\overline{B}$ minus 1
L	L	H	H	Logic 1	minus 1
L	H	L	L	$\overline{A} + \overline{B}$	A plus (A + $\overline{B}$)
L	H	L	H	$\overline{B}$	AB plus (A + $\overline{B}$)
L	H	H	L	$\overline{A} \oplus \overline{B}$	A minus B minus 1
L	H	H	H	$A + \overline{B}$	A + $\overline{B}$
H	L	L	L	$\overline{A}$ B	A plus (A + B)
H	L	L	H	A ⊕ B	A plus B
H	L	H	L	B	$A\overline{B}$ plus (A + B)
H	L	H	H	A + B	A + B
H	H	L	L	Logic 0	A plus A (Note 1)
H	H	L	H	$A\overline{B}$	AB plus A
H	H	H	L	AB	$A\overline{B}$ minus A
H	H	H	H	A	A

Komponente 3: Der Befehlszähler



Das Steuerwerk als ROM



Ablauf eines Maschinenbefehls

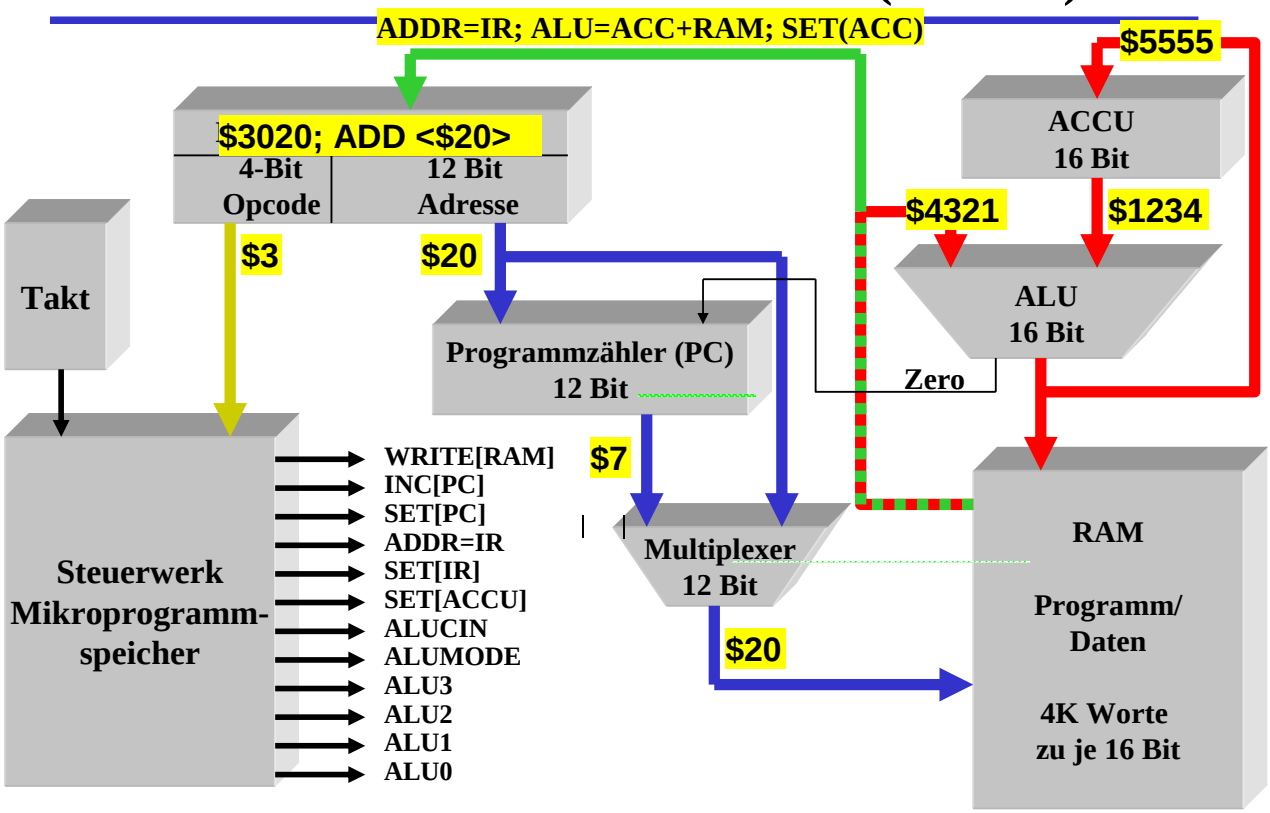
- Ab der Speicherstelle $0007 steht die Befehlssequenz:

  $0007: $3020 ; ADD <$20>
  $0008: $0030 ; STO <$30>

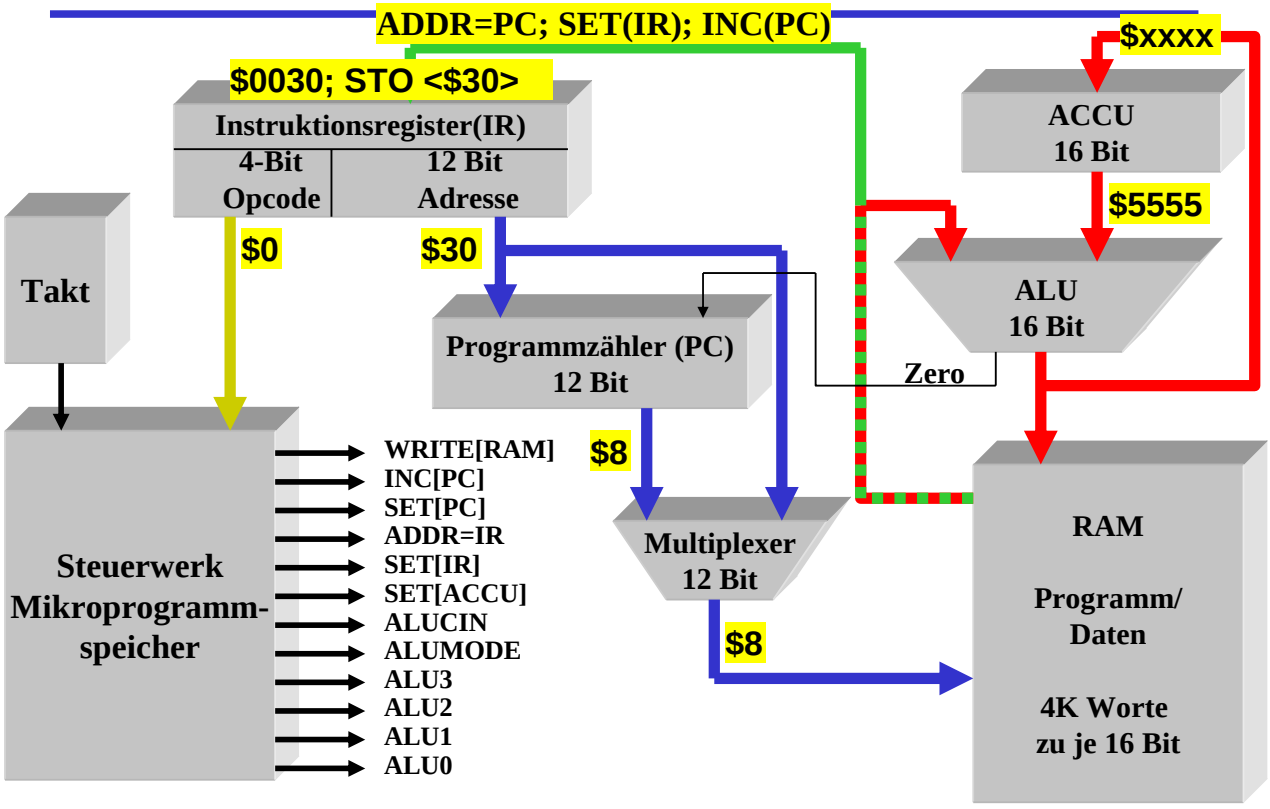
- Der Akkuinhalt ist $1234.
- Der Inhalt der Speicherstelle $20 ist $4321

- Wie werden die Befehle abgearbeitet?

Ablauf eines Maschinenbefehls (Phase 1)



Ablauf eines Maschinenbefehls (Phase 2)



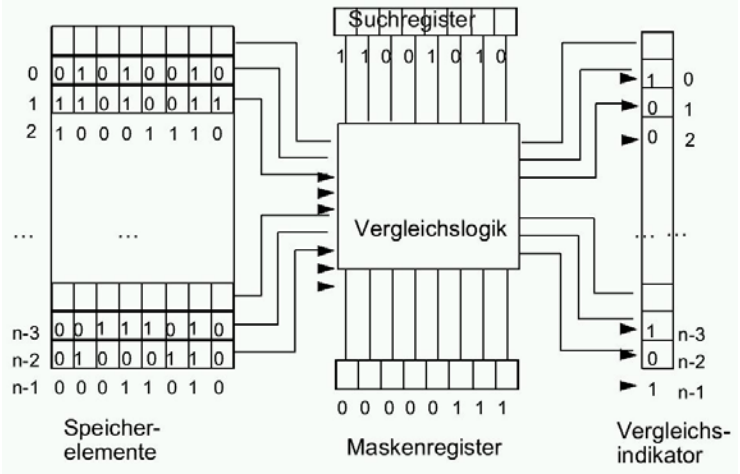
Unterschiede zu realen Rechnern

	Toy-Rechner	reale Prozessoren
Wortlänge	16 Bit Daten 12 Bit Adressen	bis 100 Bit
Mikroinstruktionen	1 Routine pro Maschinenbefehl	mehrere Routinen pro Maschinenbefehl
Umfang des Mikroprogramms	384 Bit	300 000 Bit
Verzweigungsbefehle	1 Verzweigungsbefehl	10-33 Verzweigungsbefehle
Adressierungsmodi	1 Adressierungsmodus	1-21 Adressierungsmodi
Befehlssatz	12 Befehle	bis zu 300 Befehle
Registersatz	1 Register (Akku)	32 - 512 Register

Alternative Konzepte

A. Assoziativspeicher - auch inhaltsadressierbarer Speicher oder CAM (Content Addressable Memory CAM):

- ⇨ **Inhalt Suchregister wird teil- oder vollparallel mit den Inhalten aller Speicherelemente verglichen**
- ⇨ **Inhalt Maskenregister bestimmt die Teile des Schlüsselwortes mit denen vergleichen wird**
- ⇨ **Im Trefferregister werden die Speicherworte mit Übereinstimmung angegeben**



Alternative Konzepte

B. Trennung von Datenspeicher und Programmspeicher (Harvard-Architektur) :

- ⇨ **Nächster Befehl wird bereits aus dem Datenspeicher geholt, während Ergebnisdaten in den Datenspeicher geschrieben werden**
- ⇨ **Realisierung meist auf Ebene des Cachespeichers (oder bei digitalen Signalprozessoren)**

C. Hardwareunterstützung komplexer Datentypen

Beispiel: Vektorrechner

- ⇨ **Adressierung eines Vektor mittels eines Vektor-Deskriptors:**
  - **Basisadresse**
  - **Länge (Anzahl der Elemente)**
  - **Datentyp eines Elements**
- ⇨ **Zugriff auf den gesamten Vektor wird durch Ausführen einer Operation veranlasst**

Alternative Konzepte

D. Parallelverarbeitung

- ⇨ **Gleichzeitige Ausführung mehrerer Operationen**
- ⇨ **Man unterscheidet verschiedene Abstraktionsebenen der Parallelität**

  feinkörnige Parallelität - Befehle oder Teilschritte der Abarbeitung eines Befehls

  grobkörnige Parallelität – Programme oder funktionelle Teile eines Programms
- ⇨ **Voraussetzung: mehrere Verarbeitungseinheiten stehen zur Verfügung**
  - **Rechenwerke (Vektorrechner: Feldrechner, Pipeline-Rechner)**
  - **Prozessoren (Multiprozessorsysteme)**
  - **vollständige Rechner (verteilte Systeme)**

Alternative Konzepte

Ebenen der Parallelverarbeitung:

- ⇨ **Befehlsphasen-Ebene: Laden, Dekodieren, Operanden holen, …**
- ⇨ **Ebene der Elementaroperationen: Addition, logisches UND, …**
- ⇨ **Anweisungs-Ebene: Befehle der Programmiersprache**
- ⇨ **Task-Ebene: Funktionelle Einheiten eines Programms**

  Beispiel: leichtgewichtige Threads arbeiten im gleichen Adressraum

- ⇨ **Job-Ebene: Programm, dass aus mehreren Teilen (Tasks) bestehen kann**

Alternative Konzepte

Beispiel: Befehlsphasen-Pipelining

Befehlsausführung sei in folgende Schritt aufgeteilt:

1. Befehl holen (F)
2. Befehl dekodieren (D)
3. Operanden holen (O)
4. Befehl ausführen (E)

Ziel: Die Schritte beim Pipelining sollten möglichst etwa gleich viel Zeit benötigen

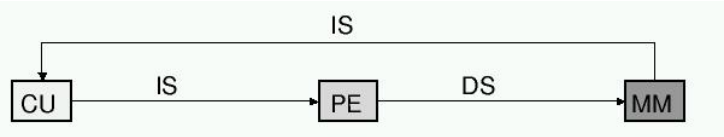
Takt 1	F_1			
Takt 2	F_2	D_1		
Takt 3	F_3	D_2	O_1	
Takt 4	F_4	D_3	O_2	E_1
Takt 5		D_4	O_3	E_2
Takt 6			O_4	E_3
Takt 7				E_4

Ausführen von *N* Befehlen benötigt *N+3* Takte statt *4N* bei Ausführung ohne Pipelining

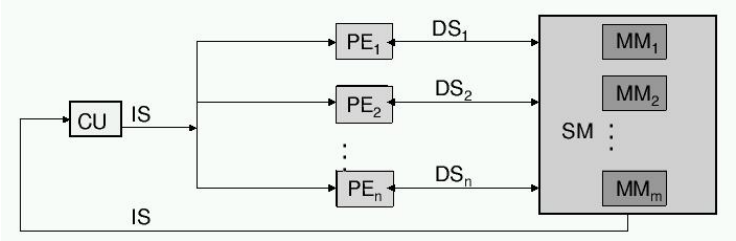
Alternative Konzepte

- **Klassifikation von Parallelrechnern nach Flynn** (1972):
  - ⇨ **Geht aus von einer Parallelisierung auf Anweisungsebene**
  - ⇨ **Es gibt andere Klassifikationsschemata.**

SISD: Single Instruction Single Data

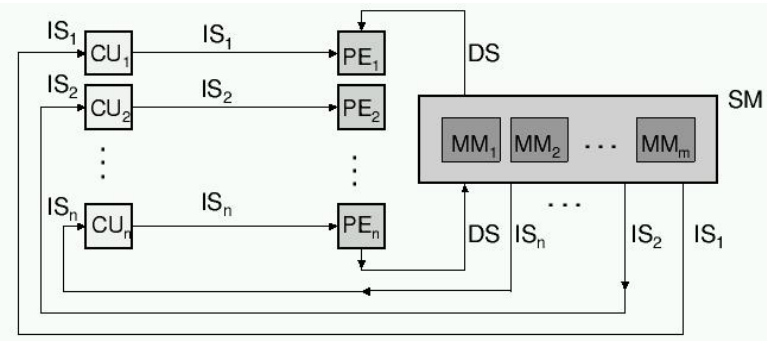


SIMD: Single Instruction Multiple Data

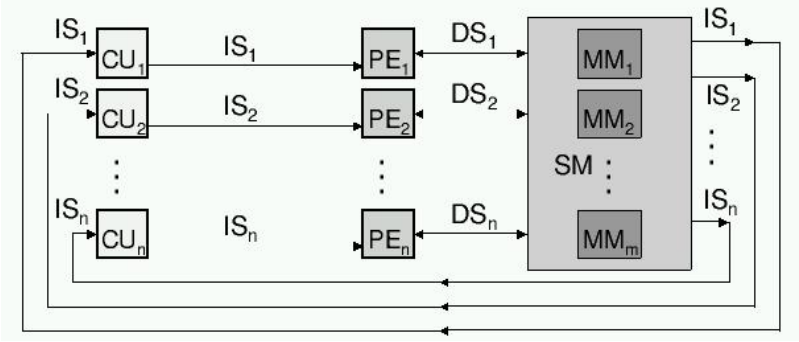


Alternative Konzepte

MISD: Multiple Instruction Single Data: nur Spezialanwendungen



MIMD: Multiple Instruction Multiple Data



Alternative Konzepte

Beispiel: Multiprozssorsysteme

- ⇨ MIMD Prinzip
  häufig **SPMD** (Single Programm Multiple Data)
- ⇨ Anwendung: grobkörnige Parallelität

- ⇨ Klassifizierung nach
  1. Art der Prozessoren
     gleiche Prozessoren → **homogenes Systeme**
     ansonsten → **heterogenes System**
  2. Funktion der Prozessoren
     gleiche Funktion → **symmetrisches System**
     ansonsten → **asymmetrisches System**
  3. Kopplung zwischen den Prozessoren
     - gemeinsamer Speicher (**shared memory**)
     - Verschicken von Nachrichten (**message passing**)

7. Maschinen- und Assemblersprache

Um Rechner einsetzen zu können, benötigt man Programme als Beschreibung der auszuführenden Rechenschritte.

Sprachen dienen der Formulierung von Programmen.

Problem: „nackte Maschine“ arbeitet auf Bitketten.

Für den Anwender ist das Programmieren auf Basis von Bitketten sehr unkomfortabel.

Lösung: Hierarchie von Sprachen mit unterschiedlichem Abstraktionsgrad, wobei jede Sprache

- ⇨ **in die Sprache der Ebene darunter übersetzbar ist**
- ⇨ **oder durch Interpretierung direkt ausführbar ist**

Jede Sprache arbeitet somit auf einer virtuellen Maschine.

Sprachebenen

1. **Problemorientierte (höhere) Programmiersprache:**
   - dient zur Formulierung von Programmen eines Problembereichs, unabhängig von einer Maschine.

2. **Assemblersprache:**
   - Maschinenorientierte Sprache, die sich an den Eigenschaften eines bestimmten Rechners orientiert
   - ähnlich wie Maschinensprache, erlaubt jedoch symbolische Notation

3. **Maschinensprache:**
   - Besteht aus allen Befehlen, die durch die CPU direkt ausgeführt werden können (Maschinenbefehle)
   - Befehle werden binär dargestellt
   - Ausführung der Befehle je nach Rechner
     - interpretativ durch Mikroprogramme
     - direkt durch „festverdrahtete“ Hardware

Sprachebenen

4. **Mikroprogrammierung:**
   - Dient der Ausführung von Maschinenbefehlen im Steuerwerk
   - Maschinenbefehle werden als Folge von Mikrobefehlen dargestellt

Maschinensprache

Maschinensprache ist die niedrigste frei für die Programmierung zugängliche Ebene.

Die Menge der verfügbaren Maschinenbefehle (Instruktionssatz) charakterisiert eine Rechnerarchitektur.

Maschinenbefehle bestehen meist aus:

- **Operationscode: Angabe der auszuführenden Operation**
- **Operandenadressen: Spezifikation der Operanden auf die die Operation angewendet werden soll durch**
  - **Konstanten**
  - **Registeradressen**
  - **Hauptspeicheradressen**

Typische Befehlsformate: OpCode OpAdr1 OpAdr2 Opadr3 …

•1-Adress-Befehl: spezifizierte Operand wird mit Inhalt ACCU verknüpft, Ergebnis steht im ACCU

Maschinensprache

- **2-Adress-Befehl: spezifizierte Operanden werden verknüpft, Ergebnis steht im zweiten Operanden**

- **3-Adress-Befehl: Erster und zweiter Operand werden verknüpft, Ergebnis steht im dritten Operanden**

→ **1-Adress-, 2-Adress oder 3-Adress-Maschinen.**

Befehlsformat kann auch abhängig von der Operation sein:
- Sprungbefehle → eine Adresse
- arithmetische Operationen → ein, zwei, oder mehr Adressen

Arten von Maschinenbefehlen:
- Datentransport
- Arithmetische und logische Operationen
- Ablaufsteuerung
- Ein/Ausgabe
- Sonderbefehle: Unterbrechungsbehandlung, Anhalten oder Rücksetzen der CPU, …

Assemblersprache

Assemblerbefehle werden nicht numerisch sondern symbolisch notiert:

- **Operationscodes erhalten symbolischen („mnemonisch“) Namen**
- **Operandenadressen können Namen zugeordnet werden. Adressierung erfolgt über diesen Namen**
- **Befehle können durch Namen (Label) gekennzeichnet werden (Festlegung von Sprungzielen)**
- **Verschiedene Datenformate (Zeichenkette, Dezimalzahl,…)**

Unterstützung verschiedener Adressformate (z.B.):

- **Direktoperand**
- **Register**
- **Adresse**
- **Adresse + Direktoperand**

→ **Falls Maschinensprache nur Adressen mit Registern unterstützt, (RISC) muss Assembleradressierung darauf abgebildet werden.**

Assemblersprache

Pseudobefehle sind zusätzliche Befehle an den Assembler:

- **Zuweisung von Werten/Adressen an symbolische Namen (Symboldefinition)**
  Diese werden beim Assemblieren durch Werte ersetzt
  Beispiel: SIZE EQU 100 – ordne dem Symbol SIZE den Wert 100 zu

- **Festlegung der Anfangsadresse des Programms**
  Beispiel: ORG $500 - beginne Programm bei $500

- **Initialisierung von Speicherplatz**
  Beispiel: DC.B „Hallo“ – belege die nächsten 5 Bytes im Speicher mit den ASCII Werten der Buchstaben H, a,…

- **Reservierung von Speicherplatz für Variablen**

- **Exportieren/Importieren von Symbolen aus anderen Assemblerprogrammen oder von Dateien**
  Beispiel: INCLUDE Datei – setze Quelltext aus Datei ein

Assemblersprache

Befehlsaufbau:

- **Markenfeld: zur symbolischen Kennzeichnung eines Assemblerbefehls (entspricht auf Maschinenebene einer Adresse)**
- **Operationsfeld: enthält entweder mnemonische Notation eines Maschinenbefehls oder Teil eines Pseudobefehls**
- **Operandenfeld: enthält (je nach Befehlsart) null, ein oder mehr Operanden (Konstante oder Adressangaben)**
- **Kommentarfeld: (optional) zur Dokumentation**

Assemblersprachen bieten wenig oder keine Konzepte zur Strukturierung von Daten und Programmabläufen.

Assemblierer (Assembler)

Assemblierer: Programm zur Übersetzung von Assemblerprogrammen in ablauffähige Maschinenprogramme

Aufgaben:

- **Syntaxanalyse der Befehle**
- **Ausführen der Pseudobefehle**
- **Konvertierung von Konstanten in Binärdarstellung**
- **Generierung des Maschinencodes für den Operationsteil des Maschinenbefehls**
- **Berechnung von Adressen die durch Symbole oder Ausdrücke gegeben sind**
- **Erstellung eines Protokolls (Fehlermeldungen, …)**

Adressberechnung:

- **relativ zum Programmanfang (absolute Adresse wird erst von Binder und Lader festgelegt)**
- **absolut (ORG Anweisung)**

Assemblierer (Assembler)

Problem: Einmaliges Durchlaufen des Assemblerprogramms beim Assemblieren genügt nicht, da Adressen von Vorwärtssprüngen (Vorwärtsreferenzen) noch nicht bekannt sind

Deshalb Assemblierung des Programms in (mindestens) 2 Durchläufen
→ 2-Pass Assemblierer

1. Durchlauf (Pass 1):

- **Aufbau der Symboltabelle, d.h. alle verwendeten Marken/Namen werden zusammen mit ihrem Wert und evtl. weiteren Angaben (z.B. ob Angabe relativ oder absolut ist) in einer Datei gesammelt**
- **Syntaxprüfung**
- **Ausführen der Pseudobefehle**

2. Durchlauf (Pass 2):

- **Einsetzen der Adresswerte für die Symbole mit Hilfe der Symboltabelle im übersetzten Programm**
- **Generierung des Maschinencodes**

Assemblierer (Assembler)

Protokollpass: Erstellung eines Protokolls der Übersetzung

- **Fehlermeldungen**
- **Programmprotokoll**
- **Symboltabelle**
- **Kreuzreferenztabelle: Zeilennummer des Vorkommens jedes Symbols**
- **Informationen für den Lader/Binder:**
  - **Länge des Programms**
  - **Adressen die relativ sind und geändert werden müssen**
  - **Ende des Programms**

Evtl. weitere Durchläufe für Sonderfunktionen:

- **Auflösung von Makros: Dienen der Zusammenfassung häufig benutzter Befehlsfolgen**
- **Codeoptimierung**

Lader/Binder

Binder:

- **Zusammenfügen mehrerer Programmsegmente (Assemblierte Programmteile und Bibliotheksprogramme) zu einem Lademodul**
  - **Auflösen externer Referenzen**
  - **Adressberechnungen**

Auch möglich: dynamisches Binden (d.h. zur Laufzeit; evtl. ist erst dann bekannt welcher Code ausgeführt werden soll)

Lader:

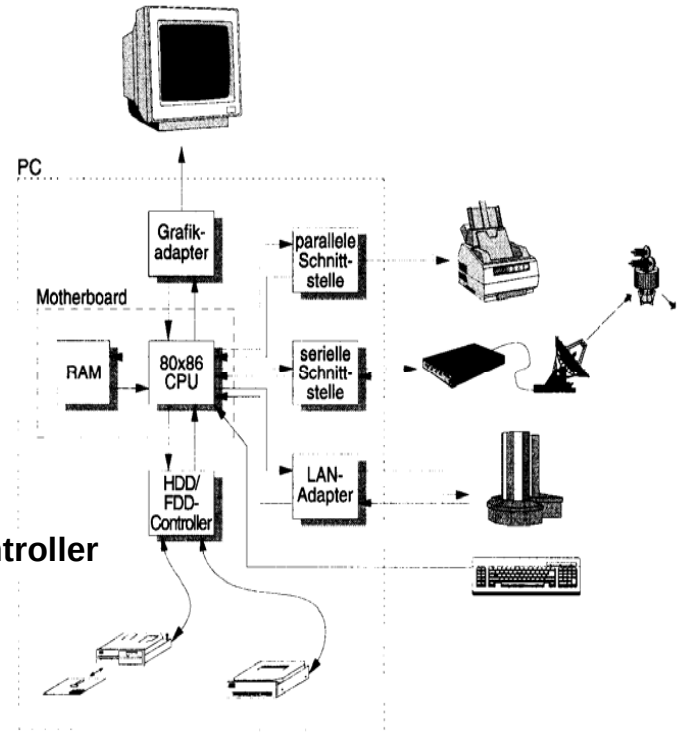
- **Speicherplatz anfordern**
- **Umrechnen von Adressen (relativ in absolut)**
  → dazu wird entsprechende Information vom Assembler benötigt: relokierbare Programme
- **Laden des Objektcodes in den Hauptspeicher**
  → jedoch heute meist virtuelle Adressierung
- **Eventuell starten der Ausführung**

Ein Beispielprogramm

```
; Variablen:
;  Loopcount=$20, Number=$21 (enthaelt zunaechst 0)
; Labels:
;  loop=$2, end=$b
;
$0020        ; STO Loopcount     ; Auswerten des initialen
                                 ; Accuinhalts
$200b        ; BRZ end           ; Schon fertig?
#--------------------------------
#loop:
$1021        ; LDA Number        ; nat. Zahl mitzaehlen
$9000        ; INC
$0021        ; STO Number
$1020        ; LDA Loopcount     ; Schleifenzaehler aktualisieren
$a000        ; DEC
$0020        ; STO Loopcount
$200b        ; BRZ end           ; Fertig?
$b000        ; ZRO               ; Nein,
$2002        ; BRZ loop          ;       dann wieder von vorn
#--------------------------------
#end:
$b000        ; ZRO
$200c        ; BRZ end           ; Endlosschleife
```

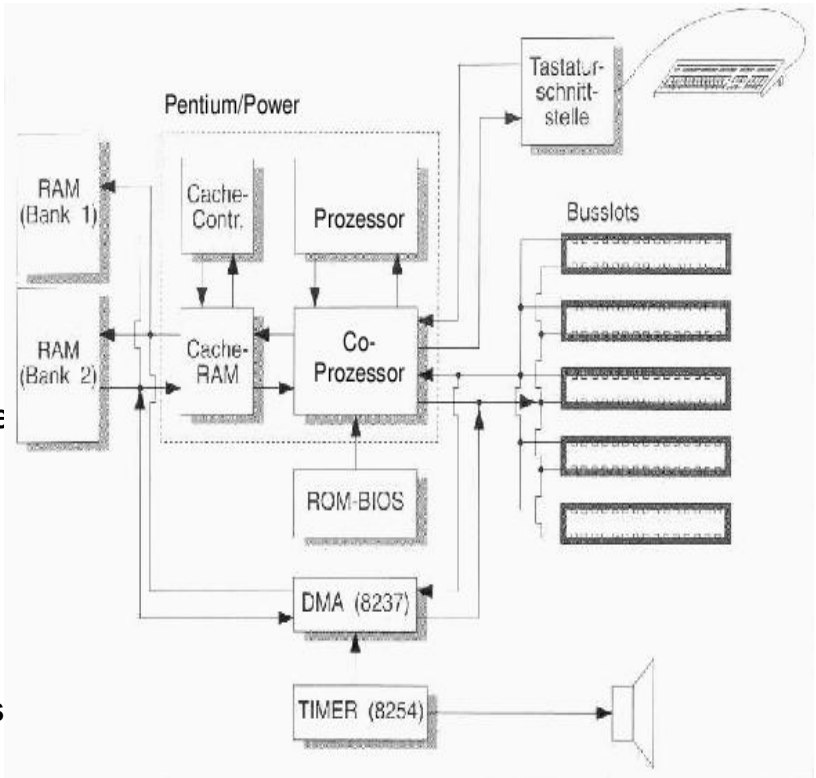
7 Aufbau von Rechnersystemen

- Speicher
  - RAM, ROM, Cache
- Prozessor
  - Integer
  - Gleitkommaarithmetik
  - Cachecontroller
- E/A
  - Tastatur
  - Grafikkarte
  - Disketten/Festplattencontroller
  - Netzwerkkarte



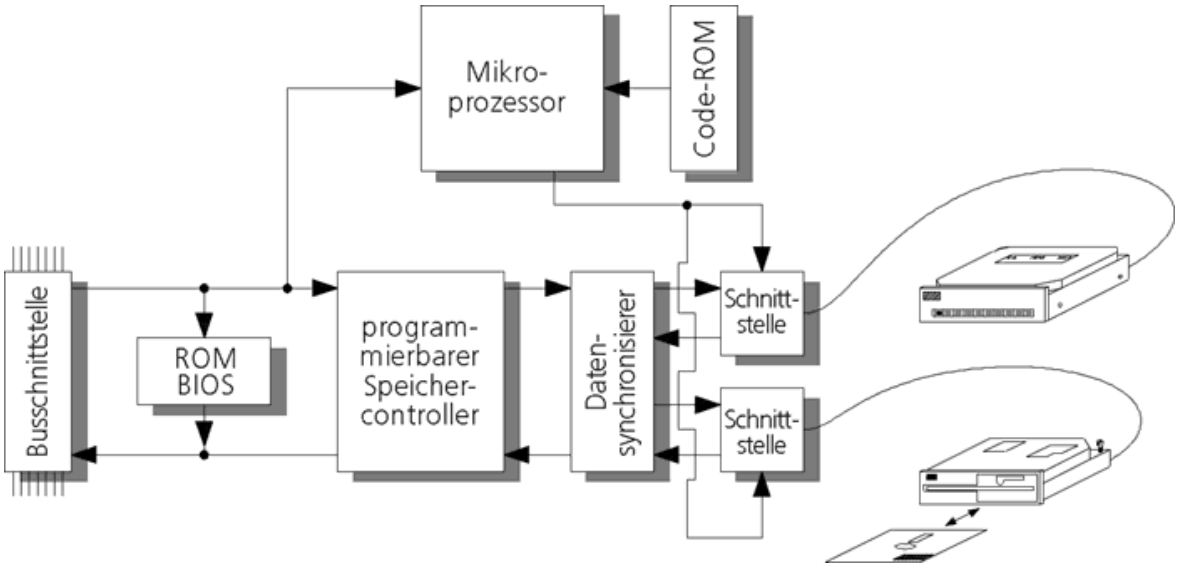
Hauptkomponenten der Zentraleinheit

- Speicher
  - RAM
  - ROM
- Prozessor
  - Integer-CPU
  - Gleitkomma -Prozessor
  - Cache
  - Cachecontrolle
- Bus
- Peripherie
  - Schnittstellen
  - Timer
  - DMA (Direct Memory Acces



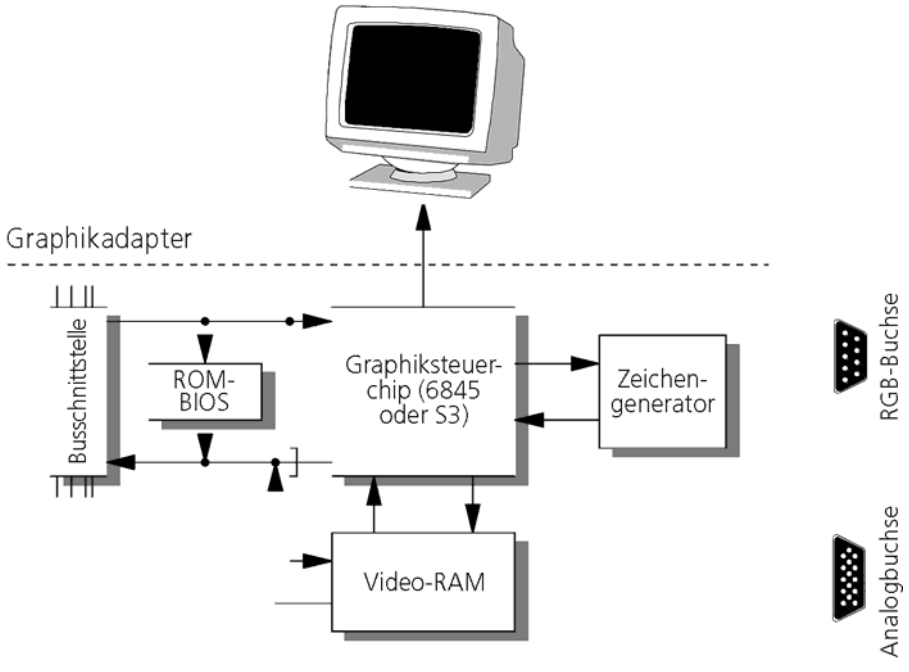
Peripherie

Festplatten- und Diskettencontroller



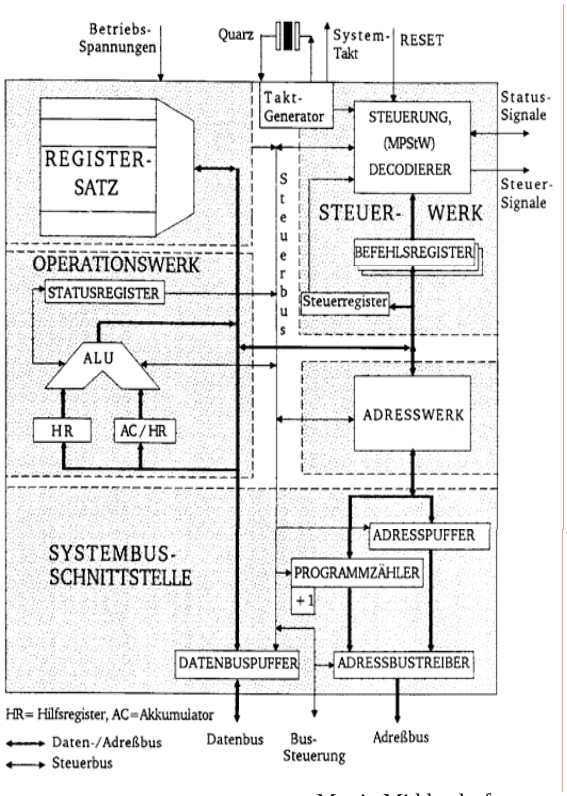
Peripherie

Grafikadapter



Prinzipieller Aufbau eines typischen Mikroprozessors

- **Steuerwerk**
  - ⇨ Liefert die Steuersignale für das Rechenwerk
  - ⇨ Steuert den Ablauf der Operationen
- **Rechenwerk (Operationswerk)**
  - ⇨ führt die arithmetischen und logischen Operationen aus
- **Registersatz**
  - ⇨ speichert die Operanden für das Rechenwerk
- **Adresswerk**
  - ⇨ Berechnet die Adressen für die Befehle oder die Operanden
- **Systembus-Schnittstelle**
  - ⇨ Treiber
  - ⇨ Zwischenspeicher
  - ⇨ Adresszähler



CISC- und RISC-Prozessoren

CISC „Complex Instruction Set Computer“:

- Viele, teilweise komplexe Instruktionen (>100)
- Viele Adressierungsarten (>10)
- Anzahl der Takte abhängig von Befehl und Adressierungsart
- Speicher/Register Operationen
- i.A. wenige Register

Beispiele: Intel 486, Intel Pentium, Motorola 680x0

RISC „Reduced Instruction Set Computer“:

- Wenige, meist sehr einfache Instruktionen (<50)
- Wenige Adressierungsarten (<4)
- Möglichst eine Instruktion pro Takt
- Register/Register-Operationen (Speicherzugriffe nur mit LOAD/STORE)
- viele Register (>32)

Beispiele: SPARC, PowerPC, ARM Alpha

Das Steuerwerk

- Synchrones Schaltwerk

- Komponenten eines typischen Steuerwerks
  - ⇨ Befehlsdekodierer: analysiert und entschlüsselt aktuellen Befehl
  - ⇨ Steuerung: generiert die Signale für das Rechenwerk
  - ⇨ Befehlsregister: speichert den aktuellen Befehl
  - ⇨ Steuerregister: liefert Bedingungen zur Entscheidung des Befehlsablaufs
    - z.B. Interrupt enable bit, aktuell erlaubte Befehle (System/User Bit)

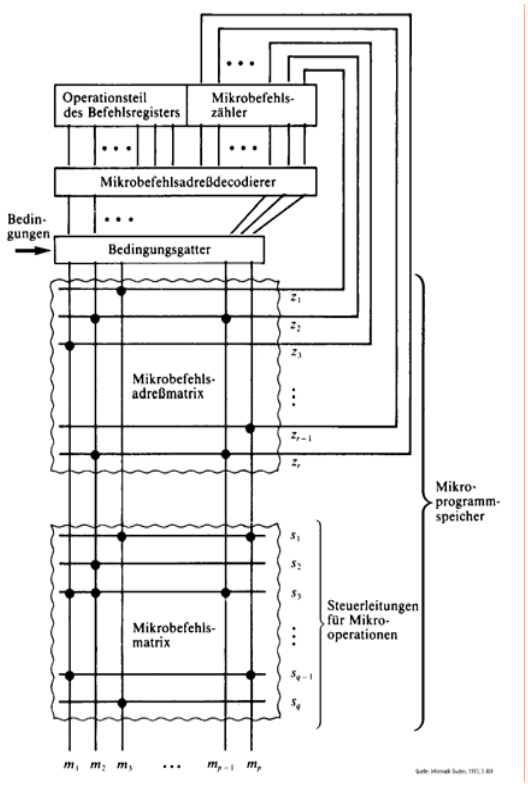
- Realisierungsmöglichkeiten für das Steuerwerk:
  - ⇨ **Festverdrahtet:** als System mehrstufiger logischer Gleichungen implementiert
  - ⇨ **Mikroprogrammiert**: in einem ROM implementiert
  - ⇨ **Mikroprogrammierbar**: in einem RAM implementiert, wird beim Neustart des Prozessors geladen

Mikroprogrammierung

- **Mikrooperationen**
  - ⇨ elementare Operationen wie das Setzen eines Registers

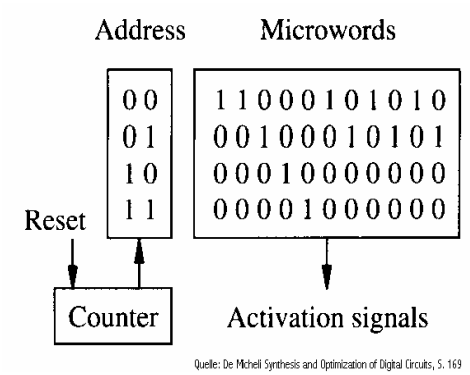
- **Mikrobefehle**
  - ⇨ Zusammenfassung bestimmter Mikrooperationen, die zu einem Taktzeitpunkt gleichzeitig ausgeführt werden können

- **Mikroprogrammierung**
  - ⇨ Realisierung der Maschinenbefehle durch eine Folge von Elementaroperationen

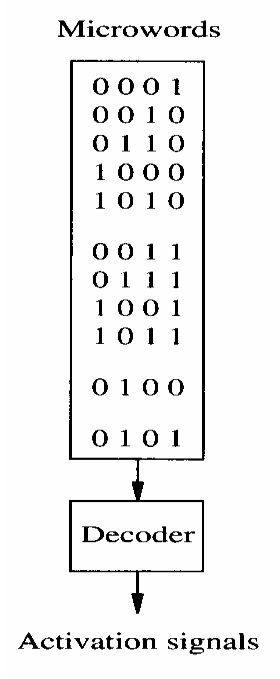


Vertikale und horizontale Mikroprogrammierung

- **Horizontale Mikroprogrammierung**
  - ⇨ **Jedes Ausgangssignal erhält eine eigene Steuerleitung**
- **Vertikale Mikroprogrammierung**
  - ⇨ **Die Ausgangssignale werden über einen Multiplexer angesteuert**



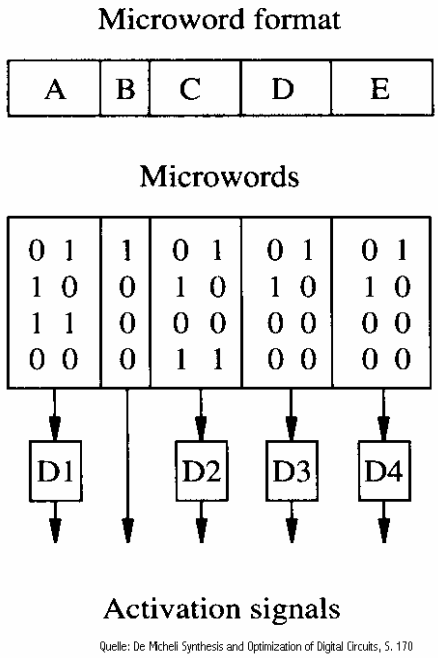
Quelle: De Micheli Synthesis and Optimization of Digital Circuits, S. 169



horizontal **vertikal**

Mischformen

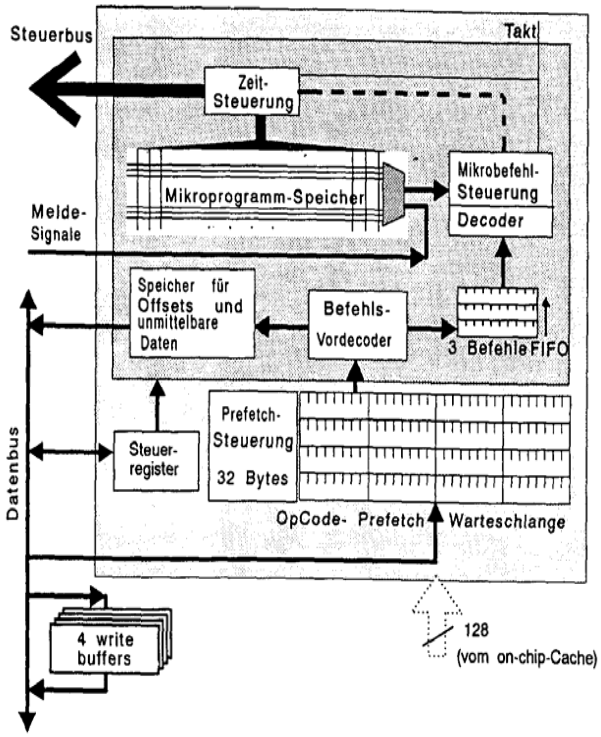
- **Diagonale Mikroprogrammierung**
  - ⇨ **<u>Unabhängige</u> Teile des horizontalen Mikrobefehlsworts werden zusammengefaßt und vertikal kodiert**



Quelle: De Micheli Synthesis and Optimization of Digital Circuits, S. 170

diagonal

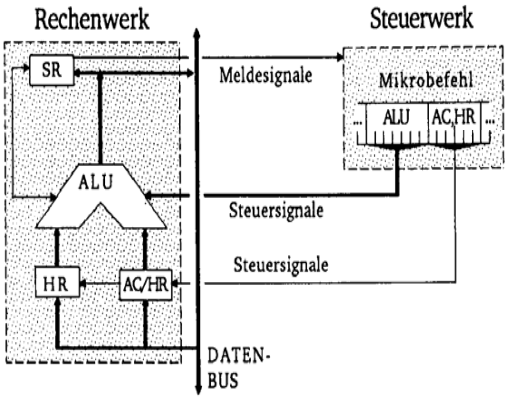
Das Steuerwerk des Intel 486



Das Rechenwerk

- **ALU**
  - ⇨ **berechnet alle Operationen**
- **Akkumulator**
  - ⇨ **speichert das Ergebnis einer Operation**
  - ⇨ **stellt einen Operanden zur Verfügung**
- **Hilfsregister**
  - ⇨ **stellt den zweiten Operanden zur Verfügung**
- **Statusregister**
  - ⇨ **Speichert besondere Ergebnisse**

AC: Akkumulator
HR: Hilfsregister
SR: Statusregister



Das Statusregister

- Einzelne logisch unabhängige Bits
  - ⇨ CF (Carry Flag) — Übertrag
  - ⇨ ZF (Zero Flag) — Ergebnis der letzten Operation ist 0
  - ⇨ SF (Sign Flag) — negatives Ergebnis bei der letzten Operation
  - ⇨ OF (Overflow Flag) — Überlauf bei der letzten Operation
  - ⇨ EF (Even Flag) — Gerades Ergebnis bei der letzten Operation
  - ⇨ PF (Parity Flag) — ungerade Anzahl der ´1´-Bits
- Diese Flags werden bei bedingten Sprüngen ausgewertet

Transfer- und Ein-/Ausgabebefehle

Mnemonic	Bedeutung
LD	Laden eines Register (*load*)
LEA	Laden eines Registers mit der Adresse eines Operanden (*load effective address*)
ST	Speichern des Inhalts eines Registers (*store*)
MOVE	Übertragen eines Datums (in beliebiger Richtung)
EXC	Vertauschen der Inhalte zweier Register bzw. eines Registers und eines Speicherwortes (*exchange*)
TFR	Übertragen eines Registerinhalts in ein anderes Register (*transfer*)
PUSH	Ablegen des Inhalts eines oder mehrerer Register im Stack
PULL (POP)	Laden eines Registers bzw. mehrerer Register aus dem Stack
STcc	Speichern eines Registerinhaltes, falls die Bedingung cc (nach Tabelle 1.14-11) erfüllt ist

Mnemonic	Bedeutung
IN, READ	Laden eines Registers aus einem Peripheriebaustein
OUT, WRITE	Übertragen eines Registerinhalts in einen Peripheriebaustein

Arithmetische und Logische Befehle

Mnemonic	Bedeutung
ABS	Absolutbetrag bilden (*absolute*)
ADD	Addition ohne Berücksichtigung des Übertrags (*add*)
ADC	Addition mit Berücksichtigung des Übertrags (*add with carry*)
CLR	Löschen eines Registers oder Speicherwortes (*clear*)
CMP	Vergleich zweier Operanden (*compare*)
COM	bitweises Invertieren eines Operanden (Einerkomplement) (*complement*)
DAA	Umwandlung eines dualen Operanden in eine Dezimalzahl (*decimal adjust A*)
DEC	Register oder Speicherwort dekrementieren (*decrement*)
DIV	Division (*divide*)
INC	Register oder Speicherwort inkrementieren (*increment*)
MUL	Multiplikation (*multiply*)
NEG	Vorzeichenwechsel im Zweierkomplement (*negate*)
SUB	Subtraktion ohne Berücksichtigung des Übertrags (*subtract*)
SBC	Subtraktion mit Berücksichtigung des Übertrags (*subtract with carry*)

Mnemonic	Bedeutung
AND	UND-Verknüpfung zweier Operanden
OR	ODER-Verknüpfung zweier Operanden
EOR	Antivalenz-Verknüpfung zweier Operanden (*exclusive or*)
NOT	Invertierung eines (Booleschen) Operanden

Flag- und Bit-Manipulationsbefehle

Mnemonic	Bedeutung
SE<f>	Setzen eines Bedingungs-Flags (*set*)
CL<f>	Löschen eines Bedingungs-Flags (*clear*)
BSET	Setzen eines Bits (*bit set*)
BCLR	Rücksetzen eines Bits (*bit clear*)
BCHG	Invertieren eines Bits (*bit change*)
TST	Prüfen eines bestimmten Flags oder Bits (*test*)
BF...	Bitfeld-Befehle, insbesondere:
BFCLR	Zurücksetzen der Bits auf '0' (*clear*)
BFSET	Setzen der Bits auf '1' (*set*)
BFFFO	Finden der ersten '1' in einem Bitfeld (*find first one*)
BFEXT	Lesen eines Bitfeldes (*extract*)
BFINS	Einfügen eines Bitfeldes (*insert*)

(<f> Abkürzung für ein Flag, z.B. C *carry flag*)

Schiebe- und Rotationsbefehle

Mnemonic	Bedeutung	
SHF	Verschieben eines Registerinhaltes	(*shift*)
	insbesondere:	
ASL	arithmetische Links-Verschiebung	(*arithm. shift left*)
ASR	arithmetische Rechts-Verschiebung	(*arithm. shift right*)
LSL	logische Links-Verschiebung	(*logical shift left*)
LSR	logische Rechts-Verschiebung	(*logical shift right*)
ROT	Rotation eines Registerinhaltes	(*rotate*)
	insbesondere:	
ROL	Rotation nach links	(*rotate left*)
RCL	Rotation nach links durchs Übertragsbit	(*rotate with carry left*)
ROR	Rotation nach rechts	(*rotate right*)
RCR	Rotation nach rechts durchs Übertragsbit	(*rotate with carry right*)
SWAP	Vertauschen der beiden Hälften eines Registers	

Befehle zur Programmsteuerung

Sprung und Verzweigungsbefehle

Mnemonic	Bedeutung	
JMP	unbedingter Sprung zu einer Adresse	(*jump*)
Bcc	Verzweigen, falls die Bedingung cc erfüllt ist	(*branch*)
BRA	Verzweigen ohne Abfrage einer Bedingung	(*branch always*)

Unterprogrammaufrufe und Rücksprünge, Software-Interrupts

Mnemonic	Bedeutung
JSR, CALL	unbedingter Sprung in ein Unterprogramm (*jump to subroutine*)
BSRcc	Verzweigung in ein Unterprogramm, falls die Bedingung cc gilt (*branch to subroutine*)
RTS	Rücksprung aus einem Unterprogramm (*return from subroutine*)
SWI,TRAP,INT	Unterbrechungsanforderung durch Software (*software interrupt*)
RTI, RTE	Rücksprung aus einer Unterbrechungsroutine (*return from interrupt/exception*)

Bedingungen für Sprünge

cc	Bedingung	Bezeichnung
CS	CF = 1	*branch on carry set*
CC	CF = 0	*branch on carry clear* ,
VS	OF = 1	*branch on overflow*
VC	OF = 0	*branch on not overflow*
EQ	ZF = 1	*branch on zero/equal*
NE	ZF = 0	*branch on not zero/equal*
MI	SF = 1	*branch on minus*
PL	SF = 0	*branch on plus*
PA	PF = 1	*branch on parity/parity even*
NP	PF = 0	*branch on not parity/parity odd*
nicht vorzeichenbehaftete Operanden		
LO	CF = 1 (vgl. CS)	*branch on lower than*
LS	CF v ZF = 1	*branch on lower or same*
HI	CF v ZF = 0	*branch on higher than*
HS	CF = 0 (vgl. CC)	*branch on higher or same*
vorzeichenbehaftete Operanden		
LT	SF ≠ OF = 1	*branch on less than*
LE	ZF v (SF ≠ OF) = 1	*branch on less or equal*
GT	ZF v (SF ≠ OF) = 0	*branch on greater than*
GE	SF ≠ OF = 0	*branch on greater or equal*

(Bezeichnungen: ≠ Antivalenz, v logisches ODER)

Sonstige Befehle

Systembefehle

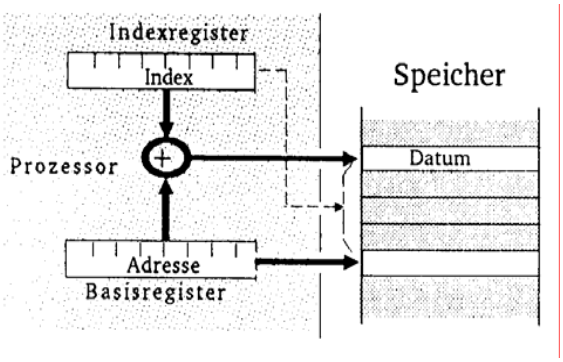
Mnemonic	Bedeutung
NOP	keine Operation, nächsten Befehl ansprechen (*no operation*)
WAIT	Warten, bis ein Signal an einem speziellen Eingang auftritt
SYNC	Warten auf einen Interrupt
HALT, STOP	Anhalten des Prozessors, Beenden jeder Programmausführung
RESET	Ausgabe eines Rücksetz-Signals für die Peripherie-Bausteine
SVC	(geschützter) Aufruf des Betriebssystem-Kerns (*supervisor call*)

Stringbefehle

Mnemonic	Bedeutung	
MOVS	Transferieren eines Blocks	(*move string*)
INS	Einlesen eines Blocks von der Peripherie	(*input string*)
OUTS	Ausgabe eines Blocks an die Peripherie	(*output string*)
CMPS	Vergleich zweier Blöcke	(*compare string*)
COPS	Kopieren eines Blocks	(*copy string*)
SCAS	Suchen eines Zeichens (Wortes) in einem Block	(*scan string*)

Der Registersatz

- Datenregister
  - Integerregister
  - Akkumulator
- Adressregister
  - Basisregister
  - Indexregister



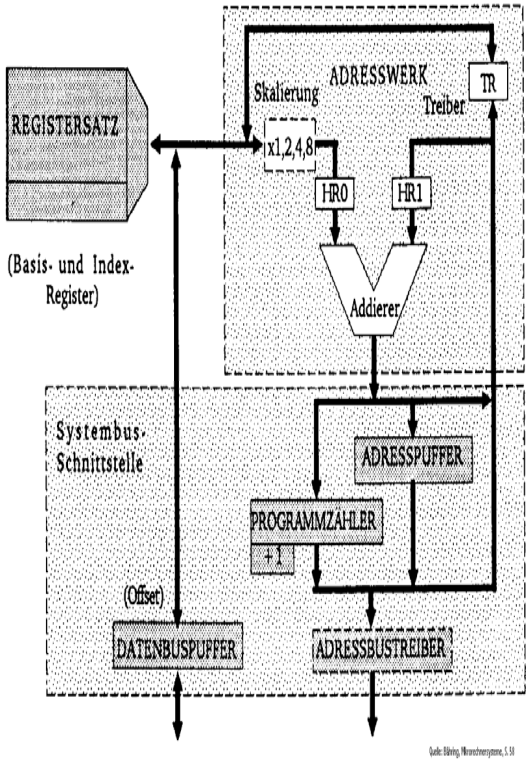
- Spezialregister
  - Statusregister
  - Programmzähler
  - Stackpointer
  - Segmentregister

Die Register im Intel 80x86

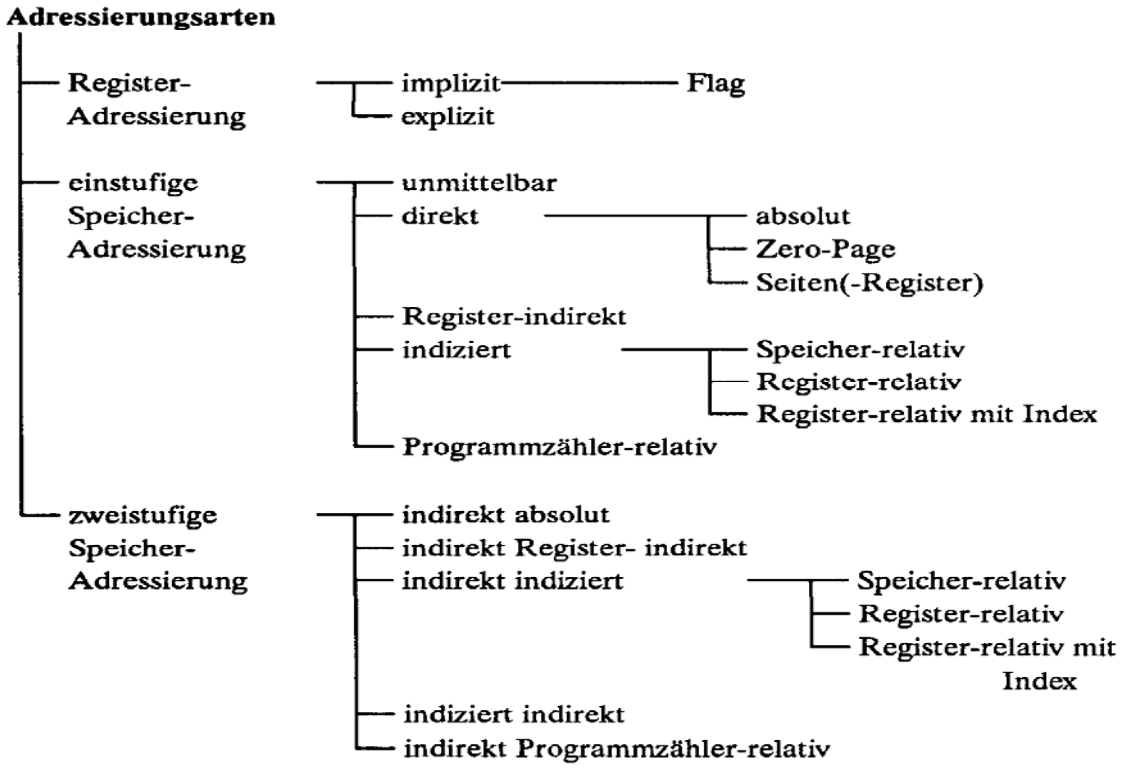
- AX (AH und AL)
  - accumlator
  - Akkumulator
- BX (BH und BL)
  - base register
  - Basisregister zur Adressierung der Anfangsadresse einer Datenstruktur
- CX (CH und CL)
  - count register
  - Schleifenzähler, wird bei Schleifen und Verschiebeoperationen benötigt
- DX
  - date register
  - Datenregister Register für den zweiten Operand
- SI und DI
  - source register und destination register
  - Indexregister für die Adressierung von Speicherbereichen
- SP
  - stack pointer
  - Verwaltung eines Stapelbereichs

Das Adresswerk

- Nach den Vorgaben des Steuerwerks werden Speicheradressen gebildet
  - ⇨ aus Registerinhalten
  - ⇨ aus Speicherzellen
- Adressaddierer
- TR-Register speichert den Inhalt des aktuellen Adresszählers bei Sprüngen
- Adressprüfung bei Byte-, Halbwort-, Doppelwort- und Quadwort-Zugriffen

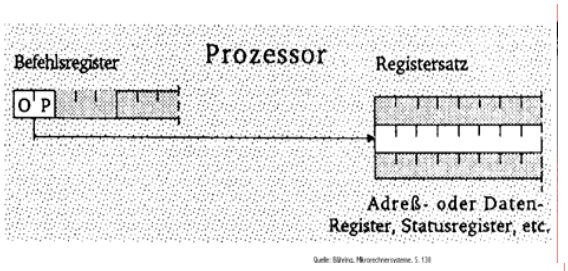


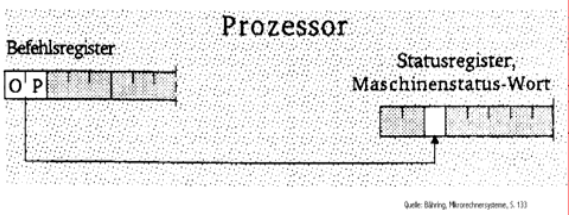
Adressierungsarten

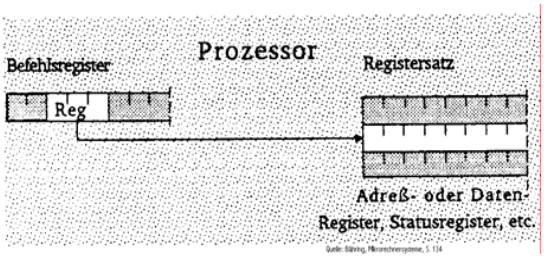


Register- Adressierung

- **Implizite Adressierung**
  - ⇨ **Adresse des Operanden ist im OP-Code enthalten**
  - ⇨ **Beispiel: `LSRA`**
    - **logical shift right accumulator**
- **Flag-Adressierung**
  - ⇨ **ein einzelnes Bit wird angesprochen**
  - ⇨ **Beispiel: `SEC`**
    - **set carry flag**
- **Explizite Adressierung**
  - ⇨ **Adresse des Operandenregisters wird im OP-Code angegeben**
  - ⇨ **Beispiel: `DEC r0`**
    - **decrement register 0**

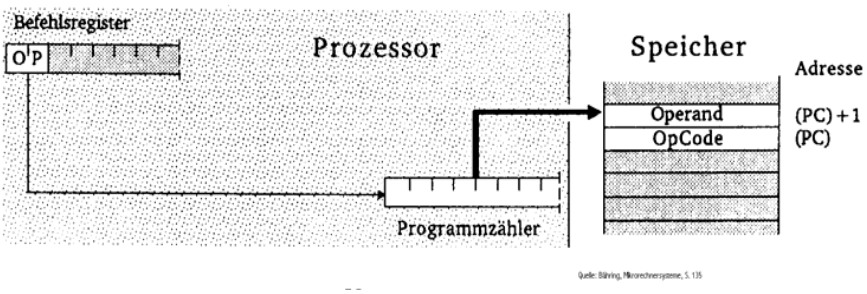






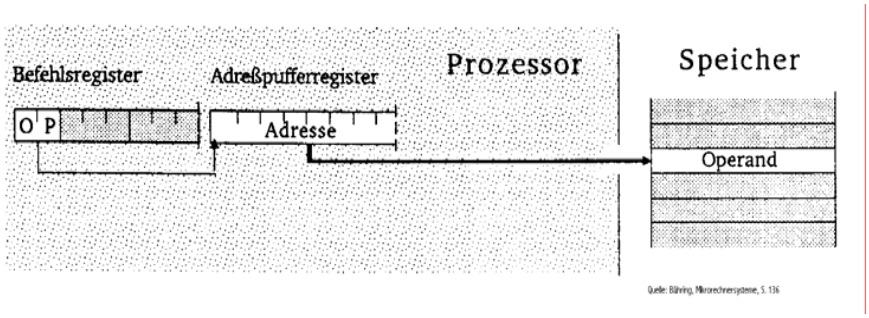
Unmittelbare Adressierung

- **Unmittelbare Adressierung**
  - ⇨ **Speicherwort das dem Befehl folgt enthält den Operanden**
  - ⇨ **Beispiel: `LDA #$A3`**
    - **load accu $A3_{16}$**



Direkte Adressierung

- **Absolute Adressierung**
  - ⇨ **Der Befehl enthält die Adresse des Operanden**
  - ⇨ **Beispiel: `JMP $07FE`**
    - **jump to $07FE**



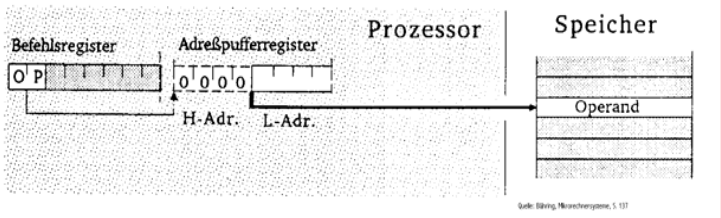
Direkte Adressierung

Weitere direkte Adressierungen: Seitenadressierung

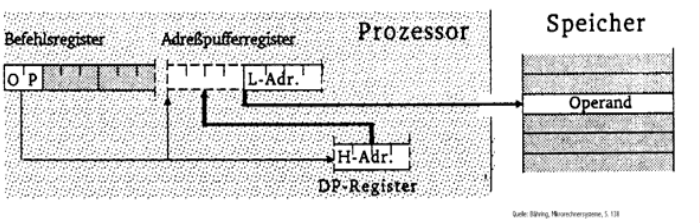
Bei Prozessoren mit unterschiedlicher Daten- und Adressbusbreite

- ⇨ **man spart Speicherplatz und Zeit des Lesens der höherwertigen Bits**

- **Zero-Page Adressierung**
  - ⇨ **schneller Zugriff auf die Speicherseite 0**
  - ⇨ **Beispiel: `INC $007F`**
    - **erhöhe Speicherzelle $7F um 1**

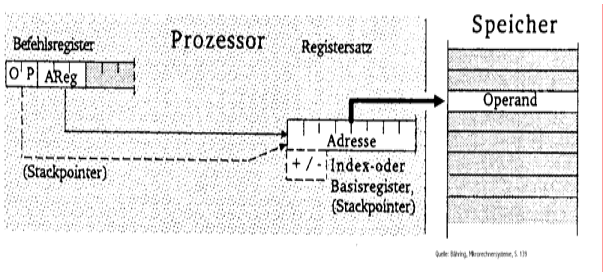


- **Seiten-Register-Adressierung**
  - ⇨ **Höherwertige Adressteil wird von einem Register zur Verfügung gestellt**
  - ⇨ **Beispiel: `LDA R0, <$7F`**
    - **Load accu mit Inhalt von Adresse, die sich aus Inhalt von R0 $7F zusammensetzt**



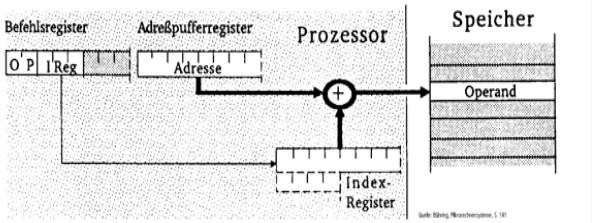
Register-Indirekte Adressierung

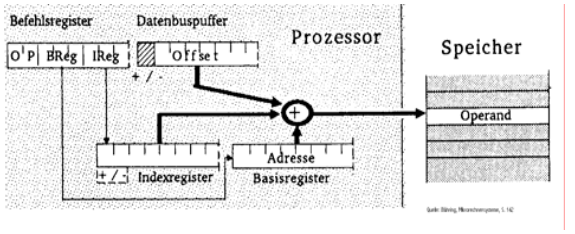
- **Register-Indirekte Adressierung (auch Zeigeradressierung)**
  - ⇨ **Der Inhalt eines Registers wird als Adresse des Operanden verwendet**
- **postincrement:** `LD R1,(R0)+`
  - ⇨ **Lade R1 mit dem Inhalt der Speicherzelle, auf die R0 zeigt, und inkrementiere anschließend R0**
- **preincrement:** `INC +(R0)`
  - ⇨ **Erhöhe zunächst das Register R0 um 1 und danach die Speicherzelle, auf die das neue R0 zeigt**
- **postdecrement:** `LD R1,(R0)-`
  - ⇨ **Lade R1 mit dem Inhalt der Speicherzelle, auf die R0 zeigt, und dekrementiere anschließend R0**
- **predecrement:** `CLR -(R0)`
  - ⇨ **Dekrementiere zunächst R0 und lösche die Speicherzelle, auf die das neue R0 zeigt**



Indizierte Adressierung

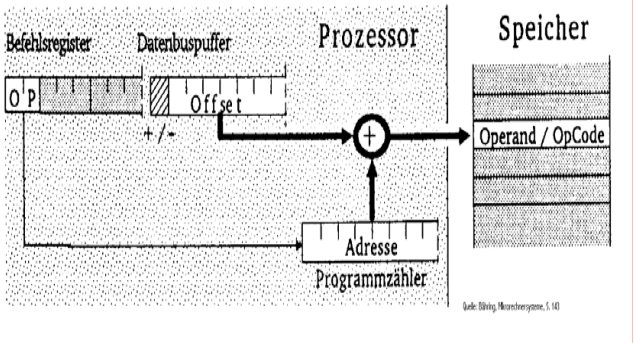
- **Speicher-relative Adressierung**
  - ⇨ **Der Basiswert, der zum Indexregister addiert wird, ist im Befehlswort enthalten**
  - ⇨ **Beispiel:** `ST R1, $A704(R0)`
    - **Speichere R1 an die Adresse, die sich aus der Summe des Inhalts des Registers R0 und $A704 ergibt**
- **Register-relative Adressierung mit Offset**
  - ⇨ **Der Basiswert befindet sich in einem speziellen Basisregister**
  - ⇨ **Der Inhalt des Indexregisters und ein Offset wird zum Basiswert addiert**
  - ⇨ **autoincrement und autodecrement**
  - ⇨ **Beispiel:** `ST R1, $A7(B0)(I0)+`
    - **Speichere R1 an Adresse, die sich durch Addition von B0, I0 und dem Offset ergibt, und inkrementiere I0 anschließend**





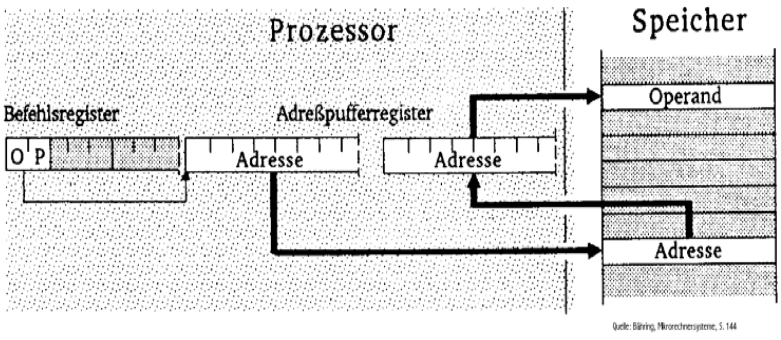
Programmzähler-relative Adressierung

- **Programmzähler-relative Adressierung**
  - **Der im Befehlscode angegebene Offset wird zum aktuellen Befehlszähler hinzuaddiert**
  - **Beispiel:** `BCS $47(PC)`
    - **Verzweige an die Adresse PC+$47 sofern das Carry-Flag gesetzt ist**



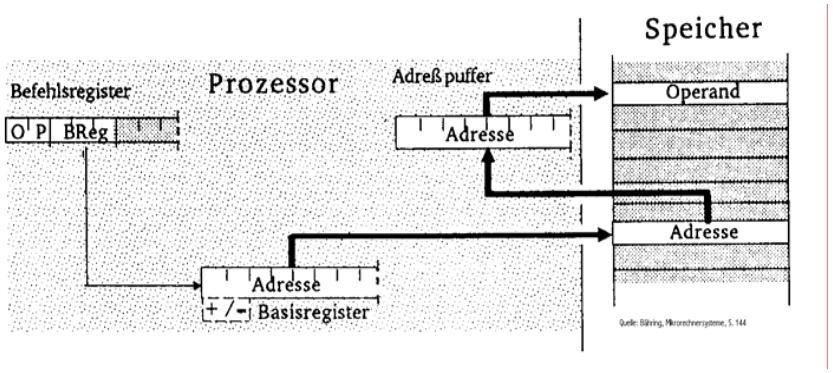
Zweistufige Speicheradressierung

- **Indirekte absolute Adressierung**
  - **Der Befehl enthält eine absolute Adresse, die auf ein Speicherwort zeigt. Dieses Speicherwort enthält die gesuchte Adresse**
  - **Beispiel:** `LDA ($A345)`
    - **Lade den Accu mit dem Inhalt des Speicherworts, dessen Adresse in $A345 steht**



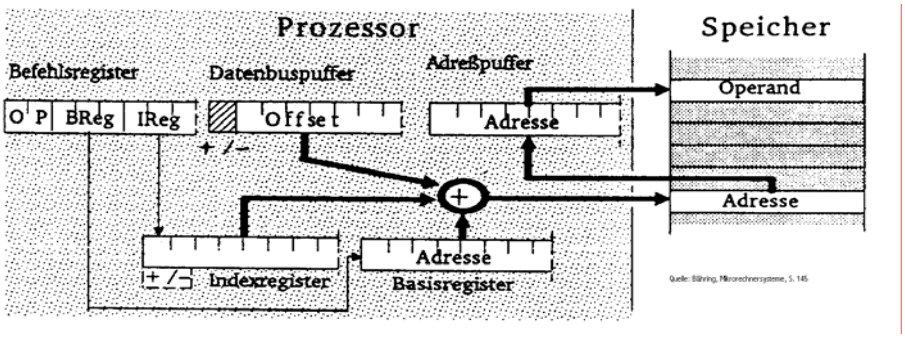
Zweistufige Speicheradressierung

- **Indirekte Register-indirekte Adressierung**
  - **Der Befehl bezeichnet ein Register, dessen Inhalt die Speicherzelle ist, deren Inhalt als Adresse für das Speicherwort verwendet wird**
  - **Beispiel:** `LD R1, ((R0))`
    - **Lade R1 mit dem Inhalt der Adresse, die in der Speicherzelle steht, auf die R0 zeigt**



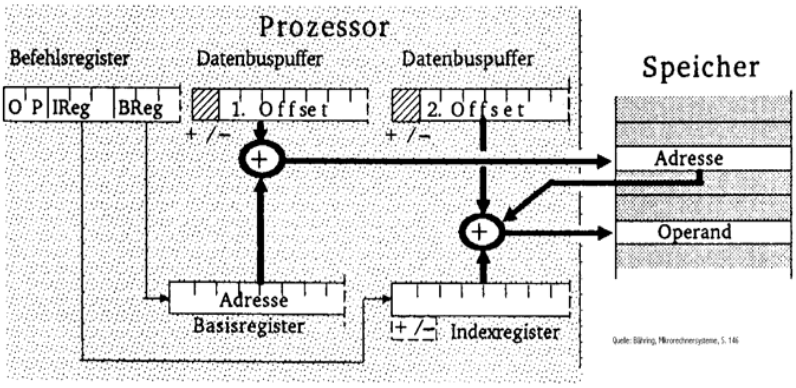
Zweistufige Speicheradressierung

- **Indirekte indizierte Adressierung**
  - **Die Adresse des Speicherworts wird aus der Summe von Offset, Basisregister und Indexregister gebildet. Dieses Speicherwort enthält die Adresse des Ziels**
  - **Beispiel:** `INC ($A7(B0)(I0))`
    - **Erhöhe die Speicherzelle mit der Adresse $A7+B0+I0 um 1**



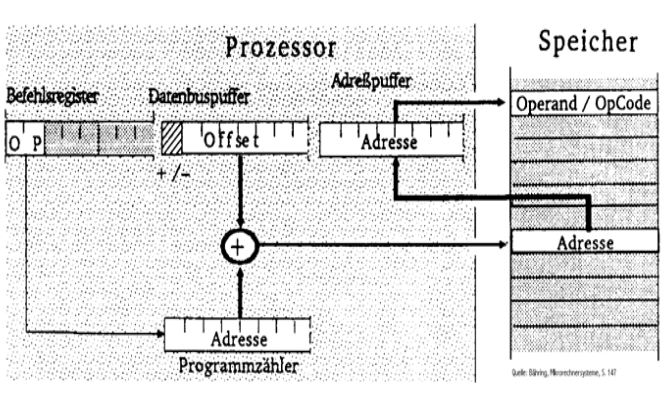
Zweistufige Speicheradressierung

- **Indizierte indirekte Adressierung**
  - **Die Adresse des Speicherworts wird aus dem 1. Offset und dem Basisregister gebildet. Der Inhalt dieses Speicherworts wird zum Indexregister und dem 2. Offset hinzuaddiert und bildet die Adresse des gesuchten Speicherworts**
  - **Beispiel:** `INC $A7($10(B0))(I2)`
    - **Addiere den Offset $10 zum Inhalt des Basisregisters. Der Inhalt dieser Speicherzelle plus Indexregister und zweiter Offset $A7 ergibt den Wert der gesuchten Adresse**



Zweistufige Speicheradressierung

- **Indirekte Programmzähler-relative Adressierung**
  - **Die Summe aus Programmzähler und Offset ergibt die Adresse, die auf das Ziel zeigt**
  - **Beispiel:** `JMP ($A7(PC))`
    - **Springe an die Stelle die im Speicherwort mit der Adresse PC plus $A7 steht.**



Speicherhierarchie

	Zugriffszeit (ns)	Größe (MB)	Kosten ($/MB)
→ auf dem Prozessorchip:			
Register	0.5	0.0005	
L1 Cache	2.0	0.05	100
→ extern (oder intern):			
L2 Cache	6.0	1	30
Hauptspeicher	100	1000	1
Sekundärspeicher (Festplatte)	10.000.000	100.000	0.05

Wenn L2-Cache auf dem Prozessorchip, oft (extern): L3-Cache (z.B. 32 MB)

Speicherhierarchie

Ziel: Schaffe hohe Wahrscheinlichkeit dafür, dass ein Datum sich im Cache befindet, wenn darauf zugegriffen wird (d.h. hohe Hit/Miss-Rate)

Wenn Datum in den Cache geladen werden soll und kein freier Platz vorhanden ist, wird ein anderes Element gelöscht.

Strategien für die Auswahl des zu löschenden Elementes:

- **Zufallsauswahl:**
  - ⇨ **Wähle zufällig ein Datum aus dem Cache aus**
- **Least Recently Used (LRU):**
  - ⇨ **Ersetze das Datum auf welches am längsten nicht zugegriffen wurde**
  - ⇨ **Erfordert zusätzliche Hardware, um die Reihenfolge bzgl. letzter Zugriffszeit zu verwalten**
- **First In First Out (FIFO):**
  - ⇨ **Ersetze das Datum welches sich am längsten im Cache befindet**
  - ⇨ **Erfordert zusätzliche Hardware, um die Zugriffszeiten zu verwalten**

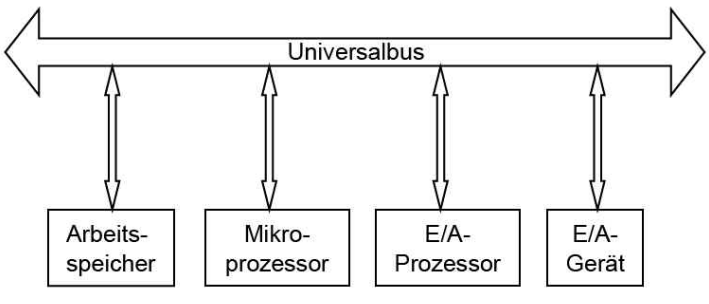
Speicherhierarchie

Strategien beim Schreiben eines Datums in den Cache:

- **Write back:**
  - Schreiben auf ein Datum im Cache verändert nur die Kopie im Cache
  - Erst beim Verdrängen aus dem Cache wird das Datum in den Hauptspeicher geschrieben, wenn es verändert wurde (dirty bit)
- **Write through:**
  - Bei jedem Schreiben in den Cache wird das Datum auch in den Hauptspeicher geschrieben
- **Cache miss:**
  - **Fetch on write**: Block wird in den Cache geladen und nach einer der beiden obigen Strategien geschrieben (meist bei Write back)
  - **Write around**: Das Datum wird nur im Hauptspeicher geschrieben und nicht in den Cache geladen (meist bei Write through)

8 Rechner- und Gerätebusse

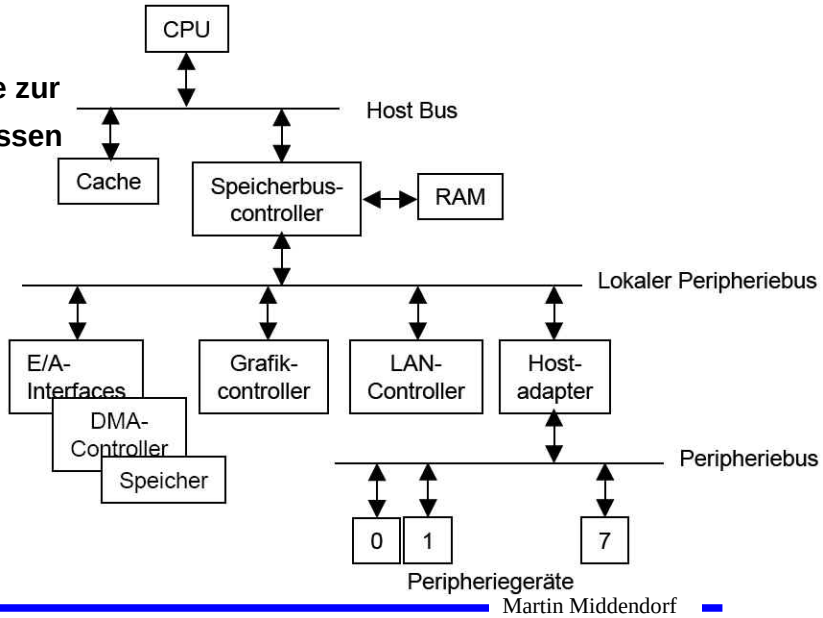
- **Busse** verbinden Komponenten eines Rechnersystems
- Probleme eines **Einbussystems**:
  - unterschiedliche Arbeitsgeschwindigkeiten der Geräte
  - unterschiedliche Breiten der auszutauschenden Daten (Adressen (8-64 Bit), Signale, Daten (16-64 Bit))
  - nur eine Komponente kann zu einer Zeit den Bus verwenden
  - Anschluss vieler Komponenten senkt die maximal mögliche Arbeitsgeschwindigkeit



Ein-Bus-System

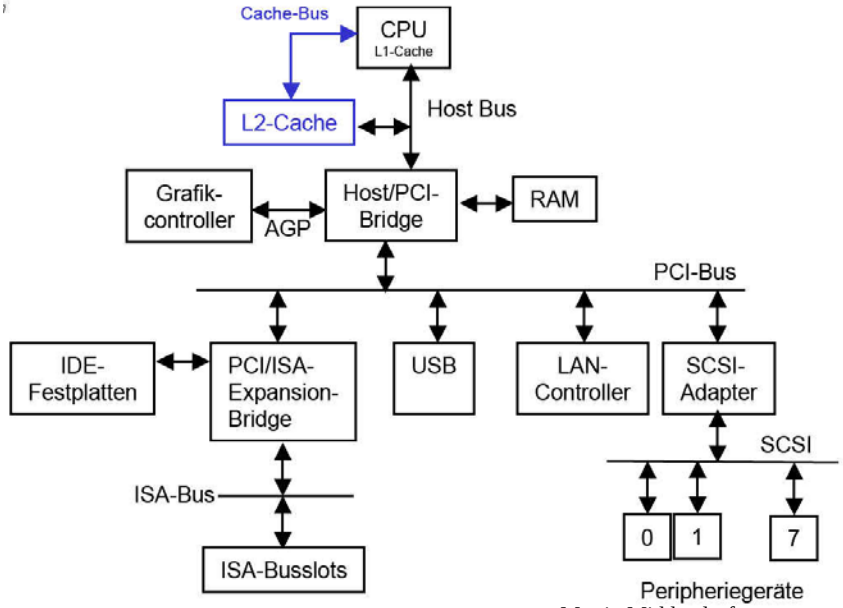
PC-Busstrukturen

- Heutige PC-Bussysteme besitzen mindestens:
  - ⇨ Systembus (unterteilt in Hostbus/Prozessorbus (Übertragungsrate >1000 MB/s) und Speicherbus (Übertragungsrate mehrere 100 MB/s),
  - ⇨ lokalen Peripheriebus
  - ⇨ Peripheriebus
- Controller/Adapter/Bridge zur Verbindung zwischen Bussen



PC-Busstrukturen

- Weitere spezielle Busse dienen der Anbindung von
  - ⇨ Cache-Speicher (Cache-Bus)
  - ⇨ Graphikkarten (AGP-Bus, Accelerated Graphics Port)
  - ⇨ unterschiedlichen Peripheriegeräten



Rechner- und Gerätebusse

- lokale Peripheriebusse (Rechnerbusse)
  - Busse, die rechnerinterne Komponenten verbinden
  - AT-Bus PC/XT (8088/ 8086)
  - ISA-Bus AT (80286)
  - EISA 80386 und 80486
  - VESA ab 80486
  - PCI ab 80486 bis Pentium4

- Peripheriebusse (Gerätebusse)
  - Busse, die externe Komponenten mit einem Rechnersystem verbinden
  - IEC Gerätebus
  - EIDE Festplatten
  - SCSI Geräte und Festplattenbus

Rechner- und Gerätebusse

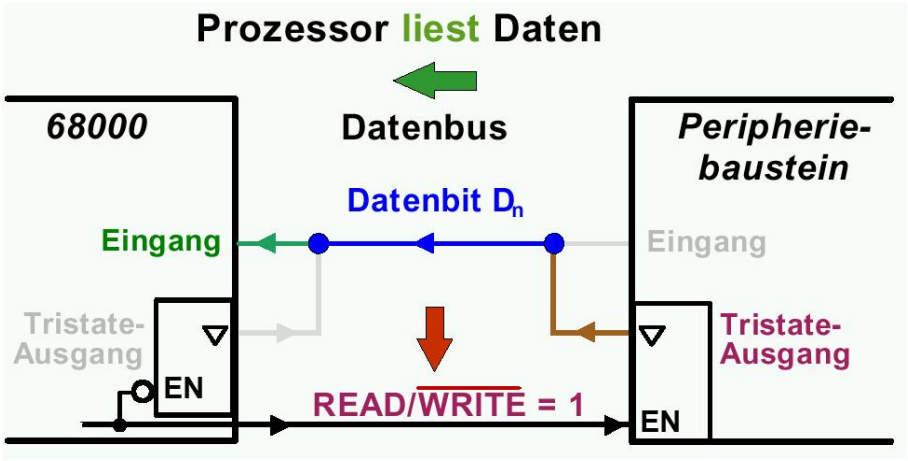
- Anforderungen an Busse
  - Unterschiedliche Übertragungsraten
  - Unterstützung von Lokalität durch das Bussystem (z.B. CPU greift meist auf den Cache zu)
  - Kompatibilität zu anderen Bussystemen
  - Leichte Erweiterbarkeit auch für den Anwender

Notwendig/Sinnvoll: Hierarchische Bussysteme

- Erhöhung der Übertragungsrate
  - Höhere Taktrate (zu hohe Taktrate ist nicht sinnvoll)
  - Erhöhung der Wortbreite
  - Split-Cycle-Übertragung (Bus wird freigegeben zwischen Senden der Adresse und der Daten)

Bidirektionaler Bus

- Realisierung mit Tristate-Ausgängen, die durch Signal gesteuert werden.



- READ/$\overline{\text{WRITE}}$=1
  - ⇨ deaktiviert den Tristate-Ausgang des Prozessors (legt ihn auf hochohmig), so dass er keinen Einfluss auf den Bus hat
  - ⇨ aktiviert den Tristate-Ausgang des Speichers, der Low oder High anlegt

PC/XT-/ISA-/EISA-Bus

- Der PC/XT-Bus
  - ⇨ Systembus
  - ⇨ 8 Bit Daten und 20 Bit Adressen
  - ⇨ Zugriffe mit max. 8 MHz
  - ⇨ 16-Bit-Zugriffe beim XT mussten auf 2 8-Bit-Zugriffe abgebildet werden

- Der ISA-Bus
  - ⇨ Industrial Standard Architecture
  - ⇨ 16 Bit Daten und 24 Bit Adressen
  - ⇨ Zugriffe mit max. 8,33 MHz
  - ⇨ Karten für den XT-Bus konnten weiter verwendet werden
  - ⇨ Heute als Ergänzung zum PCI-Bus für einfache Erweiterungskarten

- Extended ISA
  - ⇨ 32 Bit Daten und 32 Bit Adressen
  - ⇨ Zugriffe mit max. 8.33 MHz
  - ⇨ Steckplatz ist kompatibel zu ISA Steckkarten

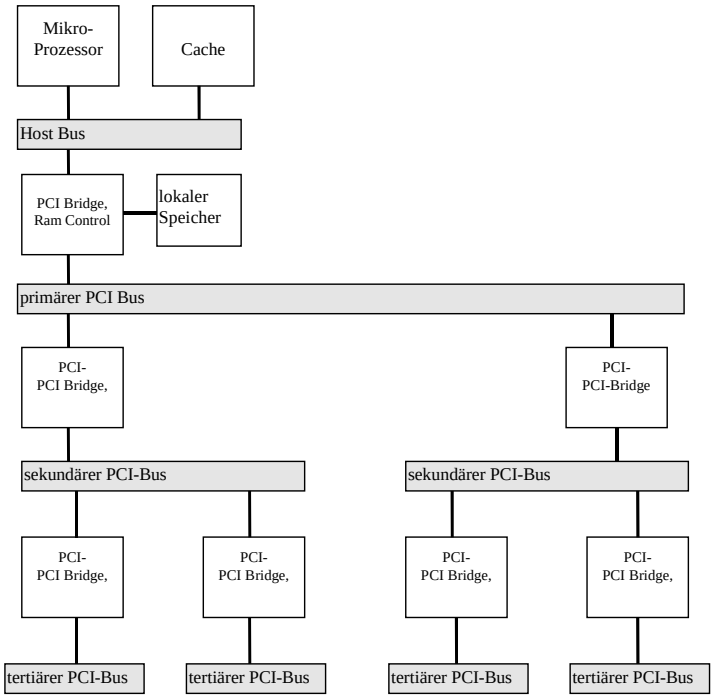
Der PCI-Bus

Peripheral Component Interconnect Bus

- Entkopplung von Prozessor und Erweiterungsbus durch eine Bridge
- 32-Bit-Standardbusbreite mit maximal 133MByte/s Transferrate
  Erweiterung auf 64 Bit mit maximal 266MByte/s Übertragungsrate
- Betriebsfrequenz: 0 – 33 MHz (Version 2.1 66MHz)
- Unterstützung von 5V- und 3,3V-Versorgungsspannungen
- prozessorunabhängige Spezifikation
- Unterstützung von Mehrprozessorsystemen
- Multimaster-Fähigkeit
  - ⇨ Prioritäts-/Zufallsverfahren entscheidet, wenn mehrere Devices Master werden wollen
  - ⇨ Watch-Dog-Timer entzieht einem Master nach bestimmter Zeit die Kontrolle, wenn andere Master werden wollen
- Bis zu 10 Devices anschließbar
- Da Bridge wie ein Device behandelt wird, können PCI-Busse kaskadiert zusammengeschaltet werden (bis zu 256)

Der PCI-Bus

Kaskadiert verbundene PCI-Busse



Der PCI-Bus

- Es existieren Host-PCI-Bridges, die mehrere PCI-Busse gleichzeitig an einen Host ankoppeln
  - ⇨ Beispiel: Apple PCI-Bridge 'Bandit' koppelt PowerPC-Prozessorbus an vier eigenständige PCI-Busse

- zeitlich gemultiplexter Adress- und Datenbus
  - ⇨ dadurch geringe Pin-Anzahl

Standard-Transfer: Es wird zunächst die Adresse und dann das Datum übertragen

Burst-Transfer: Übertragung von Adresse und dann Block beliebiger Länge von im Speicher aufeinander folgender Daten
- ⇨ Bridge fasst Daten wenn möglich selbständig zu Bursts zusammen
- ⇨ Latenztimer ermöglicht Unterbrechung
- ⇨ Übertragungsrate und Latenzzeit steigen bei zunehmender Blockgröße

Empfänger (Target) bricht Transfer nach aktuellem Datenwort ab, wenn bis zur Bereitstellung des nächsten Datenworts >8 Taktzyklen nötig sind

Der PCI-Bus

- Lesetransfer: **Wartezyklus** zwischen Adresse und Datenblock (unterschiedliche Quellen für Adresse (Initiator) und Daten (Target))

Anzahl der notwendigen Taktzyklen:

32-Bit Bus	4 Langworte	128 Langworte	2^{30} Langworte
Standard Read	12	384	$2^{31} + 2^{30}$
Standard Write	8	256	2^{31}
Burst Read	6	130	$2^{30} + 2$
Burst Write	5	129	$2^{30} + 1$

- Maximale **Datentransferraten** des PCI-Bus:

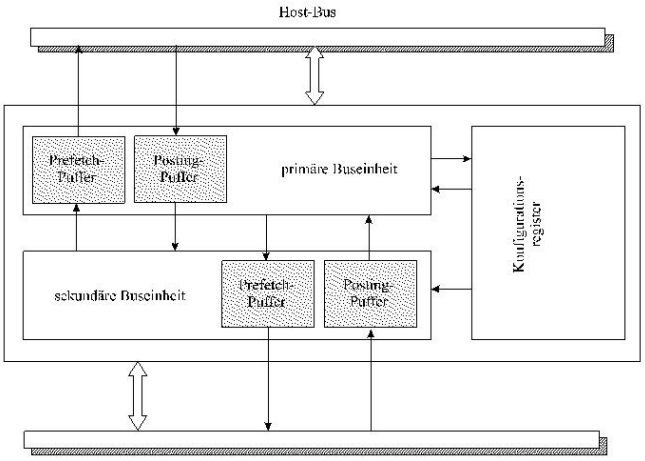
Transferrate = Anzahl Bytes / (Buszykluszeit * Anzahl Zyklen)

Buszykluszeit von 30ns (33 Mhz) ergibt folgende Transferraten (in Mbyte/s):

32-Bit Bus	4 Langworte	128 Langworte	2^{30} Langworte
Standard Read	44,43	44,43	44,43
Standard Write	66,67	66,67	66,67
Burst Read	88,89	131,28	133,33
Burst Write	106,67	132,30	133,33

Der PCI-Bus

- **Write Posting und Read Prefetching**
- **Fehlererkennung:**
  Parity-Fehlererkennung,
  Retry-Option, Timeouts
- **Unterstützung für**
  ISA-/EISA-/MCA-Busse
- **Konfigurierung über Software,**
  Register
- **Nachfolger: PCI-X Bus bis 133 MHz**
  (Version 2 PCI-X 266 bis 2,1 GByte/s)

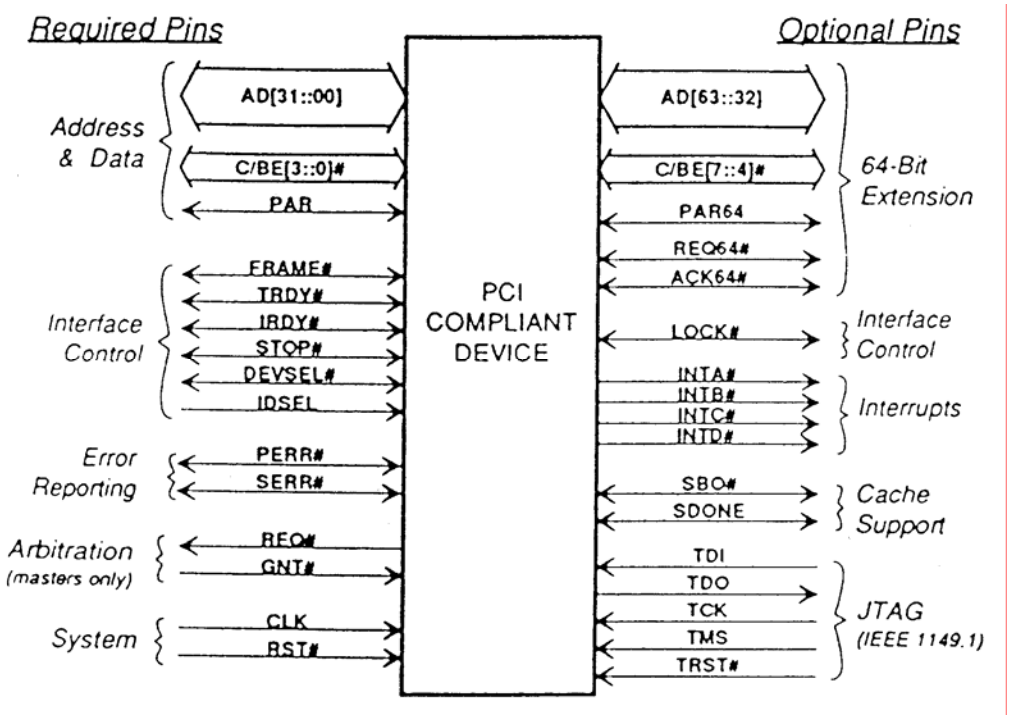


Bezeichnungen:

Agent: **jeder Busteilnehmer**
Target: **PCI-Agent, der auf Transaktionsanforderung antwortet [Slave]**
Initiator: **PCI-Agent, der Transaktionszyklus anfordert und kontrolliert (Steuersignale) [Master]**
Arbiter: **Einheit, die zentral Busbelegungsrechte vergibt (oft auf dem Mainboard)**

Der PCI-Bus

- **Signale des PCI-Bus:**



Der PCI-Bus

Signal	Typ	Beschreibung
AD[31:0]	t/s	**Adress/Datenbus**
C/BE#[3:0]	t/s	**Command/ByteEnable** identifiziert Transfertyp (während Adressphase) oder gibt Datenbytes frei (während Datenphase)
PAR	t/s	**Parität**, gerade Parität über AD[31:0] und C/BE#[3:0]
FRAME#	s/t/s	**Frame**, kennzeichnet Start und Länge eines Transfers
TRDY#	s/t/s	**Target Ready**, kennzeichnet die Bereitschaft des Target, den aktuellen Datentransfer abzuschliessen
IRDY#	s/t/s	**Initiator Ready**, kennzeichnet die Bereitschaft des Initiators (Master), den aktuellen Datentransfer abzuschliessen
STOP#	s/t/s	**Stop**, durch aktivieren dieses Signals kann ein Target einen vom Initiator gestarteten Datentransfer stoppen

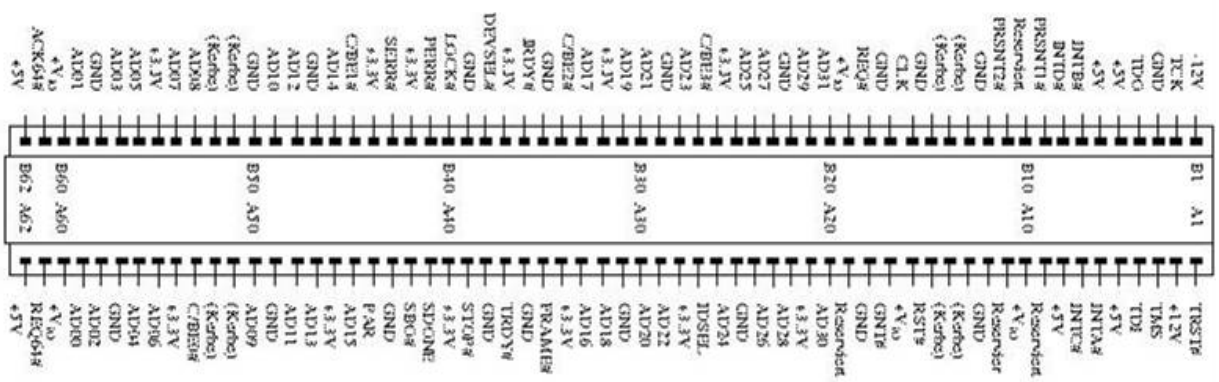
Der PCI-Bus

Signal	Typ	Beschreibung
DEVSEL#	s/t/s	**Device Select**, bestätigt die Adressdekodierung durch ein Target
IDSEL	in	**Initialization Device Select**, selektiert einen Device während der Konfigurationsphase
PERR#	s/t/s	**Parity Error**, signalisiert das Auftreten eines Parity-Errors auf AD[31:0] oder C/BE#[3:0]
SERR#	o/d	**System-Error**, signalisiert das Auftreten eines katastrophalen Fehlers (z.B. Parity-Error während der Adressierungsphase)
REQ#	t/s	**Request**, Bus-Anforderung an den zentralen Arbiter durch einen Master
GNT#	t/s	**Grant**, Bus-Gewährung durch den Arbiter
CLK	in	**PCI System Takt**, 0 .. 33MHz
RST#	in	**System Reset**, rücksetzen aller PCI-Devices

kennzeichnet ein Aktiv-Low-Signal in: einfaches Eingangssignal
t/s: bidirektionaler Tri-State-Ausgang mit Eingangsport. Parallelschaltung mehrerer Treiber möglich
s/t/s: sustained Tristate - activ low, idle high, höchstens ein Treiber, ein Turn-Around ist immer nötig.
o/d open-drain Signal (active low, idle high), langsames Signal (wired-OR, mehrere können gleichzeitig Schreiben)

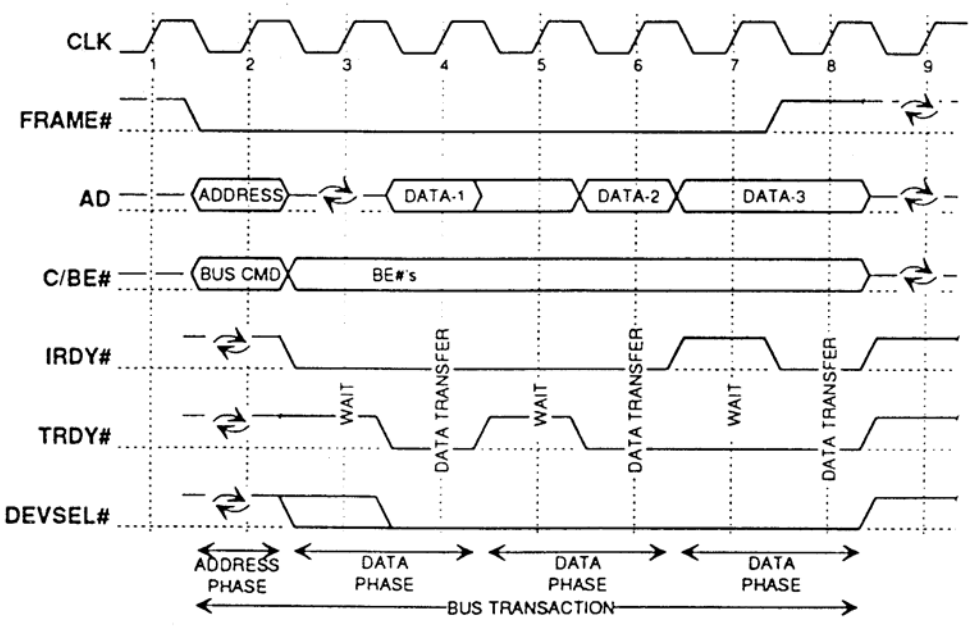
Der PCI-Bus

Steckerbelegung des 32-Bit Steckers:



Der PCI-Bus

Beispiel eines Lesezyklus:



Adressphase:

Lesezyklus beginnt in Takt 2 durch Aktivierung von FRAME#

Zwischen Adress- und Datenphase ist hier ein Turnaround erforderlich, da die Adresse vom Initiator, die Daten aber vom Target kommen

Datenphase:

Initiator und Target lösen verschiedene Wait-States aus.

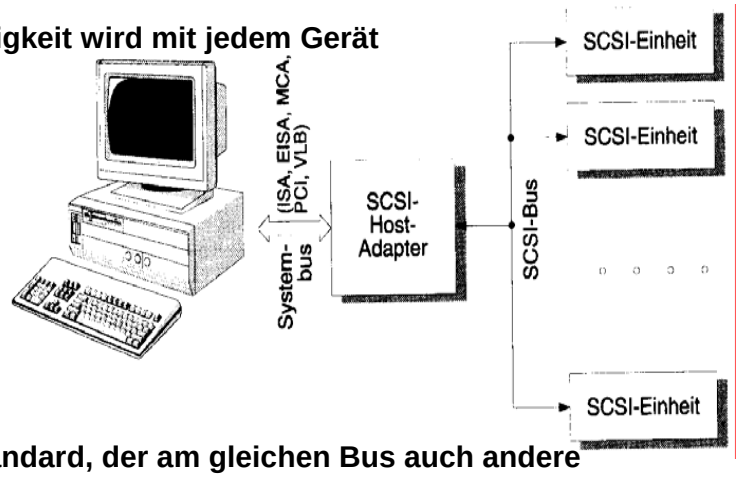
= Turnaround: Wartezyklus, zur Vermeidung von Kollisionen bei Ab- und Umschaltvorgängen (Adress/Datenmultiplex, abwechselnde Aktionen von verschiedenen Initiatoren)

Peripheriebusse

- Verbindung peripherer Geräte untereinander, Anschluß an den Systembus über eine Steuereinheit
  - ⇨ Deutlich langsamer als der Systembus
  - ⇨ Dafür flexibler, größere Leitungslängen, weniger Leitungsaufwand
- Beispiele: SCSI, USB

SCSI-Bus

- Small Computer Systems Interface (SCSI)
  - ⇨ Maximal 8 Einheiten (inkl. Host)
  - ⇨ Identifikation durch SCSI-ID (Adresse muss per Schalter eingestellt werden)
  - ⇨ Übertragungsgeschwindigkeit wird mit jedem Gerät ausgehandelt
  - ⇨ 8 Bit Übertragung



- Weitere SCSI-Standards
  - ⇨ SCSI-II: Erster richtiger Standard, der am gleichen Bus auch andere Geräte außer Festplatten berücksichtigt
  - ⇨ Fast SCSI: maximale Taktfrequenz wurde auf 10 MHz erhöht
  - ⇨ Wide SCSI: 16 Bit und 32 Bit Erweiterung der Datenbreite

USB-Bus

Universal Serial Bus (USB)

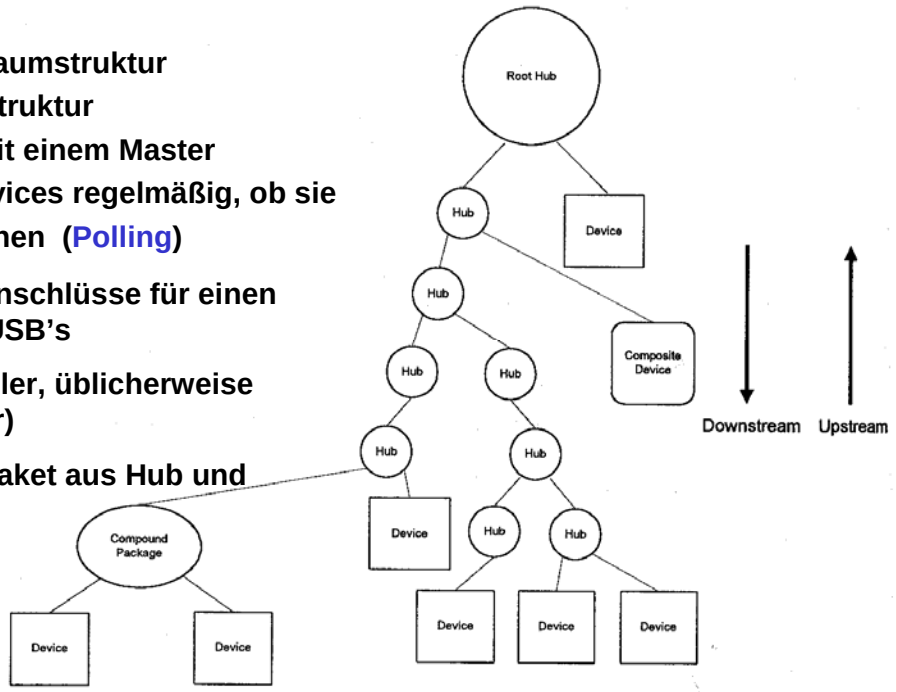
- Ziel: einfache Verbindung vieler peripherer Geräte über seriellen Bus
- Topologie
  - ⇨ Physikalisch: Baumstruktur
  - ⇨ Logisch: Sternstruktur
- Master-Slave Bus mit einem Master
  - ⇨ Master fragt Devices regelmäßig, ob sie Transfer wünschen (Polling)

***Hub*: Verteiler, bietet Anschlüsse für einen oder mehrere weitere USB's**

***Root-Hub*: erster Verteiler, üblicherweise der PC (einziger Master)**

***Compound Package*: Paket aus Hub und einigen Devices**

***Device*: Endgerät**

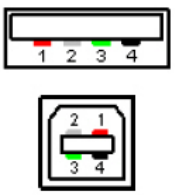


USB-Bus

Physikalisches Übertragungsprotokoll:

- ⇨ serielle, bitorientierte,
- ⇨ synchrone,
- ⇨ halbduplex-Datenübertragung
- ⇨ 4-Draht Leitung: 2 Daten, GND, Power
- ⇨ Leitungslänge bis 5m (ohne Hub)
- ⇨ Stromversorgung einfacher Geräte über den Bus

Pinbelegung:



Pin	Farbe	Symbol	Funktion
1 (länger)	rot	V_{BUS}	5V
2	weiss	$D-$	Datenleitung
3	grün	$D+$	Datenleitung
4 (länger)	schwarz	GND	Masse

- Zwei Signalzustände *J* und *K* → für Full Speed und Low Speed Übertragung invers definiert (differentielles Signal):
  Full Speed: *J:* VD+ > VD- *K:* VD+ < VD-

USB-Bus

- Drei Übertragungsgeschwindigkeiten (Device muß nicht alle unterstützen):
  - ⇨ *High Speed* = 480 MBits/sec (USB 2.0)
  - ⇨ *Full* Speed = 12 MBit/sec
  - ⇨ *Low Speed* = 1,5 MBits/sec

- Datencodierung mittels "*Non Return To Zero Inverted*" (NRZI) Verfahren:
  - ⇨ Gleichbleibender Signalzustand (J J oder K K) = logische 1
  - ⇨ Wechselnder Signalzustand (J K oder K J) = logische 0

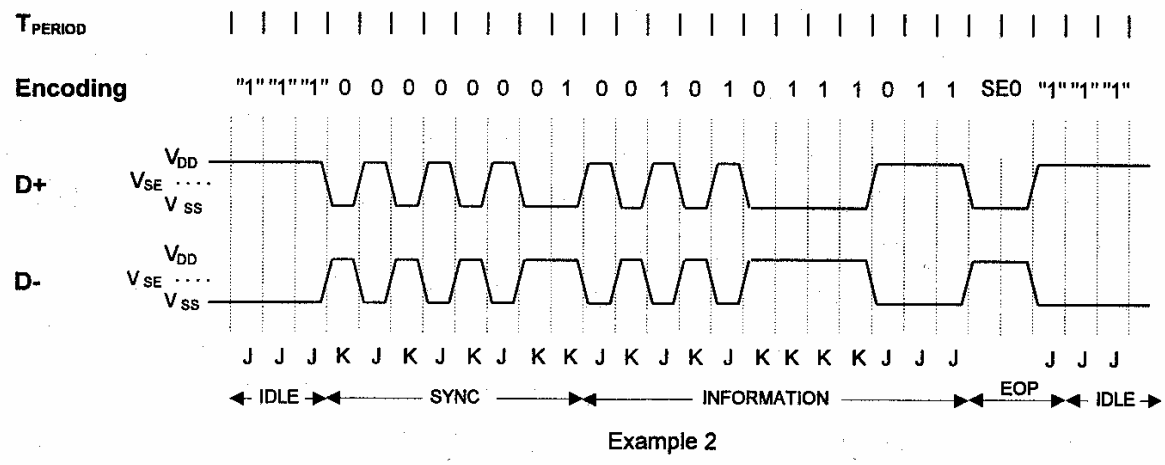
- Problem bei NRZI: lange 1-Folgen erzeugt konstantes Signal
  ⇒ Taktsynchronität gefährdet

  Abhilfe: *Bit Stuffing:*
  Nach 6 aufeinander folgenden 1-Bits wird zwangsweise ein 0-Bit eingeführt (welches bei der Dekodierung dann ignoriert wird)
  → erzwungener Pegelwechsel

- Synchronisation:
  - ⇨ Synchronisationsphase zu Beginn jeder Informationsübertragung
    → mehrfacher Wechsel zwischen J und K
  - ⇨ Takt für die weitere Übertragung kann ermittelt und synchronisiert werden

USB-Bus

- Beispiel einer Übertragung (Full Speed)



Example 2

USB-Bus

Logisches Übertragungsprotokoll

- USB Device wird durch 7-Bit logische Adresse identifiziert
  - ⇨ Adresse 0 ist reserviert
  - ⇨ Root-Hub benötigt eine Adresse
  - ⇨ 126 Adressen stehen für weitere Devices zur Verfügung

- Neues Gerät wird von Hub erkannt und an den Root-Hub gemeldet.
  - ⇨ Von dort Adresszuweisung

- Verschiedene Arten des Datentransfers
  - ⇨ Interrupt-Transfer: für periodisches Übertragen kleiner Datenmengen (Maus,Tastatur, Low-Speed), garantierte Übertragungszeit (Wert kann vom Gerät gewünscht werden, bei Fehler Wiederhloung in nächster Periode
  - ⇨ Isochroner-Transfer: für Audio-/Videodaten, garantierte maximale Latenzzeit und Bandbreite, keine Sendewiederholung bei Fehler, kein Handshake
  - ⇨ Bulk-Transfer: Übertragung großer Datenmengen (Drucker, Scanner) ohne Garantie für Übertragungsdauer

USB-Bus

- Datenübertragung erfolgt in Form von Paketen
  - ⇨ Transaktion besteht aus
    - Token-Paket
    - Daten-Paketen
    - Handshake-Paket

USB-Bus

- **Token:** kennzeichnet Typ der Transaktion, enthält Device Adresse und Endpunkt Nummer
  - ⇨ **Endpunkt**: Jedes Device kann mehrere Endpunkte für unabhängige Datenübertragungen einrichten

  Typen:
  - ⇨ **OUT:** Datenübertragung vom Host zum Device
  - ⇨ **IN:** Datenübertragung vom Device zum Host
  - ⇨ **SETUP:** Übertragung von Konfigurationsdaten
  - ⇨ **FRAME:** Hochgeschwindigkeits Blockdatenübertragung (nur für Full Speed Devices, Dauer jedes Frame: 1 sec)

- **Daten:** enthält die eigentlichen Daten einer Transaktion

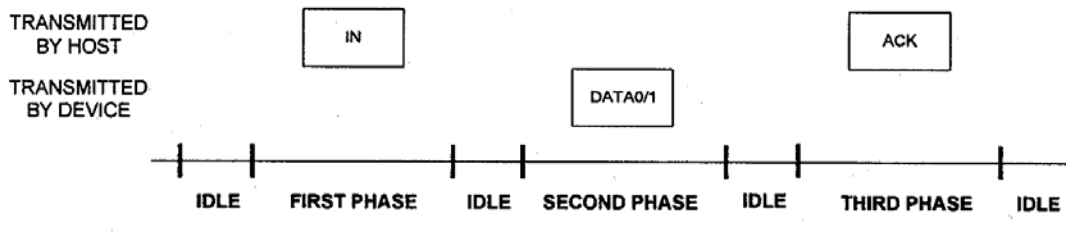
  Datenpakete besitzen folgende Felder:
  - Typ
  - Länge
  - Inhalt
  - CRC (Cyclic Redundancy Check): zur Fehlererkennung, erkennt 1 und Doppelbitfehler

USB-Bus

- **Handshake:** enthält Informationen über den Datenfluss

  Mögliche Handshake-Informationen:
  - ⇨ **ACK:** kennzeichnet erfolgreichen Datentransfer
  - ⇨ **NACK:** kennzeichnet fehlgeschlagenen Datentransfer
  - ⇨ **STALL:** kennzeichnet ernsten Fehler im Device, der eine Host-Interaktion (z.B. Neukonfiguration) erforderlich macht, bevor die Datenübertragung fortgesetzt werden kann

- **Beispiel: erfolgreiche Datenübertragung vom Device zum Host**



USB-Bus

- **Deskriptoren dienen der Beschreibung von Geräten und zugehörigen Übertragungsarten**

- **Device-Descriptor: enthält Informationen über Device, Identität des Gerätes**
- **Configuration-Descriptor: mehrere pro Device möglich, immer nur eine aktiv**
- **Interface-Descriptor: mehrere können aktiv sein**
- **Endpoint-Descriptor:**
- **String-Descriptor: enthält menschenlesbare Information über das Gerät**

USB-Bus

- **Device-Decriptor:**

Offset	Bezeichnung	Länge	Beschreibung	z.B.
0	bLength	1	Länge des Descriptors	0x12
1	bDescriptorType	1	Typ = Device Descriptor	0x01
2	bcdUSB	2	USB-Version 1.1	0x0110
4	bDeviceClass	1	Geräte-Klassen-ID	0x00
5	bDeviceSubClass	1	Geräte-Unterklassen-ID	0x00
6	bDeviceProtocol	1	Geräte-Protokoll-ID	0x00
7	bMaxPacketSize0	1	max. Paketlänge auf EP0	0x08
8	idVendor	2	Hersteller-ID	0x6666
10	idProduct	2	Produkt-ID	0x2342
12	bcdDevice	2	Produkt-Version	0x0100
14	iManufacturer	1	String-Nr. Hersteller-Name	0x01
15	iProduct	1	String-Nr. Produkt-Name	0x02
16	iSerialNumber	1	String-Nr. Seriennummer	0x00
17	bNumConfigurations	1	Anzahl Konfigurationen	0x01

USB-Bus

- **Configuration-Descriptor:**

Offset	Bezeichnung	Länge	Beschreibung	z.B.
0	bLength	1	Länge des Descriptors	0x09
1	bDescriptorType	1	Typ = Configuration Descriptor	0x02
2	wTotalLength	2	Gesamtlänge der Konfiguration	0x19
4	bNumInterfaces	1	Anzahl der Interfaces	0x01
5	bConfigurationValue	1	Nr. dieser Konfiguration	0x01
6	iConfiguration	1	String-Nr. dieser Konfiguration	0x00
7	bmAttributes	1	Bit 7: Stromversorgung über USB	0x80
			Bit 6: eigene Stromversorgung	
			Bit 5: unterstützt Remote-Wakeup	
8	MaxPower	1	Stromaufnahme in $2mA$-Einheiten	0xC8

Zu einer Konfiguration gehören die Interface-Descriptor und die Endpoint-Deskriptor

USB-Bus

- **Interface-Descriptor:**

Offset	Bezeichnung	Länge	Beschreibung	z.B.
0	bLength	1	Länge des Descriptors	0x09
1	bDescriptorType	1	Typ = Interface Descriptor	0x04
2	bInterfaceNumber	1	Nr. des Interfaces	0x00
3	bAlternateSetting	1	Nr. der Einstellung	0x00
4	bNumEndpoints	1	Anzahl Endpoints	0x01
5	bInterfaceClass	1	Interface-Klassen-ID	0x00
6	bInterfaceSubClass	1	Interface-Unterklassen-ID	0x00
7	bInterfaceProtocol	1	Interface-Protokoll-ID	0x00
8	iInterface	1	String-Nr. Interface-Name	0x00

Ein Interface kann mehrere Einstellungen besitzen: z.B. Audio-Übertragung mit mehreren Geschwindigkeiten

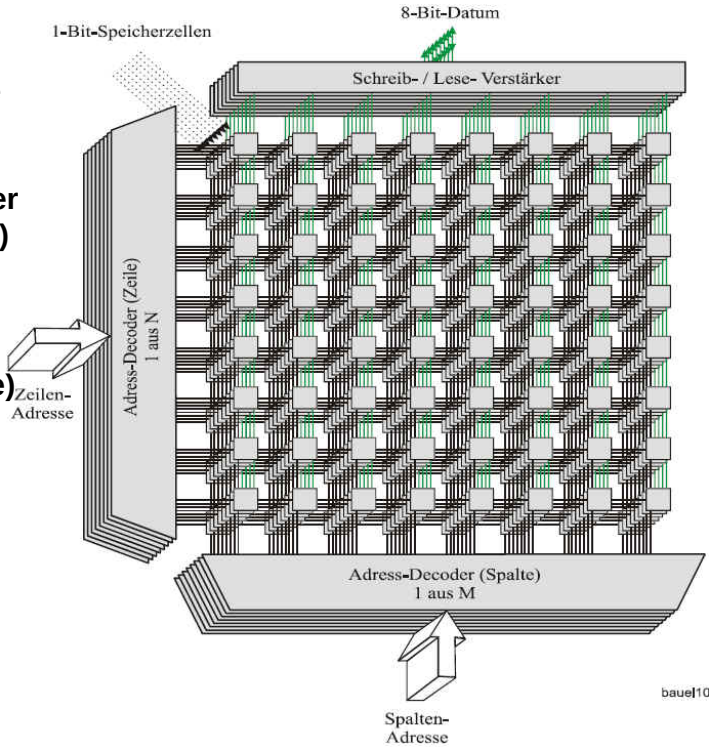
USB-Bus

- **Endpoint-Descriptor:**

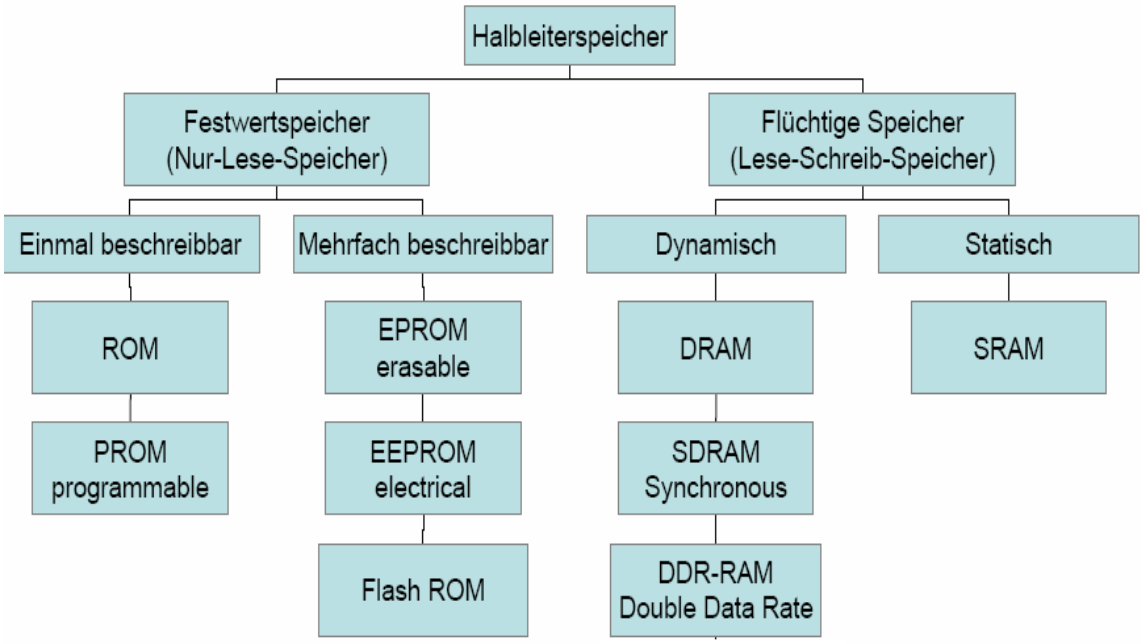
Offset	Bezeichnung	Länge	Beschreibung	z.B.
0	bLength	1	Länge des Descriptors	0x07
1	bDescriptorType	1	Typ = Endpoint Descriptor	0x05
2	bEndpointAddress	1	Endpoint-Nummer	0x81
		1	Bit 7: 0=Output, 1=Input	
3	bmAttributes	1	Transferart	0x03
			0x00: Control-Transfer	
			0x01: Isochronous-Transfer	
			0x02: Bulk-Transfer	
			0x03: Interrupt-Transfer	
4	wMaxPacketSize	2	maximale Paketlänge	0x08
6	bIntervall	1	Abfrage-Interval (für Interrupt-Transfer)	0x10

9 Aufbau von Speicherzellen

- **Speicherung von Daten oder von logischen Funktionen**
- **Arten der Speicherung**
  - ⇨ **irreversibel programmierbare**
  - ⇨ **reversibel programmierbare**
- **Speicherelement: Speicherung der**
- **kleinsten Informationseinheit (Bit)**
- **Speicherzelle**
  - ⇨ **Speicherelemente, die unter gemeinsamer Adresse ansprechbar sind (z.B. 1 Byte)**
- **Speicherwort**
  - ⇨ **Datenbusbreite (z.B 4 Byte)**
- **Organisation**
  - ⇨ **#Speicherzellen n**
  - ⇨ **#Speicherelemente pro Speicherzelle m**
  - ⇨ **n*m Bit-Speicher**
- **Kapazität**
  - ⇨ **Zahl der Speicherelemente**

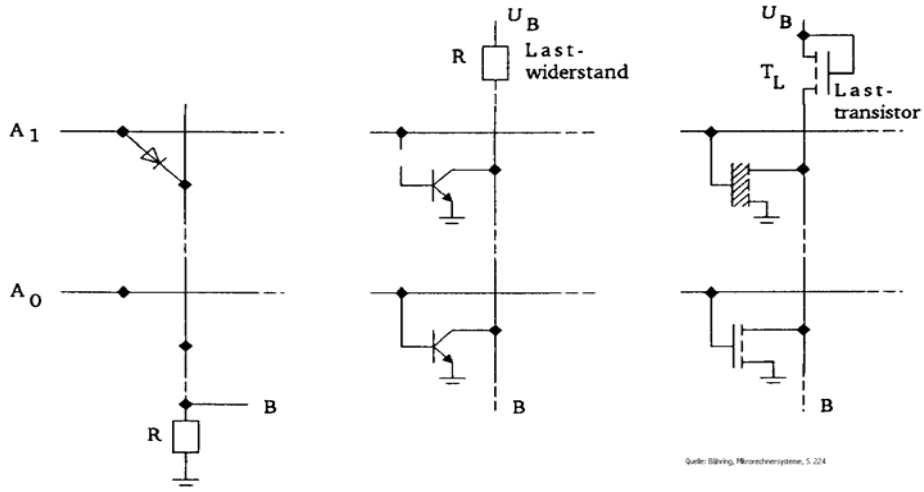


Klassifizierung von Halbleiterspeichern



Speicherzellen für maskenprogrammierbare Speicherelemente

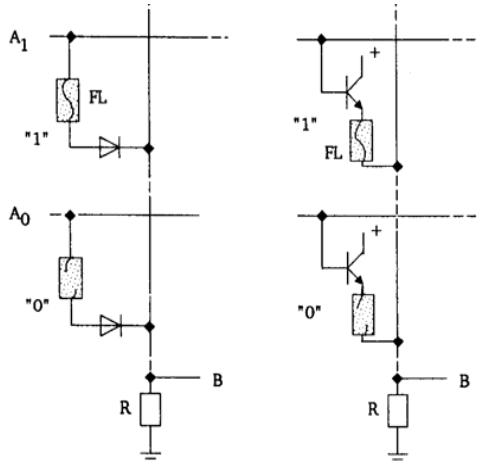
ROM:



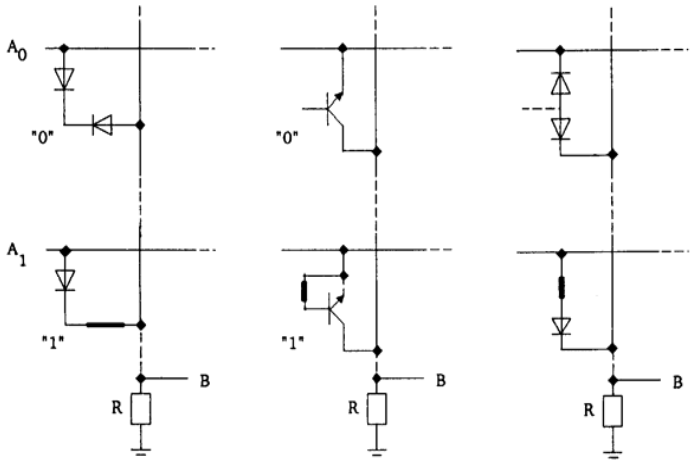
- **Maskenprogrammierbare Speicherelemente erhalten ihre Information bei der Herstellung des Chips**
  - ⇨ **Information steht auf einer Maske**
  - ⇨ **Inhalt ist nicht veränderbar**
- **Bauelemente wie Dioden, Bipolar- oder MOS-Transistoren werden bei der Herstellung deaktiviert**
  - ⇨ **Bei MOS-Transistoren ist die Dicke der Gate-Isolation ausschlaggebend**

Speicherzellen für programmierbare Speicherelemente

PROM:



Speicherzellen mit Schmelzsicherungen

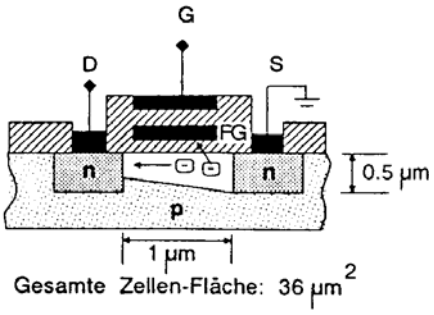


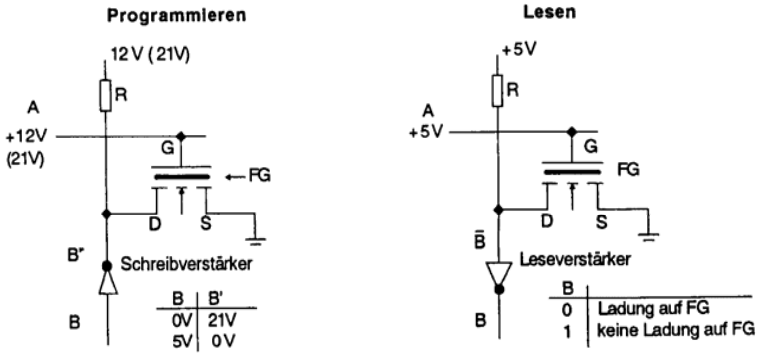
AIM-Speicherzellen

- Programmierung in Programmiergerät durch Überspannungen
  - ⇨ Schmelzsicherung
  - ⇨ Zerstören von Dioden (dauernd leitend)
- Information nur einmal schreibbar und nicht veränderbar

Löschbare Speicherelemente

EPROM: FAMOS (floating gate avalanche MOS-transisistor)



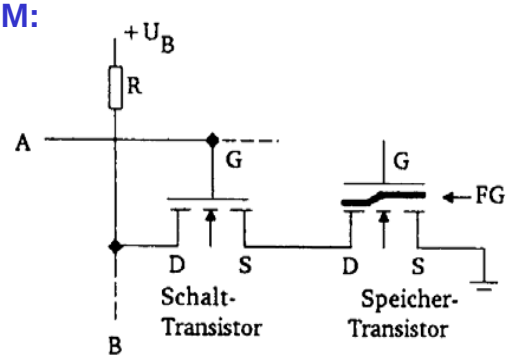


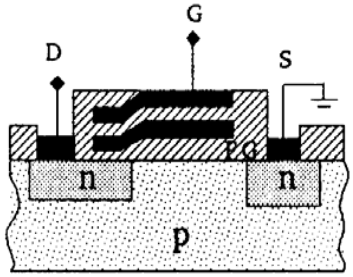
Programmieren und Lesen einer EPROM-Zelle

- Besitzt zweites Gate, das vollständig isoliert ist
  - ⇨ Speicherung der Ladung über 30 Jahre
- Löschen durch UV-Licht (senkt Widerstand der Isolierschicht
- Programmierung durch hohe Spannung (12-21 V)
  - ⇨ Elektronen werden angezogen
- Lesen durch Anlegen einer niederen Spannung (5 V)
  - ⇨ ist das Floating-Gate geladen, schaltet der Transistor nicht

Elektrisch löschbare Speicherelemente

EEPROM:

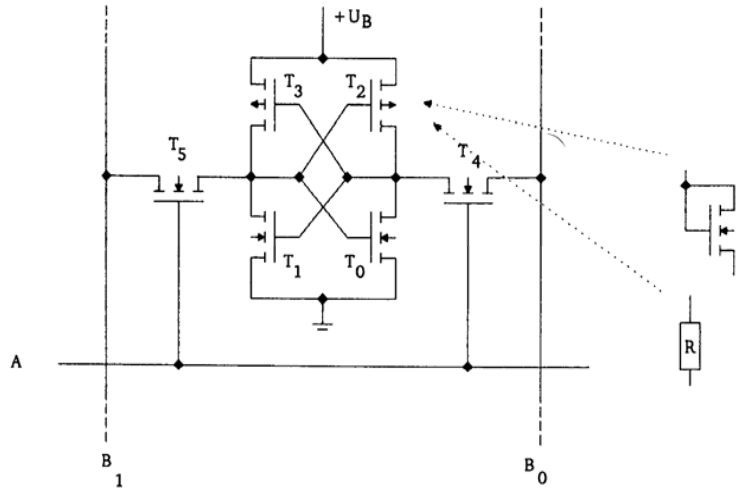




- Dünne Isolierschicht des Floating Gates
  - ⇨ Lesen: Wenn das Floating Gate des Transistors (negativ) geladen ist, sperrt dieser
  - ⇨ Löschen: Hohe Spannung (21 V) am Gate-Anschluss des Transistors lädt das Floating Gate (U_B = 0V)
  - ⇨ Programmieren: 0 V am Gate und eine hohe Spannung am Drain-Anschluss des Transistors entlädt einzelne Floating Gates (logisch 0)
- Spezielle Weiterentwicklung sind Flash-Speicher, die mehrere Bits pro Zelle speichern können (durch mehrere Ladungslevel)

Statische MOS-Speicherelemente

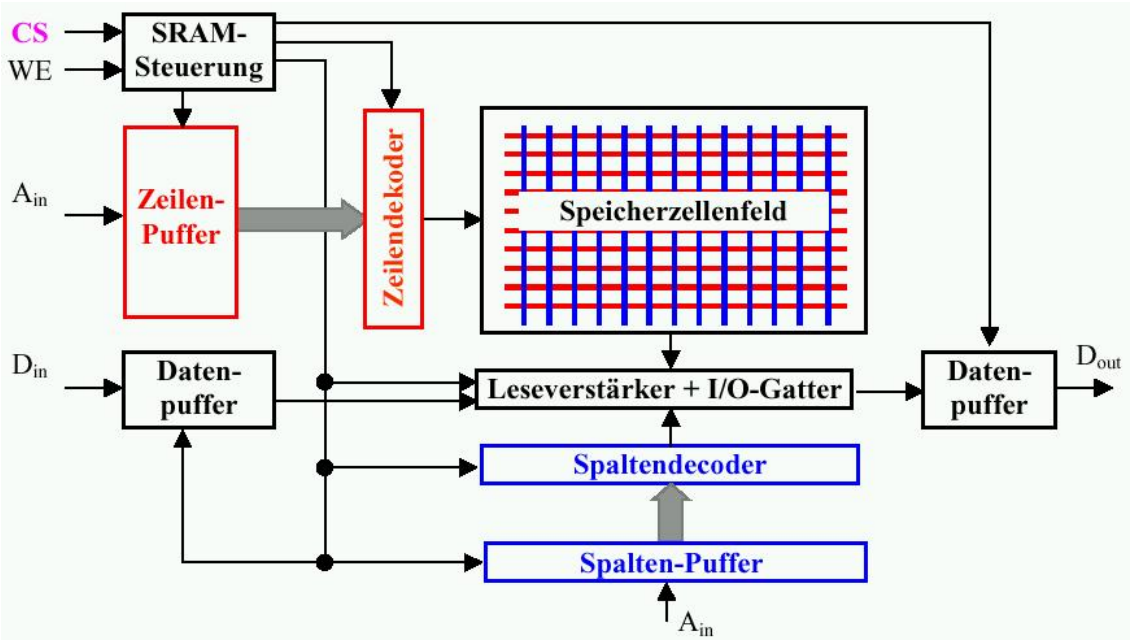
SRAM:



- 6-Transistoren
  - ⇨ Statt T_2 und T_3 können auch n-MOS-Transistoren oder Widerstände eingesetzt werden
  - ⇨ T_4 und T_5 dienen zur Ankopplung an die Bitleitungen
- Anwendung: Cache-Speicher

SRAM-Speicher

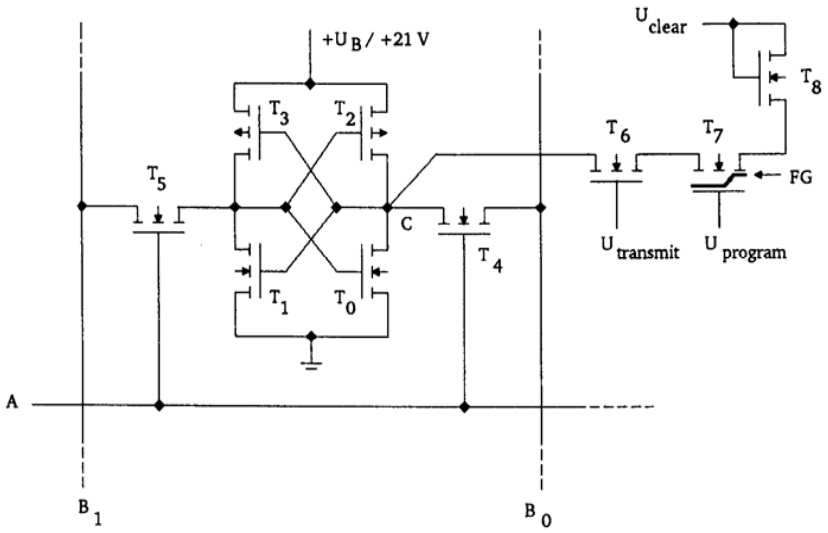
Aufbau eines SRAM-Speicherbausteins



WE = Write Enable
CS = Chip Select

NVRAM-Speicherelemente

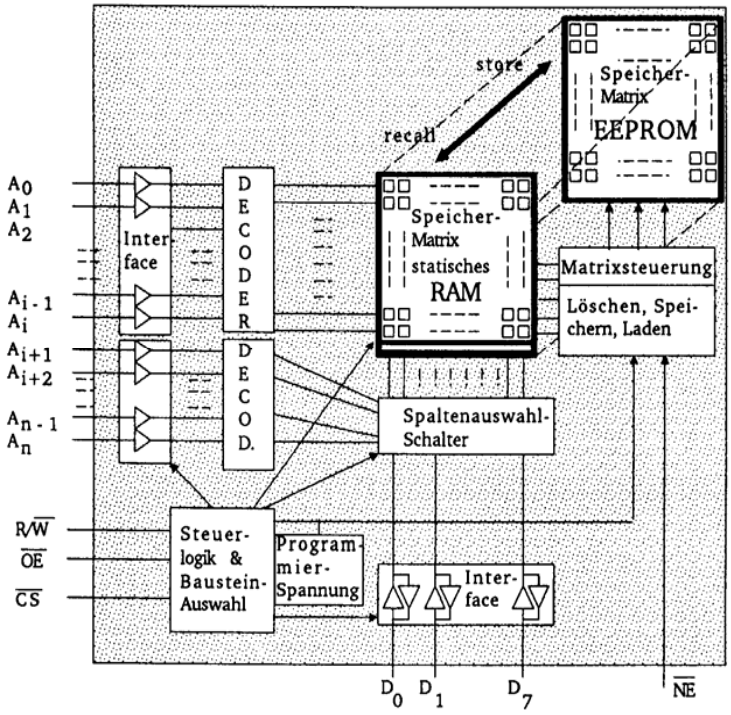
NVRAM:



- **Kombination eines statischen mit einem EEPROM Speicherelement**
  - ⇨ **wenn die Spannung abfällt oder das Gerät eingeschaltet wird, findet eine Übertragung von bzw. in die EEPROM-Zelle statt**

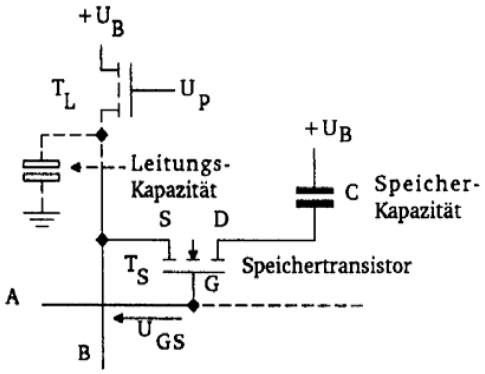
NVRAM-Bausteine

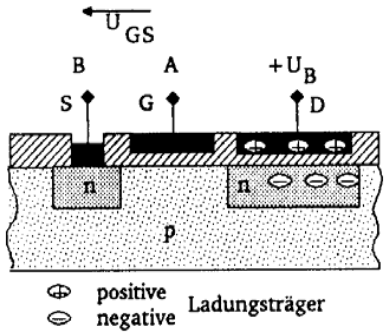
Aufbau eines NVRAM-Speicherbausteins



Dynamische Speicherelemente

DRAM:

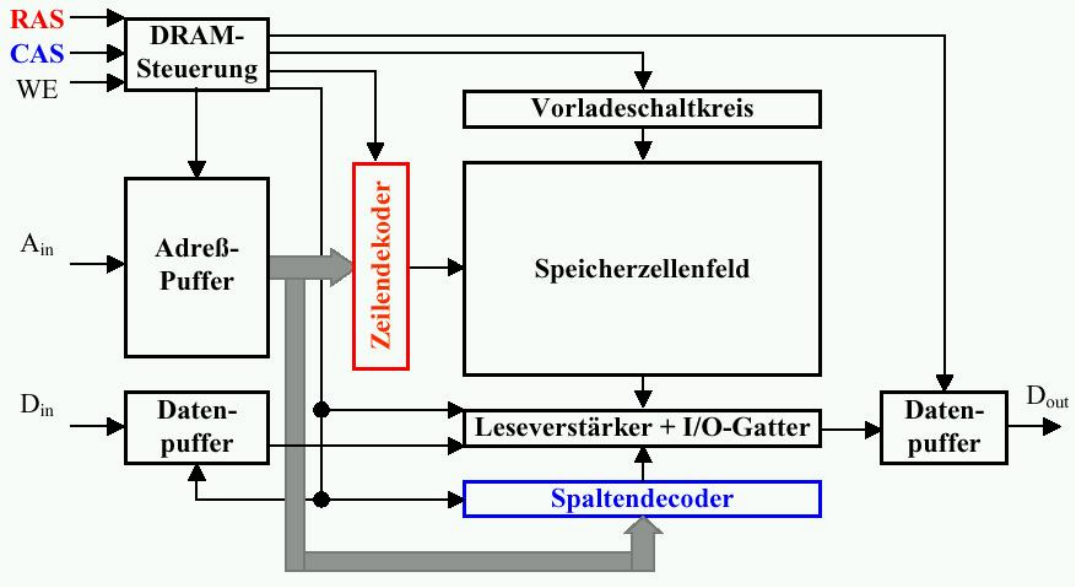




- Die Information wird in Kondensator gespeichert
  - ⇨ vergrößerte Drain-Zone
  - ⇨ isoliert zur Spannungsversorgung
- Kapazität 0,1 - 0,5 pF, 100.000 - 150.000 Elektronen
  - ⇨ Selbstentladung nach ca. 2 ms, deshalb Refresh nötig
- Speichern entspricht dem Laden des Kondensators
- Lesen entlädt den Kondensator
  - ⇨ Daten müssen wieder zurückgeschrieben werden
- Anwendung: Hauptspeicher

DRAM-Speicher

Aufbau eines DRAM-Speicherbausteins



RAS = Row Address Strobe

CAS = Column Address Strobe

DRAM

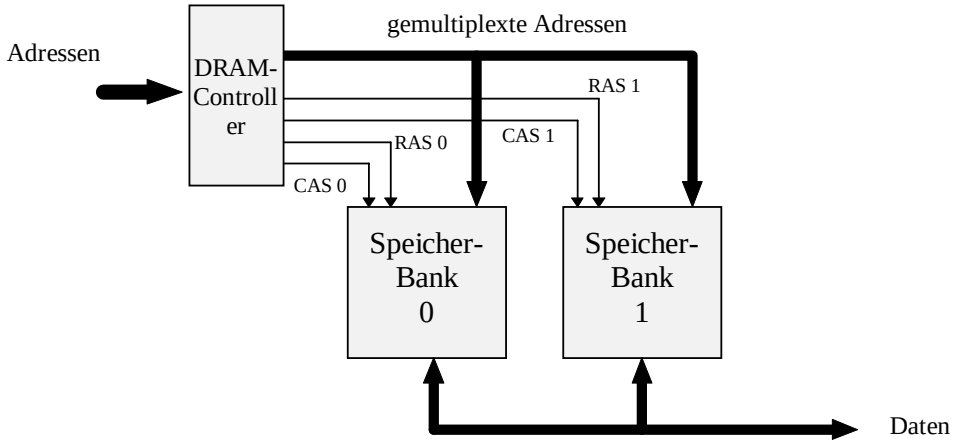
Lesezyklus DRAM

1. **Vorladeschaltkreis setzt alle Bitleitungen auf V_{cc}/2 (nötig wegen der geringen Ladungsmengen in den Zellen)**
2. **Anlegen der Zeilenadresse an den Chip und Übernahme in den Zeilendecoder**
3. **Aktivieren der Zeile und Übertragen der Ladungen der Speicherkondensatoren auf die Bitleitungen in gesamter Zeile**
4. **Signale aller Bitleitungen werden über die Leseverstärker in die I/O-Gatter geschrieben,**
   Anlegen der Spaltenadresse an den Chip und Übernahme in den Spaltendecoder
5. **Auswahl der Bitleitungen der adressierten Speicherzelle und Auslesen des verstärkten Signals,**
   gleichzeitig Refresh aller Speicherzellen in der Zeile

DRAM

- Kompensation der gegenüber den Zugriffszeiten längeren Zykluszeiten

  ⇒ Organisation in ***Interleaved Memory Banks***



Vergleich DRAM - SRAM

- Vorteile DRAM
  - ⇨ kleinere Speicherzellen (1-3 Transistoren)
  - ⇨ dadurch mehr Speicherkapazität bei gleicher Fläche (4-8 fach)
  - ⇨ Kapazität bewirkt langsames Auslesen der Daten
  - ⇨ günstiger in der Herstellung
  - ⇨ Zeitmultiplex zwischen Zeilenauswahl und Spaltenauswahl (Vorteil: weniger Leitungen, Nachteil: dauert länger)

- Vorteile SRAM
  - ⇨ keine externen Zusatzschaltkreise für Refreshgeneratoren nötig
  - ⇨ kurze Schalt- und Zykluszeiten, da stärkere Signale als bei DRAM geliefert werden (SRAM bis 10ns, DRAM 50-100ns)

Varianten DRAM

- Viele Varianten des DRAM existieren
  - ⇨ Ziel Erhöhung der Zugriffsgeschwindigkeit
    - Überlappen von Adressierung und Datenzugriff (EDO RAM)
    - synchroner Zugriff (bezogen auf den Bus) + Burst mode (SDRAM)

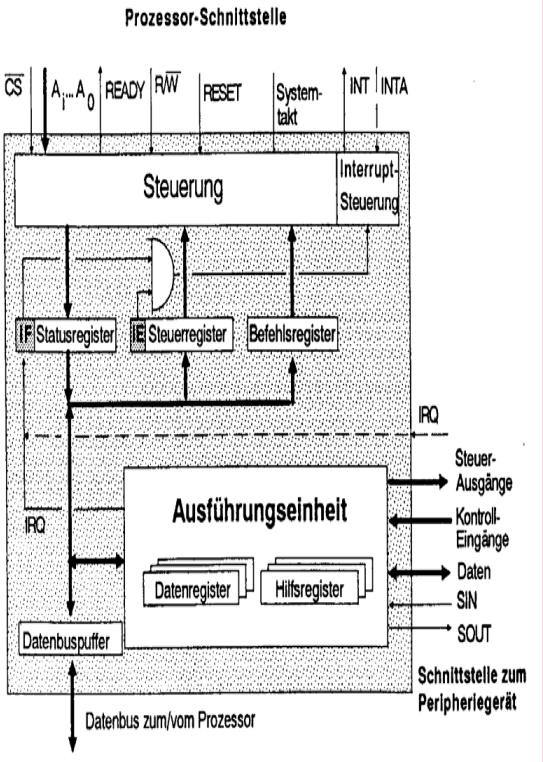
      zusätzlich Datenzugriff bei steigender und fallender Taktflanke (DDR SDRAM) – z.B. 3 ns Zugriffszeit bei 166 Mhz und burst mode

- Cached DRAM
  - ⇨ Kombination von wenig SRAM und viel DRAM
  - ⇨ SRAM wird im Speicher als Cache verwendet
  - ⇨ Zugriffe sollen möglichst auf das schnelle SRAM erfolgen

  Beispiel: Enhanced DRAM
  - ⇨ Die zuletzt zugegriffene Zeile steht im SRAM
  - ⇨ Überprüfung, ob nächster Zugriff wieder die gleiche Zeile betrifft, ist einfach

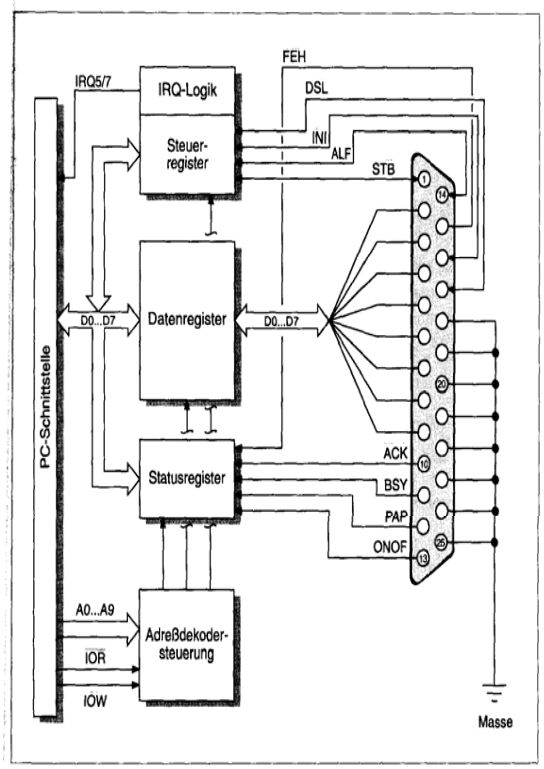
10 E/A und Peripheriegeräte

- Ein- und Ausgabe erfolgt über spezielle Speicherstellen im Adressraum des Prozessors
  - ⇨ Memory Mapped
  - ⇨ spezielle I/O-Befehle
- Adressdekodierung erzeugt das CS-Signal (chip select)
- Der Prozessor kommuniziert über
  - ⇨ Datenregister (Lesen und Schreiben der Daten)
  - ⇨ Statusregister (Zustand des Bausteins)
  - ⇨ Steuerregister (Betriebsart des Bausteins)



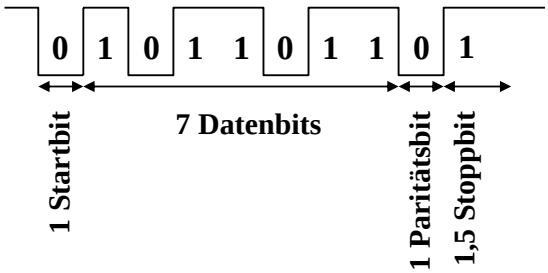
Die parallele Schnittstelle

- Verbindung zum Drucker
  - 8 Bit Daten
  - einfacher Aufbau
  - normalerweise nur Schreiben
  - bei Lesezugriff auf das Datenregister werden die Werte im Datenregister mit den momentan anliegenden Daten mit ODER Verknüpft



Serielle Datenübertragung

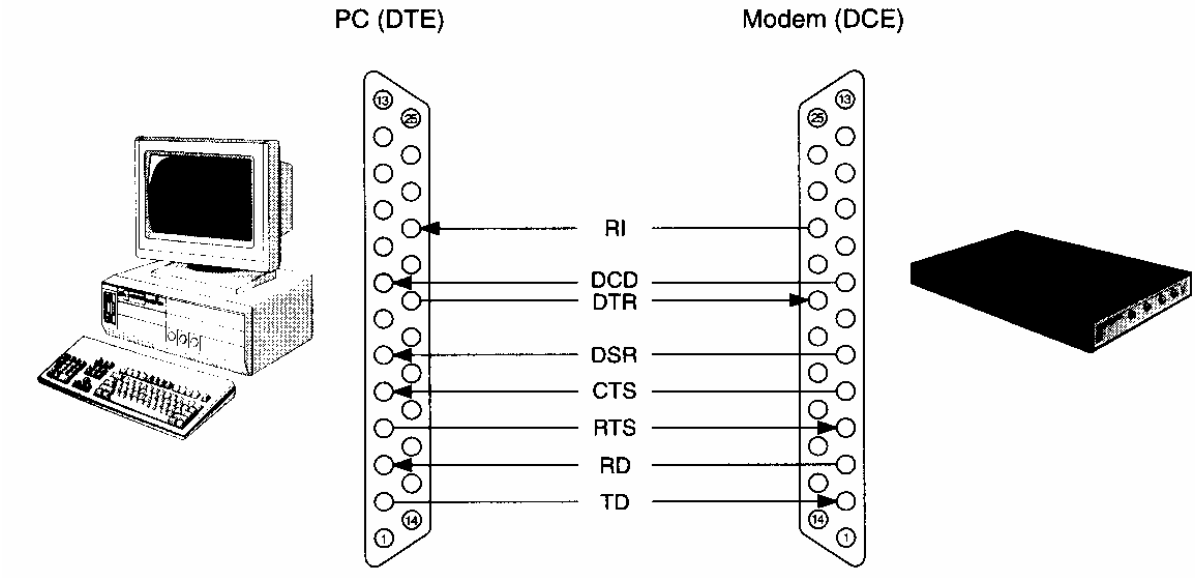
- Baud: Schrittgeschwindigkeit
- Aufbau einer Übertragungseinheit
  - Startbit
    - Kennzeichnet den Anfang einer Übertragung
  - Datenbits
    - das zu übertragende Datum
    - ASCII-Kodierung der Daten
  - Paritätsbit
    - Prüfbit zum Feststellen der Korrektheit der Übertragung
    - gerade Parität: die Zahl der 1en wird zu einer geraden Anzahl ergänzt
  - Stoppbit
    - Markiert das Ende einer Übertragungseinheit
- Das Startbit wird mit 8-facher Rate abgetastet



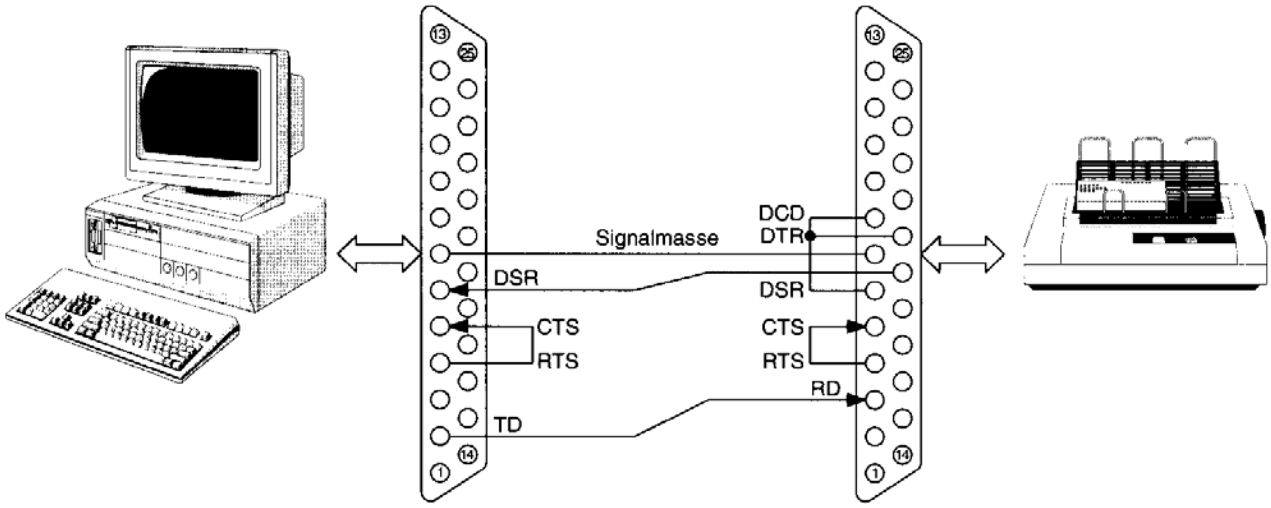
Die RS232-Schnittstelle

- RTS: request to send
  - ⇨ Sendeteil einschalten
- CTS: clear to send
  - ⇨ Übertragungseinrichtung sendebereit
- DCD: data carrier detect
  - ⇨ Trägersignal erkannt
  - ⇨ Empfangsteil einschalten
- DSR: data set ready
  - ⇨ Übertragungseinrichtung betriebsbereit
- DTR: data terminal ready
  - ⇨ Empfangseinrichtung betriebsbereit

Anschluss eines Modems

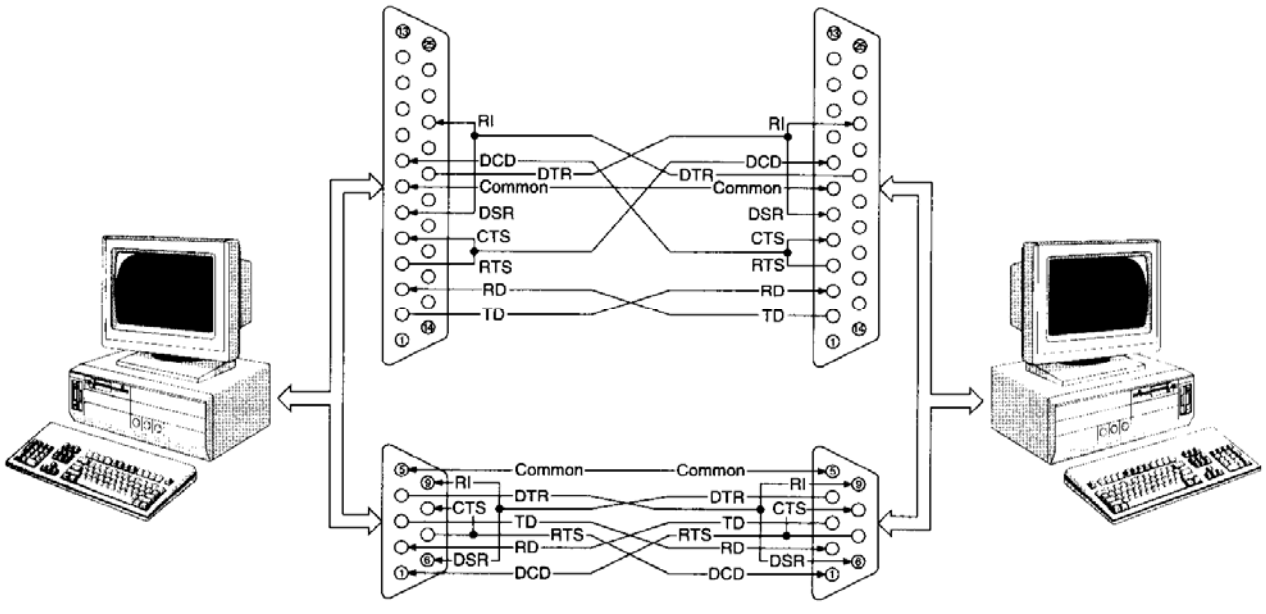


Anschluss eines Peripheriegeräts

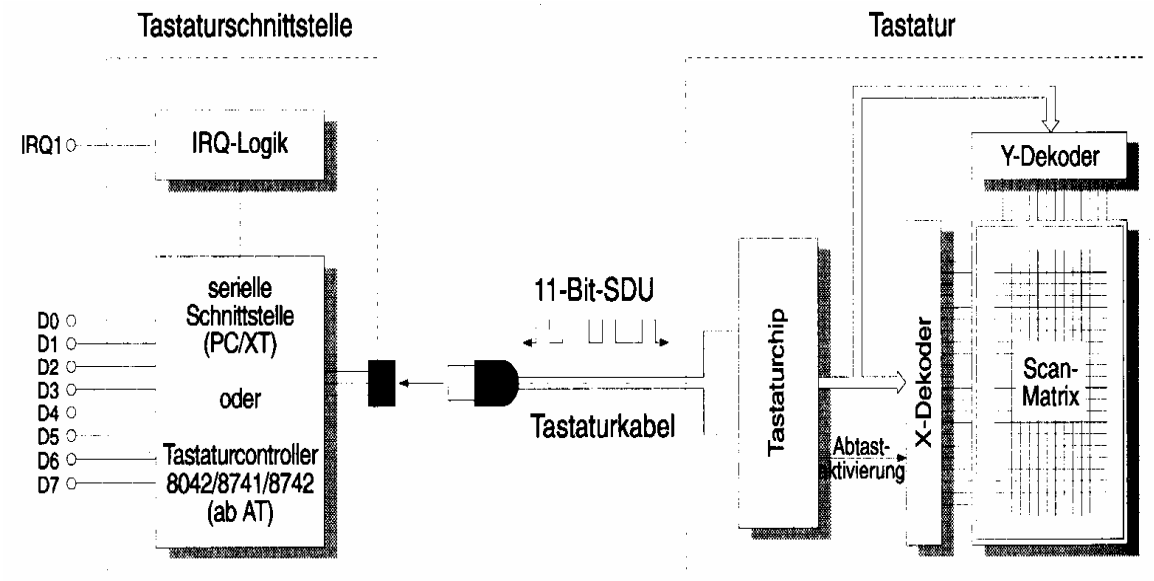


Verbindung zwischen zwei Computern

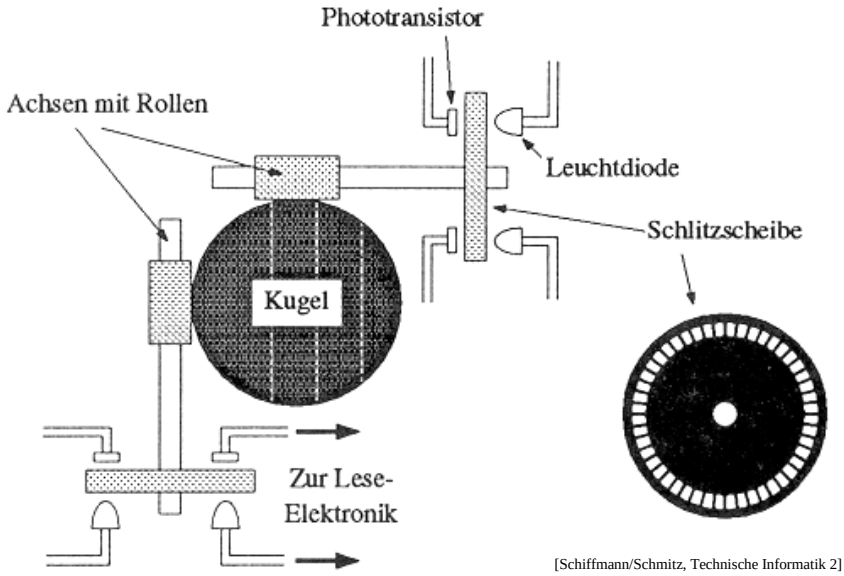
Link-Kabel



Tastatur



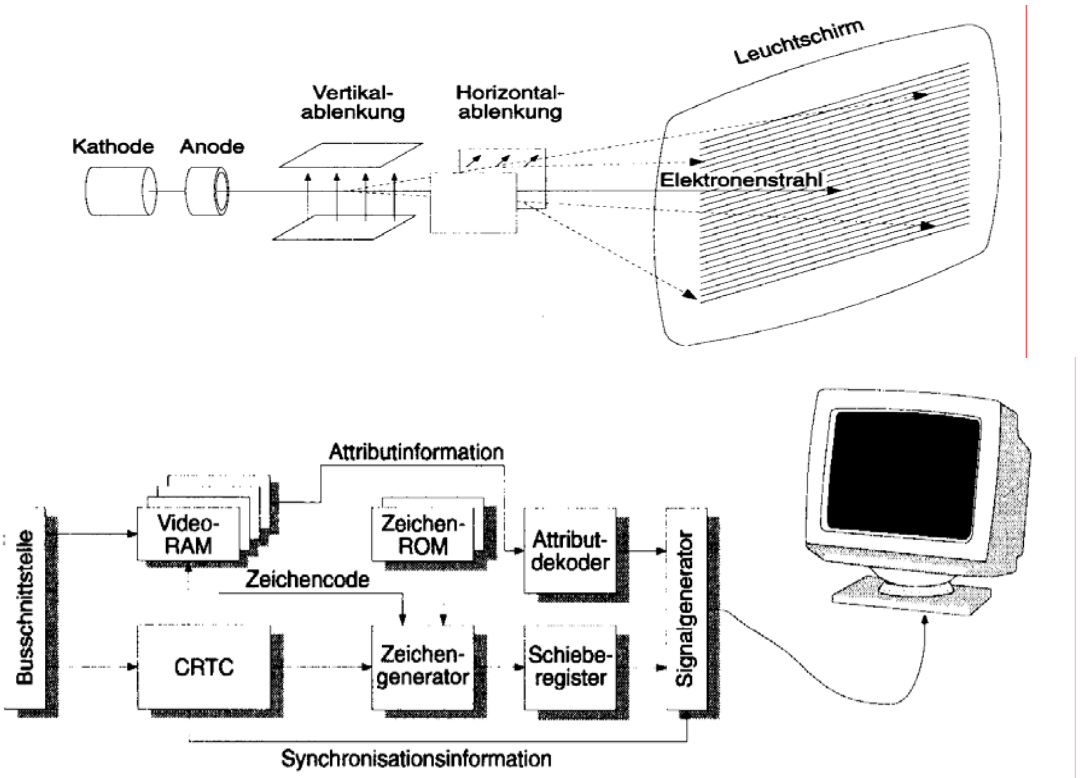
Maus



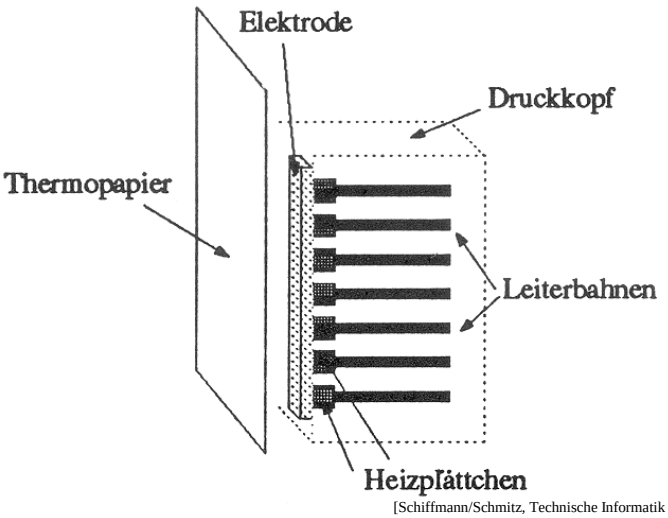
[Schiffmann/Schmitz, Technische Informatik 2]

Funktionsprinzip einer mechanischen Rollmaus

Graphikadapter

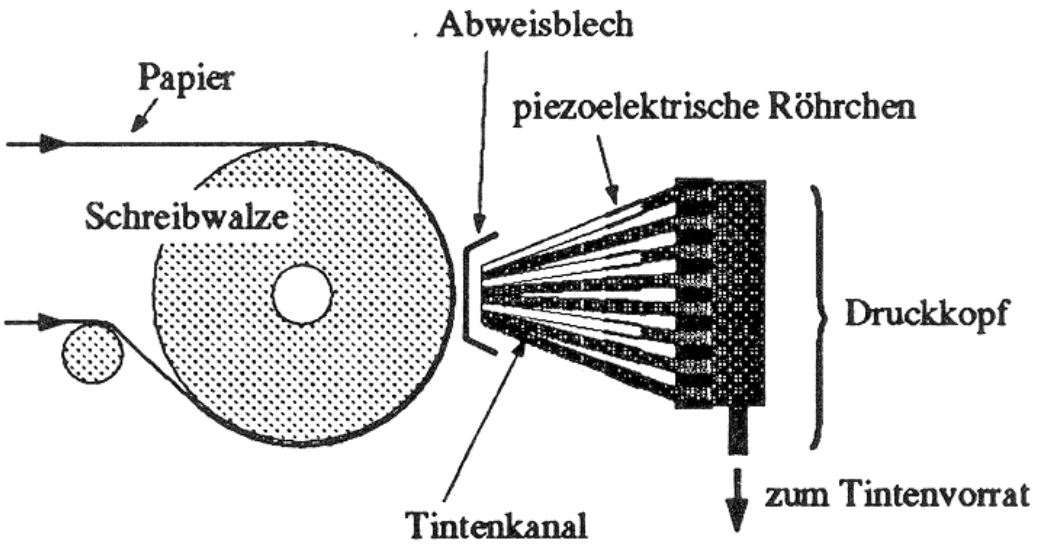


Prinzip eines Thermodruckers



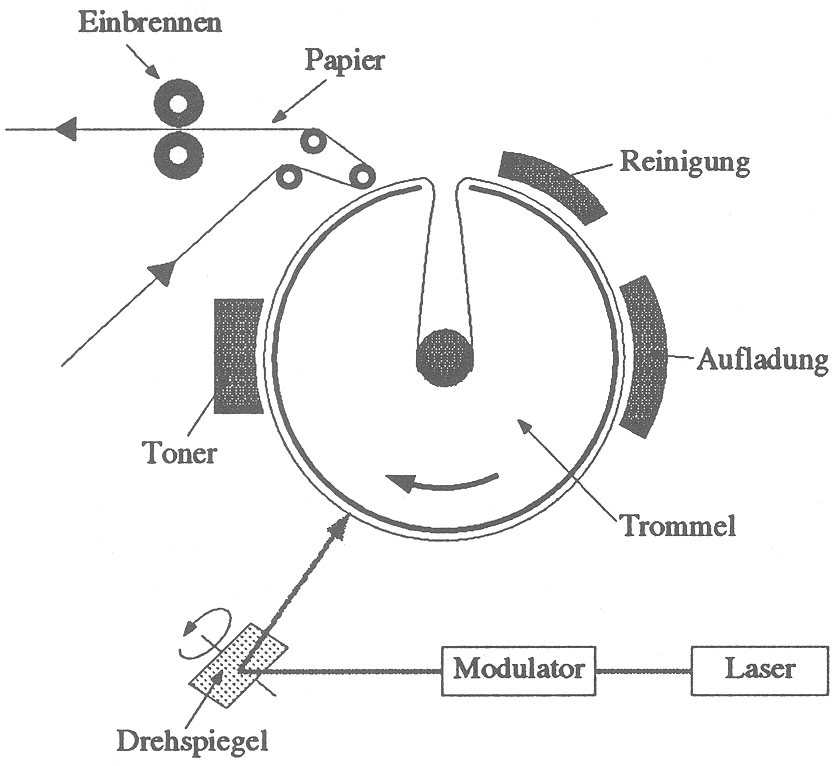
[Schiffmann/Schmitz, Technische Informatik 2]

Prinzip eines Tintenstrahldruckers

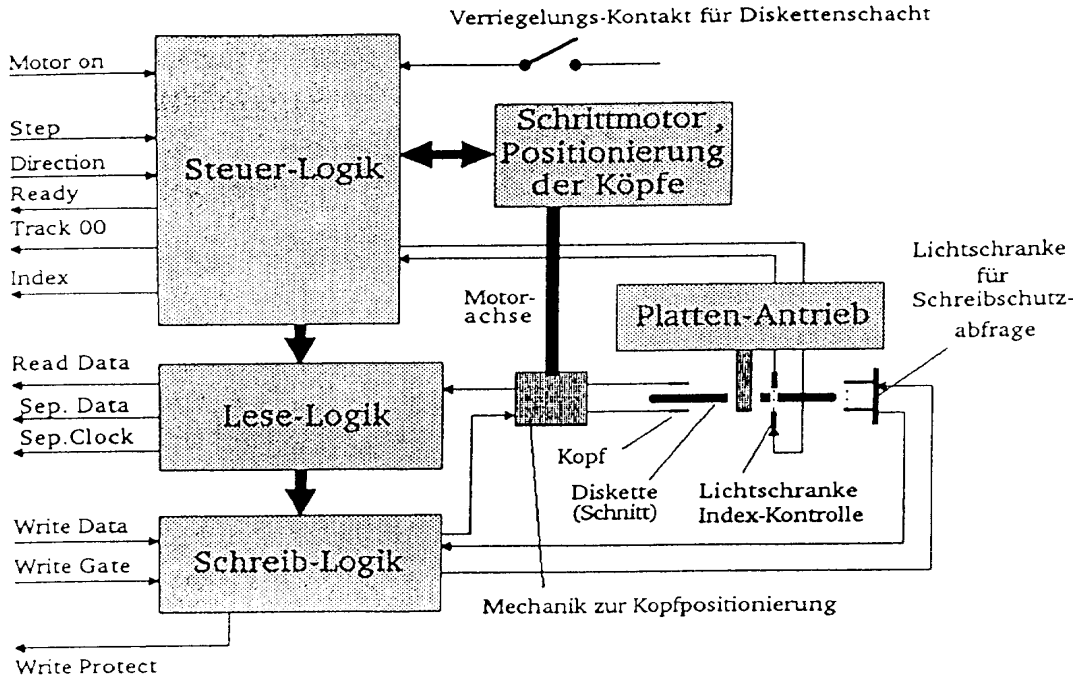


[Schiffmann/Schmitz, Technische Informatik 2]

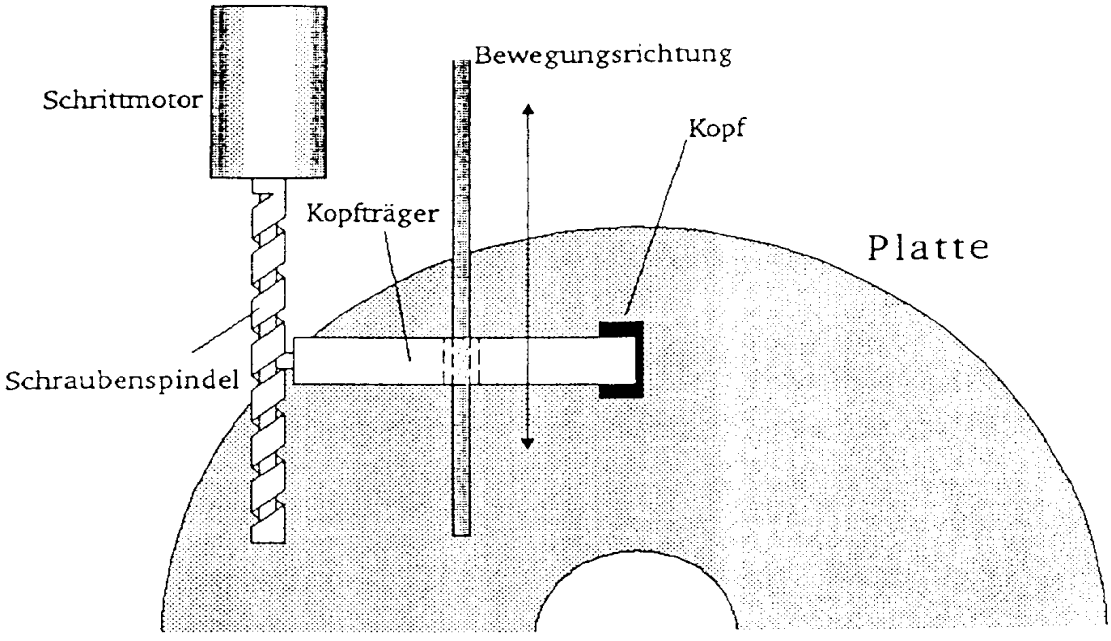
Prinzip eines Laserdruckers



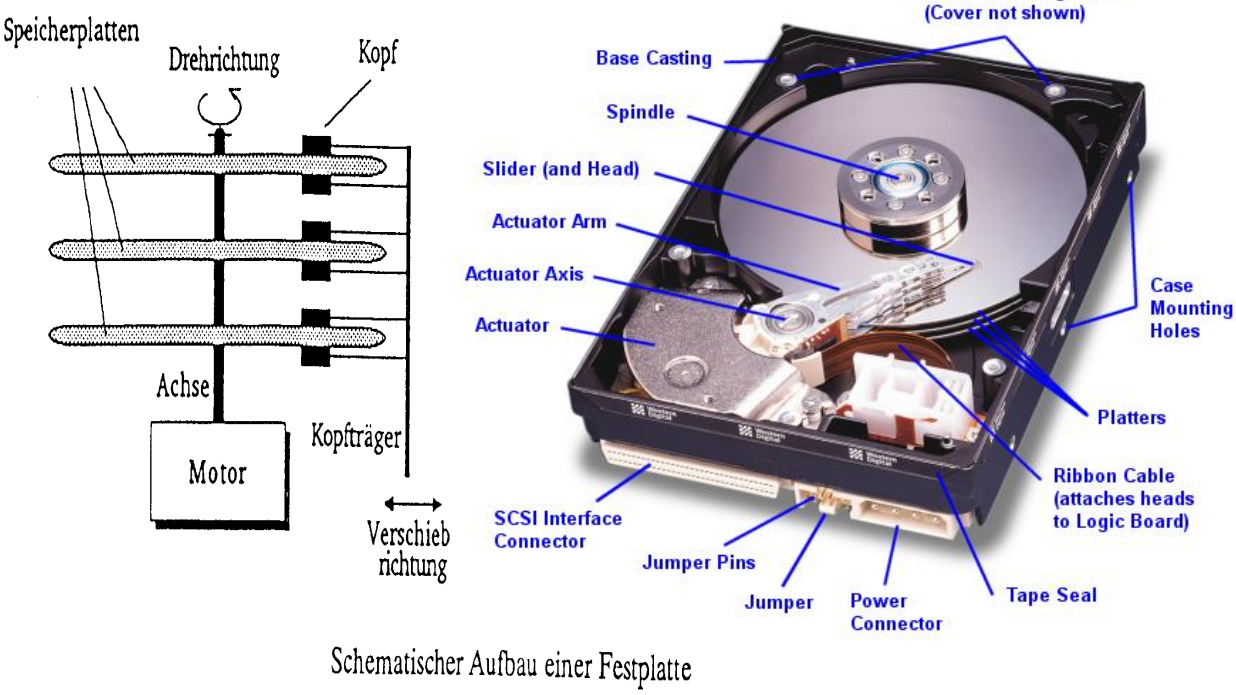
Aufbau eines Floppy-Disk-Laufwerks



Floppy

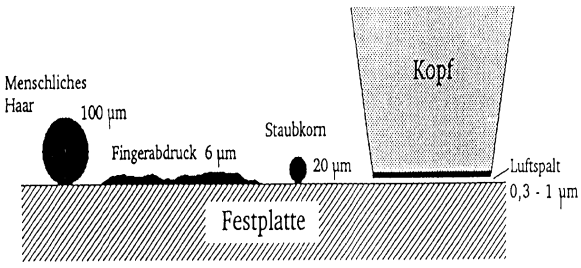


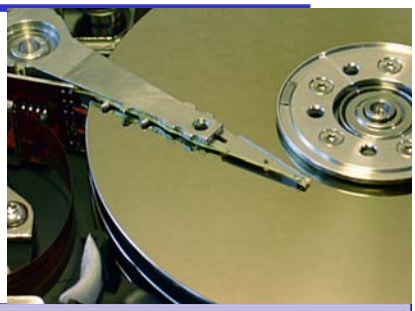
Aufbau eines Festplatten-Laufwerks



Schematischer Aufbau einer Festplatte

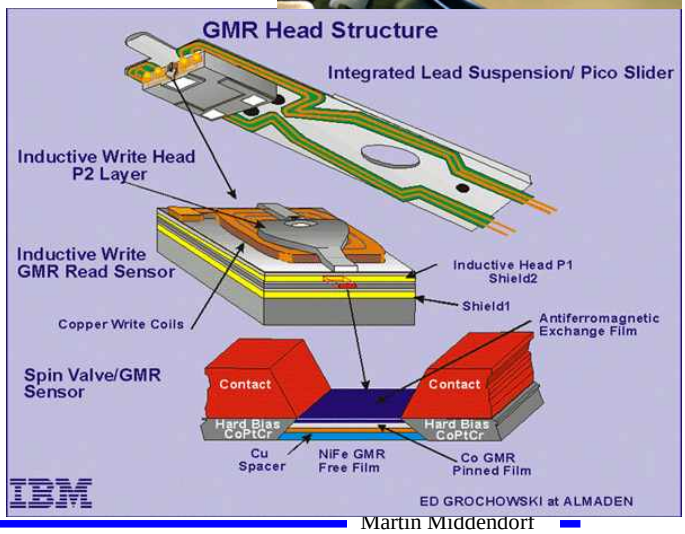
Größenverhältnisse im Festplatten-Laufwerk



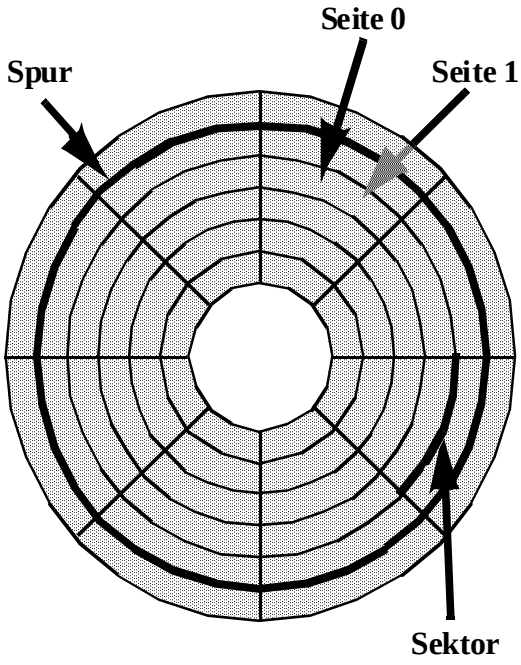


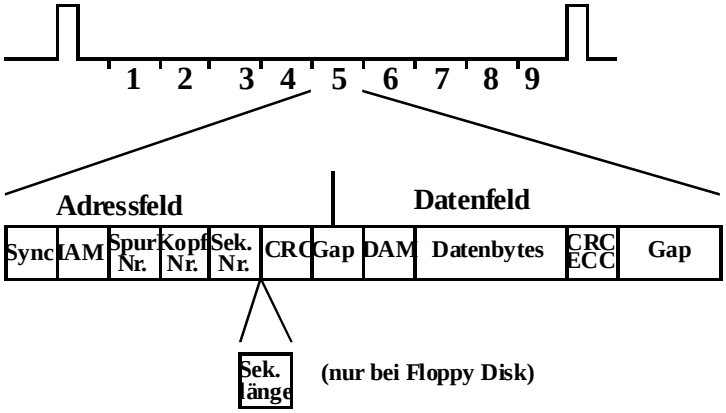
Year	Areal Density Gbits/in²	MR Element
1991	0.132	4.5 um, 64 nm
1992 1993	0.260 0.354	
1994	0.578	
1995	0.829 0.923	
1996 1997	1.32 1.45 2.64	
1998	3.0-5.0	
2000	10.0	0.5 um

Heads Exchange, Hard Bias, ADV Bias, NiFe, Spacer, Soft Film, GMR Pinned Film



Sektoren einer Festplatte





Sync: Synchronisation des Taktes

IAM: ID field address mark (Markierung)

CRC: Prüfbits

DAM: data address mark (Markierung)

Gap: Lücke

Zylinder: Menge aller Spuren, die gleichzeitig mit den Köpfen lesbar sind

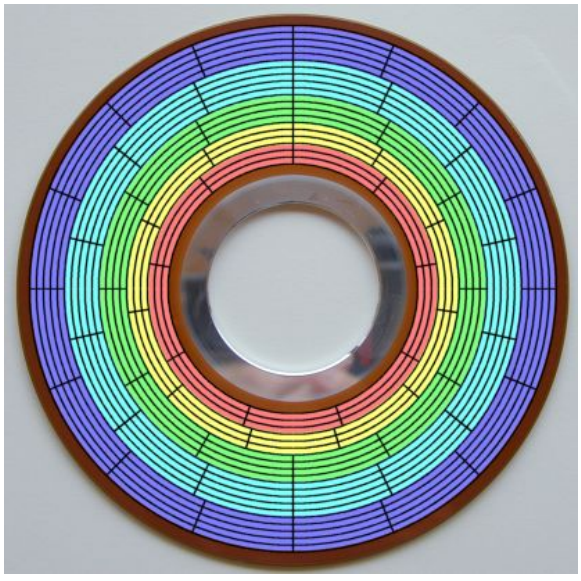
Prinzip der Datenspeicherung

- **LBA-Adressierung (Logical Block Addressing): Sektoren werden nacheinander durchnumeriert**

  → erlaubt unterschiedlich viele Sektoren pro Spur

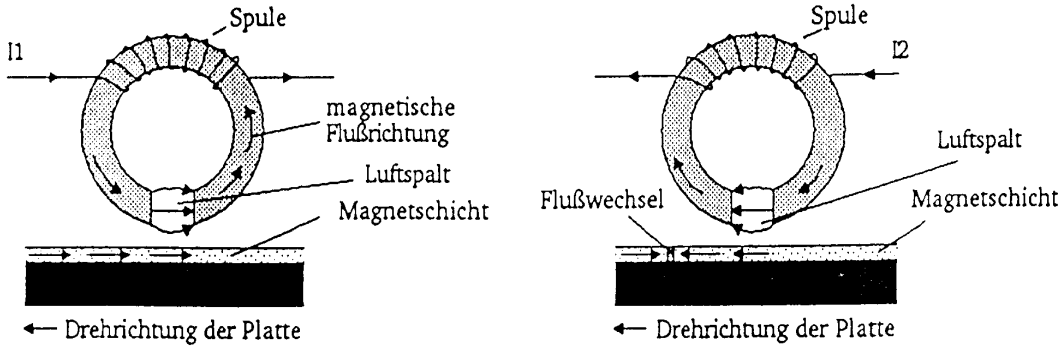
- **ZBR (Zone Bit Recording): Anzahl der Sektoren ist in verschiedenen Zonen unterschiedlich (→ bessere Ausnutzung)**

  ⇨ **Da die Winkelgeschwindigkeit gleich bleibt werden außen Daten mit höherer Rate gelesen**



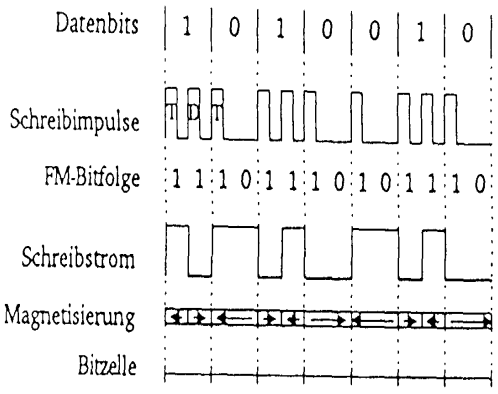
Prinzip der Datenspeicherung

- Das Prinzip der Datenaufzeichnung besteht darin, die Oberfläche der Platte informationsabhängig zu magnetisieren.
- Zur Unterscheidung der „0“- und „1“-Bits wird die Richtung der Magnetisierung verändert. Jede Änderung der Magnetisierungsrichtung wird als Flusswechsel bezeichnet.



Das Frequenzmodulations-Verfahren (FM)

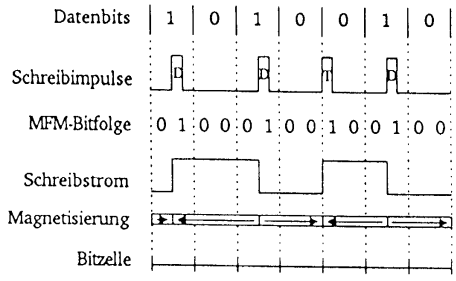
- Prinzip: Zu Beginn jeder Bitzelle wird ein Taktimpuls T abgespeichert. Nur wenn der Inhalt gleich „1“ sein soll, folgt in der Mitte der Bitzelle das Datum D als weiterer Impuls.
- Dieses Verfahren ist relativ langsam und Speicherplatzintensiv, da in jeder Bitzelle mit dem Datenbit auch der Takt aufgezeichnet werden muss. Es wird auch als Format mit einfacher Schreibdichte bezeichnet (Single Density Format).



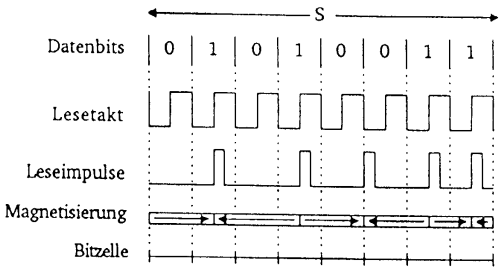
FM-Aufzeichnungsverfahren

Das modifizierte Frequenzmodulations-Verfahren (MFM)

- **Prinzip: es wird nur in solchen Zellen ein Taktimpuls abgelegt, in denen auch ein „1“-Datenbit gespeichert werden soll. Dadurch benötigt jede Bitzelle nur noch den halben Platz auf der magnetischen Oberfläche.**
- **Soll eine „1“ geschrieben werden, wird ein positiver Datenimpuls D in der Mitte geschrieben. Bei einer „0“ wird ein Taktimpuls T am Anfang der Zelle abgelegt, wenn im Takt vorher nicht eine „1“ geschrieben wurde.**
- **Damit wird bei einer Folge von „0“-Bits. Der Takt am Anfang einer jeden Bitzelle abgespeichert und ermöglicht so die Synchronisation beim Lesen der Daten.**
- **Das MFM-Format wird als Format mit doppelter Schreibdichte bezeichnet (Double Density Format).**



MFM-Aufzeichnungsverfahren

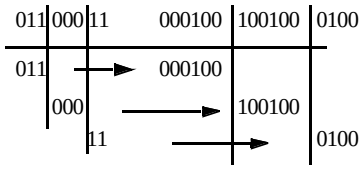


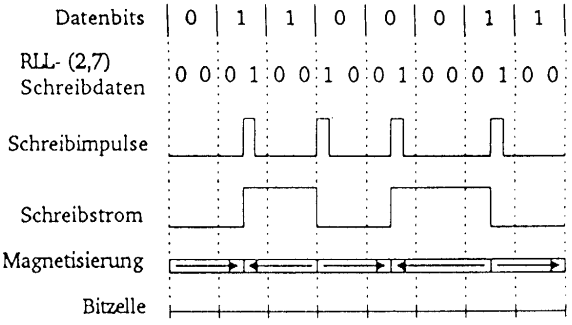
Rückgewinnung der Daten beim MFM-Verfahren

Das RLL-Verfahren

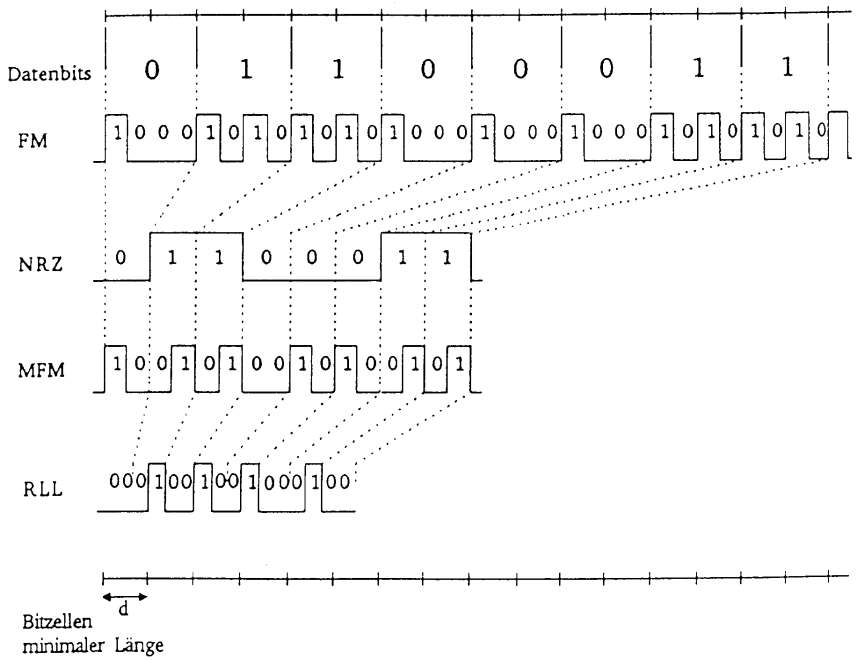
- **Ziel des Verfahrens ist, die Aufzeichnung von „0“-Läufen zu begrenzen. Dies wird durch eine geeignete Kodierung der Daten erreicht.**
- **RLL-(2,7) bedeutet, dass zwischen zwei „1“-Bits mindestens 2 jedoch höchstens 7 „0“-Bits liegen.**
- **Neben dem zu kodierenden Bit werden zusätzlich noch ein oder zwei folgende Bits berücksichtigt (kontextabhängig)**

Bitkombination Bit	Kontext	RLL-(2,7)-Code	
1	0	10	00
1	1	01	00
0	00	10	0100
0	10	00	1000
0	11	00	0100
0	010	00	001000
0	011	00	100100





Vergleich des Speicherbedarfs der verschiedenen Aufzeichnungsverfahren



Optische Speichermedien

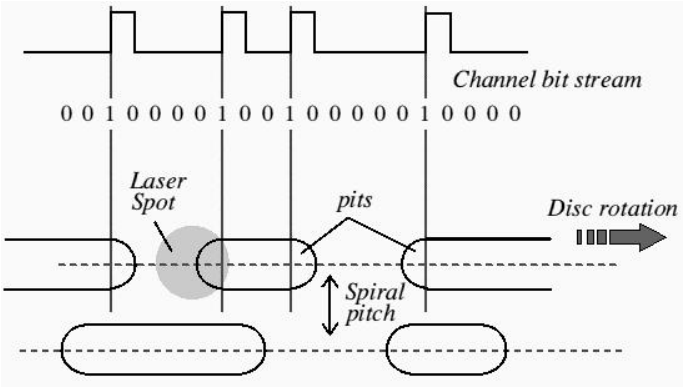
Compact Disk (CD)

- Eigenschaften:
  - ⇨ 1 Bit Information wird durch den Reflektionsgrad (Veränderung oder gleich bleibend) der entsprechenden Stelle auf der CD kodiert.
  - ⇨ Schichtenfolge: Label, Schutzschicht, Reflektionsschicht, Substratschicht (Polycarbonat)
  - ⇨ 16000 Spuren spiralförmig, 600 nm Breite, 1,6 µm Abstand
  - ⇨ konstante Lineargeschwindigkeit
  - ⇨ 1 Millionen Bits/mm^2 Speicherkapazität

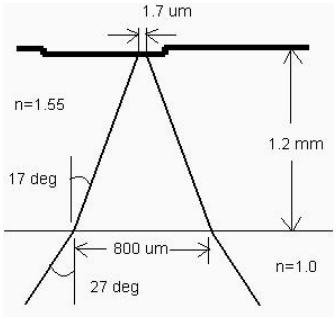
- Informationsspeicherung
  - ⇨ Vertiefungen im Substrat: Pits (schlecht reflektierend)
    Bereich dazwischen: Land (gut reflektierend)
  - ⇨ Pits: Tiefe 1/4 der Wellenlänge des Lichts, Länge Vielfaches von 0,3 µm
  - ⇨ Übergang zwischen Pit und Land: Ein Teil des Strahls wird um ½ Wellenlänge versetzt reflektiert.
    Interferenzen löschen das reflektierte Licht nahezu aus.
  - ⇨ Laserstrahl wird auf das Land fokussiert:
    - Pits streuen das Licht ←→ Land reflektiert

Optische Speichermedien

- Informationsspeicherung:

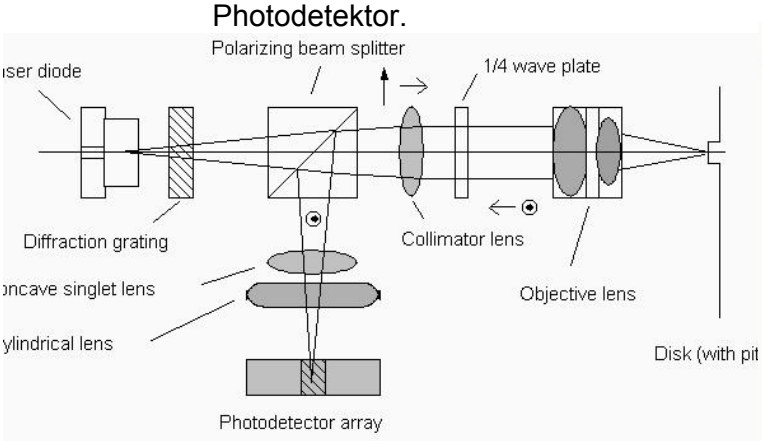


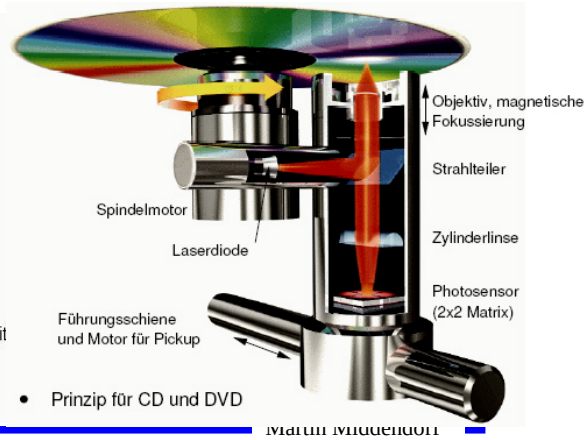
- Brechung des Lichts:



Optische Speichermedien

- Weg des Laserstrahls:
  - ⇨ Beugungsgitter: Teilt Licht in einen Hauptstrahl und zwei Nebenstrahlen.
  - ⇨ Polarisationsfilter: Lichtstrahlen schwingen in einer Richtung.
  - ⇨ 1/4 Wave Platte dreht das Licht um 90 Grad.
  - ⇨ Polycarbonat fokussiert den Lichtstrahl.
  - ⇨ Pit und Land reflektieren Licht unterschiedlich gut.
  - ⇨ 1/4 Wave Platte dreht das Licht um 90 Grad.
  - ⇨ Polarisationsfilter: Reflektiert das um 180 Grad gedrehte Licht in Richtung Photodetektor.

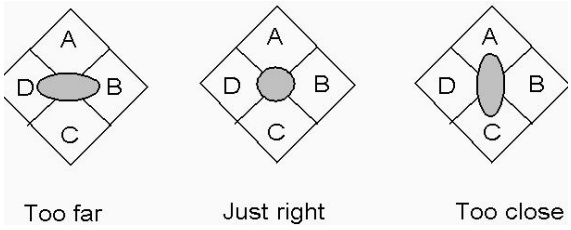




- Prinzip für CD und DVD

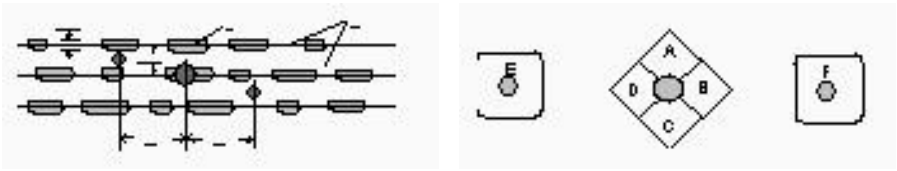
Optische Speichermedien

- Fokussierung: Zylindrische Linse bewirkt, dass Lichtstrahl bei falscher Entfernung des Lasers von der CD ellipsenförmig wird.

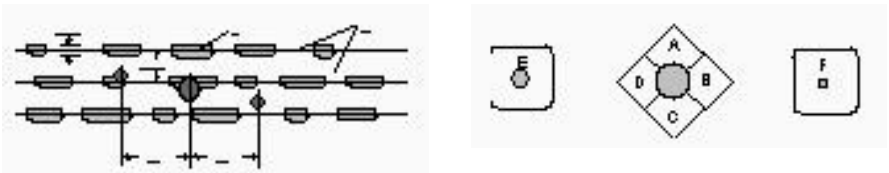


- Spurhalten: Die Nebenstrahlen werden von zwei zusätzlichen Photodetektoren empfangen. Bei richtiger Spurhaltung liegen Nebenstrahlen über Land. Bei falscher Spurhaltung bemerkt das Photosystem Ungleichheiten.

korrekt:



zu weit links:



Optische Speichermedien

CD-DA (Digital Audio): Phillips, Sony (Red Book, 1983)

- ⇨ Speicherkapazität: ca. 750 MByte
- ⇨ Bitlänge: 0,3 µm (Länge der Pits/Lands Vielfaches davon)
- ⇨ Geschwindigkeit zwischen ca. 200 U/min (außen) und 530 U/min (innen)

- Fehlerkorrektur mit Cross Interleaved Reed Salomon Code (CIRSC):
  - ⇨ Für 12 Audio Bytes jeweils 4 Byte Fehlersicherungsdaten für Reed-Solomon Code benötigt.
  - ⇨ Real hintereinander liegende Audio-Daten werden auf mehrere Rahmen (Frames) verteilt.
  - ⇨ Fehlerrate ungefähr 10^{-8}.
  - ⇨ Ein Burstfehler über 7 Rahmen ist korrigierbar (ca. 2,5 mm Spurlänge).

Optische Speichermedien

- Code:
  - ⇨ Restriktionen: Zwischen 2 Einsen stehen mindestens zwei Nullen. Höchstens 10 Nullen hintereinander (damit die Synchronisation gewährleistet bleibt).
  - ⇨ 8-auf-14 Modulation: Unter den gegeben Restriktionen sind von den 2^{14}=16384 14-Bit Folgen nur 267 möglich (von denen 256 benötigt werden).
  - ⇨ Füllbits: Zwischen zwei 14 Bit Folgen werden noch 3 Füllbits gepackt, damit es an den Grenzen keine Verletzung der Restriktionen gibt.

Beispiel:

Audio Bits	00000000	00000001
Kodierung	0 100 1000 100000	10000 100000000
Füllbits	0 10	100
Kanal Bits	0 10 0 100 1000 100000	100 10000 100000000
Pits/Lands	l pp p lll pppp llllll	ppp lllll pppppppp

Optische Speichermedien

- Frame (Rahmen):
  - ⇨ Synchronisation: 27 Kanal-Bits
  - ⇨ 1 Control/Display Byte: Die Bits ergeben über 98 Frames gelesen 8 Subchannels:
    - Q-Subchannel: relative Zeit im Takt und absolute Zeitangabe (Im Vorspann der CD zur Speicherung des Inhaltsverzeichnisses)
  - ⇨ 2 Kanäle mit je 12 Byte Audio-Daten plus 4 Byte Fehlerkorrektur
  - ⇨ Insgesamt: 27+1* (14+3)+2*(12+4)*(14+3)= 588 Bits
- Block = 98 Rahmen entspricht 24x98=2352 Audio-Bytes (insgesamt ca. 333000 Blöcke)
- Tracks: Zusammenfassung mehrere Blöcke (entspricht meist einem Lied).
  - ⇨ Track besitzt mehrere Indexpunkte (z.B. Beginn des Tracks (danach Track-pregap von 2-3 sec.), Beginn der eigentlichen Audiodaten).
  - ⇨ Wahlfreier Zugriff auf Tracks und Indexpunkte möglich.
- CD unterteilt sich in:
  - ⇨ Lead-in Bereich (Inhaltverzeichnis mit Angabe des Beginns der einzelnen Tracks),
  - ⇨ Programmbereich (eigentliche Daten, bis 99 Tracks verschiedener Länge)
  - ⇨ Lead-out Bereich

Optische Speichermedien

CD-ROM (Read Only Memory): 1985 Yellow Book

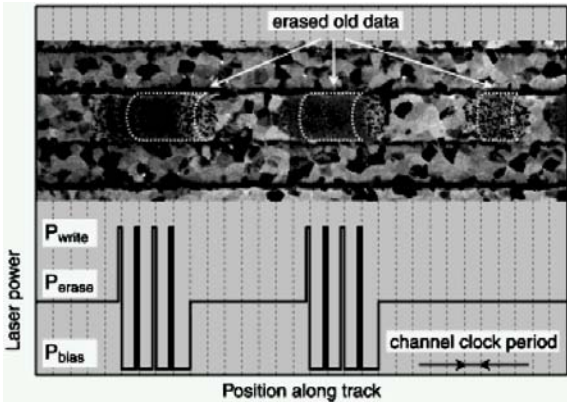
- ⇨ Kann zusätzlich zu Audio-Daten auch allgemeine Rechnerdaten speichern. Außerdem Grundlage zur Spezifikation der Speicherung für weitere Datenarten.

- Notwendig für Rechnerdaten: bessere Fehlerkorrektur
  - ⇨ Pro Block (2352 Byte)
    - 12 Byte Synchronisation
    - 2048 Byte Nutzdaten
    - 4 Byte Fehlererkennung
    - 8 Byte ungenutzt
    - 276 Byte Fehlerbehebung (dadurch Fehlerrate 2^{-13})
- Für Rechnerdaten ist ein Format für Directories wünschenswert (ISO9660):
  - ⇨ Directory-Baum
  - ⇨ Pfad-Tabelle (wird nach Einlegen der CD-ROM in den Rechner geladen): gepackte Verzeichnisse, die einen direkten Zugriff auf Dateien ermöglicht.
  - ⇨ Primary-Volume-Descriptor (ab dem 16 Block (auch Sektor genannt) im ersten Track): Enthält z.B. Länge und Adresse der Pfadtabelle.

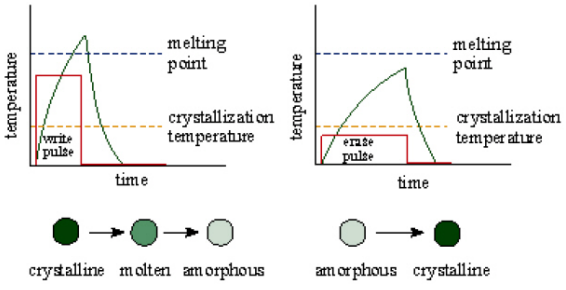
Optische Speichermedien

CD-RW (ReWritable): Orange Book Part III

- Prinzip:
  - ⇨ Polykristallines Material wird durch Erhitzen über Schmelzpunkt (ca. 600 Grad) amorph. Bei schneller Abkühlung bleibt amorpher Zustand erhalten.
  - ⇨ Erhitzung auf mittlere Temperaturen über Kristallisationspunkt (ca. 300 Grad) jedoch unterhalb des Schmelzpunktes bewirkt Wiederherstellung des polykristallinen Zustands.



- Polykristalliner Zustand reflektiert besser als der amorphe Zustand.



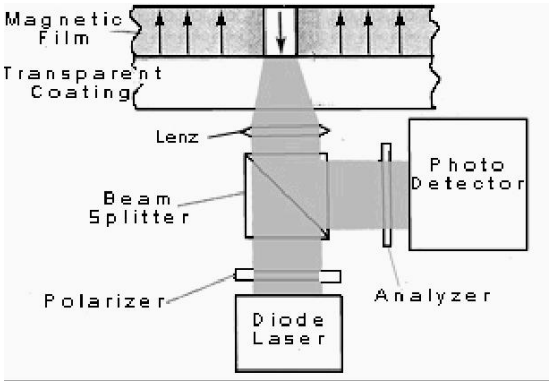
- Normale CD-Player können CD-RW's lesen, falls sie einen Verstärker haben, da Reflektion schwächer ist als bei CD-DA.

Optische Speichermedien

CD-MO (Magneto-optical): Phillips, Sony Orange Book Part I

- Unterschiede in den magnetischen Eigenschaften des Substrats dienen zur Informationsspeicherung

- Lesen:
  - ⇨Ein Laser beleuchtet die Spur mit polarisiertem Licht.
  - ⇨Abhängig von der Magnetisierung der Schicht wird das polarisierte Licht gedreht und reflektiert.
  - ⇨Das reflektierte Licht wird durch einen weiteren Polarisationsfilter geschickt. Nur nach Drehung durch abwärts gerichtetes Magnetfeld wird das reflektierte Licht durchgelassen.

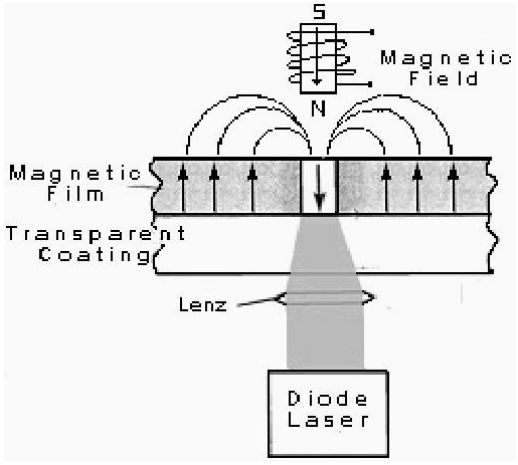


Optische Speichermedien

- Magnetische Eigenschaften des Substrats ändern sich oberhalb einer bestimmten Temperatur (Curie-Temperatur):
  - ⇨ Material nimmt dann leicht das Magnetfeld der Umgebung an.

- Löschen: Nach oben gerichtetes Magnetfeld wird angelegt und mittels Laser die gesamte Spur erhitzt.

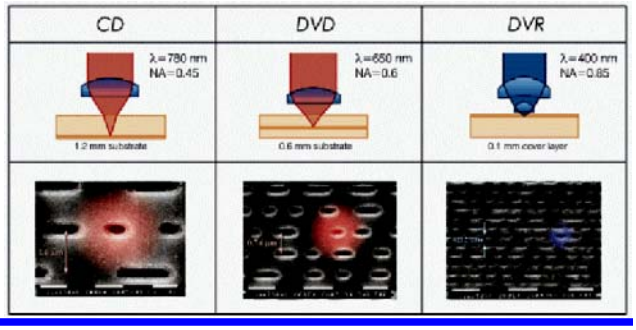
- Beschreiben: Nach unten gerichtetes Magnetfeld wird angelegt. Durch Erhitzen mittels Laserimpuls an bestimmten Stellen übernimmt das Material das nach unten gerichtete Magnetfeld (entspricht einer Eins).



Optische Speichermedien

DVD (Digital Versatile Disk): DVD Forum (Firmen) 1996

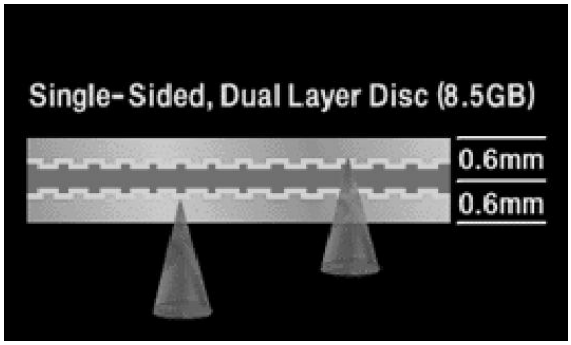
- Geschwindigkeit: Zwischen ca. 570 U/min (außen) und 1630 U/min (innen)
- Formate: DVD-ROM, DVD Video auf DVD-ROM, DVD-Audio, DVD-R, DVD-RAM
- Bessere Kapazität als CD (4,7 GB bei Single/Layer statt 650 MByte bei CD-ROM):
  - ⇨ Entfernung zwischen Tracks 0.74 µm (möglich durch kürzere Lichtwellenlänge von ca. 650 nm); statt 1,6 µm bei CD)
  - ⇨ Minimale Länge der Pits: ca. 0,4 µm (statt 0,83 µm bei CD)
  - ⇨ Modulation: 8-auf-16 (statt 8-auf-14 +3 bei CD-ROM)
  - ⇨ andere Fehlerkorrektur mit weniger Overhead: Reed-Solomon Product Code (RSCP) (Fehlerburst von 6 mm Spurlänge korrigierbar)
- Bitrate für Nutzdaten: bis zu 9,8 MBit/s für Video, Audio, Bilder insgesamt



Optische Speichermedien

- Dual/Layer:
  - ⇨ 2 Schichten übereinander (70 % Reflektion und 25-40% Reflektion)
  - ⇨ Der Laser wird auf die jeweils aktuelle Schicht fokussiert
  - ⇨ Am Ende der ersten Schicht springt Laser automatisch auf die zweite um (sie wird im Unterschied zur anderen Schicht von außen nach innen gelesen)

- Speicherkapazität:
  - ⇨ Single/Side Single/Layer: 4,6 GB
  - ⇨ Single/Side Dual/Layer: 8,5 GB
  - ⇨ Dual/Side Single/Layer: 9,4 GB
  - ⇨ Dual/Side Dual/Layer: 17 GB



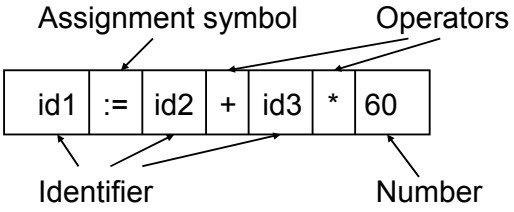
Code Generierung

- **Ziel**: Erzeuge aus einem Quellprogramm (Source) ein Zielprogramm (Target)

Vorgehen: Analyse-Phase gefolgt von einer Synthese-Phase

- Lexikalische Analyse:
  - ⇨ Teile das Source-Programm in Symbole auf
  - ⇨ Reguläre Ausdrücke werden durch endliche Automaten erkannt

position := initial + rate * 60



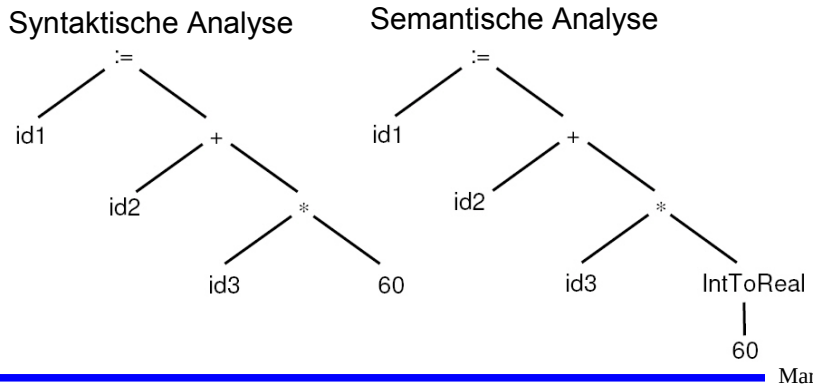
Code Generierung

- Syntaktische Analyse
  - ⇨ Parsen der Symbolfolge und Bestimmung der Sätzen
  - ⇨ Sätze werden durch eine kontextfreie Grammatik beschrieben

    A → Identifier := E

    E → E + E | E * E | Identifier | Number

- Semantische Analyse
  - ⇨ Überprüft das Programm auf Bedeutungskorrektheit
  - ⇨ Beispiel: Prüfung, ob die Datentypen zueinander passen



Code Generierung

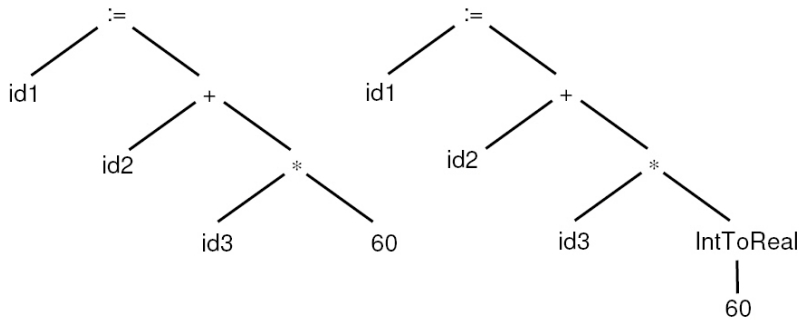
- Code Generierung:

Zwischencode-Generierung → Code-Optimierung → Code Generierung

```
tmp1 := IntToReal(60)
tmp2 := id3 * tmp1
tmp3 := id2 + tmp2
id1 := tmp3
```

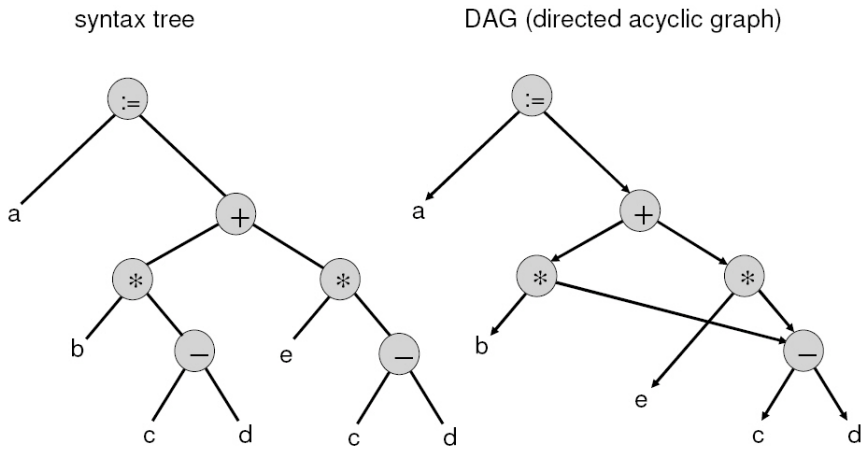
```
tmp1 := id3 * 60.0
id1 := id2 + tmp1
```

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```



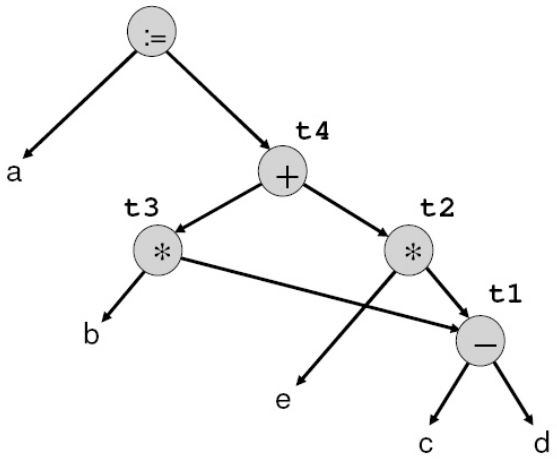
Code Generierung

- a := b * (c - d) + e * (c - d)



Code Generierung

- Generierung von 3-Adress Code



t1 := c – d
t2 := e * t1
t3 := b * t1
t4 := t2 + t3
a:= t4

Code Generierung

- Basic Blocks

  Definition: Ein **Basic-Block** ist eine Folge von Befehlen bei der der Kontrollfluß am Anfang reingeht und am Ende herausgeht er zwischendurch weder stoppt (außer am Ende des Programms) noch verzweigt.

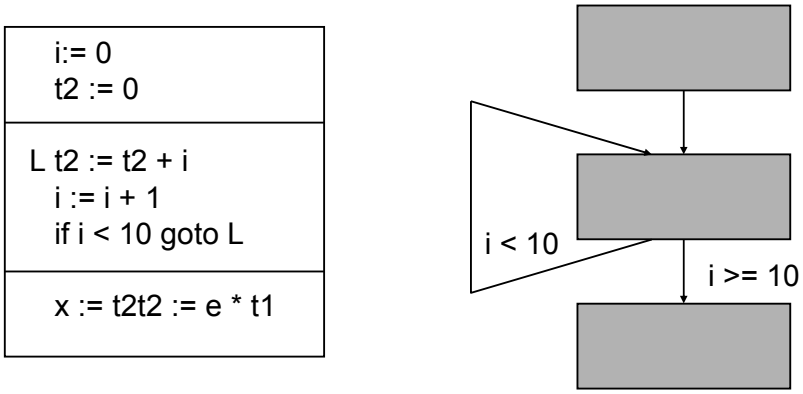
  Ein Folge von 3-Adressbefehlen entspricht einer Menge von Basic-Blocks
  Diese kann folgendermaßen bestimmt werden:

  ⇨ 1. Bestimme den Anfang der Blöcke:
  - Der erste Befehl
  - Ziele von un-/bedingten Sprüngen
  - Befehle, die auf einen un-/bedingten Sprung folgen

  ⇨ 2. Bestimme die Blöcke
  - Für jeden Anfang gibt es einen Block
  - Der Block geht vom Anfang bis vor den Anfang des nächsten Blocks oder zum Ende des Programms

Code Generierung

- „Degenerierter“ Kontrollflußgraph (CFG)
  - ⇨ Knoten sind die Basic-Blocks



Code Generierung

- Verbessere den Code für einen Basic-Blocks mit Hilfe des gerichteten azyklischen Graphen für den Basic-Blocks.
  Eliminiere dabei mehrfach vorkommende gleiche Teilausdrücke und redundante Befehle.

- Definition: Der gerichtete azyklische Graph (DAG) eines Basic-Blocks ist ein Graph wobei gilt
  - ⇨ Blätter werden mit den Namen der Variablen/Konstanten bezeichnet. Variablen mit initialem Wert bekommen den Index 0.
  - ⇨ Innere Knoten werden mit dem Symbol einer Operation bezeichnet (der Operator bestimmt ob der Wert einer Variablen oder ihre Adresse der Operand ist).
  - ⇨ Ein Knoten kann zusätzlich mit einer Folge von Namen von Variablen bezeichnet werden.

Code Generierung

- Erzeuge den DAG eines Basic-Blocks durch einen Vorwärts-Pass über die Befehle folgendermaßen:

  For x := y *op* z;
  - ⇨ Find node labeled y, or create one
  - ⇨ Find node labeled z, or create one
  - ⇨ Create new node for *op*, or find an existing one with descendants y, z (use hash scheme)
  - ⇨ Add x to list of labels for new node
  - ⇨ Remove label x from node on which it appeared

  For x := y;
  - ⇨ Add x to list of labels of node which currently holds y

Code Generierung

- Beispiel:

C-Programm

```
int i, prod, a[20], b[20];
…
prod=0;
i = 0;
do {
  prod = prod + a[i] * b[i];
  i++;
} while (i < 20);
```

3 Adress-Code

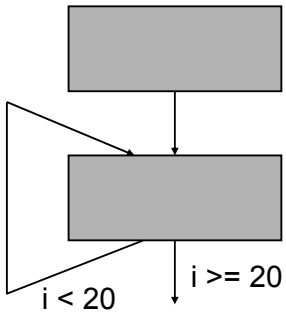
```
(1)  prod := 0
(2)  i := 0
(3)  t1 := 4 * i
(4)  t2 := a[t1]
(5)  t3 := 4 * i;
(6)  t4 := b[t3]
(7)  t5 := t2 * t4
(8)  t6 := prod + t5
(9)  prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i < 20 goto (3)
```

Basic Blocks

```
B1
(1)  prod := 0
(2)  i := 0

B2
(3)  t1 := 4 * i
(4)  t2 := a[t1]
(5)  t3 := 4 * i;
(6)  t4 := b[t3]
(7)  t5 := t2 * t4
(8)  t6 := prod + t5
(9)  prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i < 20 goto (3)
```

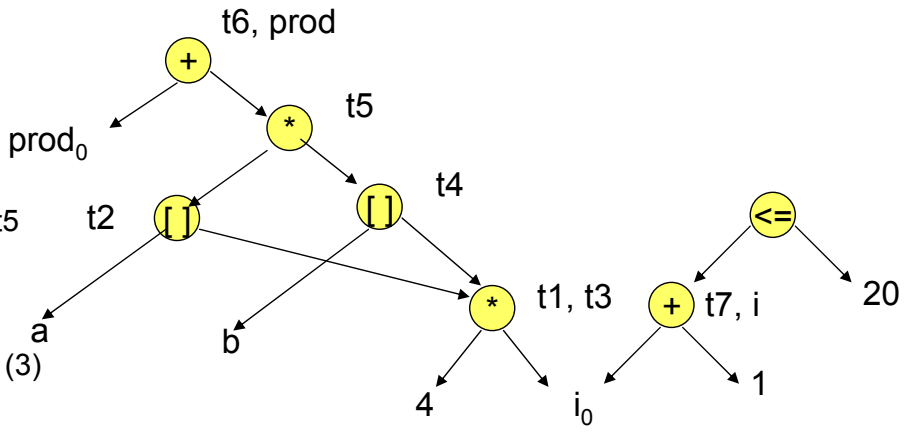
CFG



Code Generierung

- Erzeuge verbesserten Code für einen Block mittels der DAG für den Block mit folgenden Regeln:
  - ⇨ Jede topologische Sortierung des DAG ergibt ein gültige Auswertungsreihenfolge
  - ⇨ Ein Knoten ohne Markierung wird nicht berücksichtigt
  - ⇨ Bevorzuge den Namen einer (normalen) Variablen gegenüber einer temporären Variablen

```
t1 := 4 * i
t2 := a[t1]
t3 := 4 * i;
t4 := b[t3]
t5 := t2 * t4
t6 := prod + t5
prod := t6
t7 := i + 1
i := t7
if i < 20 goto (3)
```




```
t1   := 4 * i;
t2   := a[t1]
t4   := b[t1]
t5   := t2 * t4
prod := prod + t5
i    := i + 1
if i <=20 goto (3)
```


Code Generierung

- **Verwendung von Registern**:
  - ⇨ Befehle, die mit Registern arbeiten können oft in weniger Takten ausgeführt werden als Befehle, die auf den Hauptspeicher zugreifen (CISC)
  - ⇨ Versuche die Anzahl der LOAD-/STORE-Befehle zu minimieren (RISC)

- **Register Allocation**: Bestimme für jeden Zeitpunkt/Takt die Menge der Variablen, die in den Registern gehalten wird

Code Generierung

- **Register Sufficiency Problem:**
  - Gegeben: Eine DAG G=(V,E), V={v_1,…,v_n} für einen Block (wobei jeder Knoten maximal 2 Nachfolger hat) und eine natürliche Zahl k.
  - Frage: Gibt es zulässige Ausführungsreihenfolge der Operationen der DAG, so dass höchstens k Register benötigt werden, d.h. gibt es eine Ordnung v_1,…,v_n der Knoten von V und eine Folge von Teilmengen S_0,…,S_n von V mit
    - $|S_i| \leq k$, S_0 ist leer
    - S_n enthält alle Knoten ohne Vorgänger
    - für $1 \leq i \leq n$ ist v_i in S_i, $S_i - \{v_i\}$ ist Teilmenge von S_{i-1} und
    - S_{i-1} enthält alle Knoten u mit (v_i,u) in E?

  Es gilt: Das Register Sufficiency Problem ist NP-schwer, d.h. es kann nicht in polynomieller Zeit entschieden werden (falls P≠NP)

Code Generierung

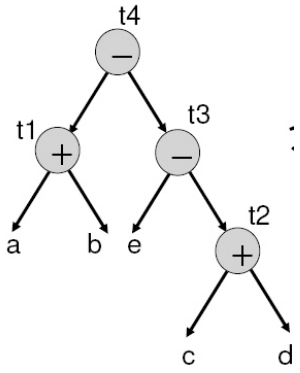
- **Register Assignment**: Ordne die Variablen den einzelnen Registern zu
- **Feasible Register Assignment Problem**:
  - Gegeben: Eine DAG G=(V,E), V={v_1,…,v_n} für einen Block (wobei jeder Knoten maximal 2 Nachfolger hat), eine natürliche Zahl k und eine Registerzuordnung f:V→{R_1,…,R_k}.
  - Frage: Gibt es eine Berechung für G, welche die gegebene Registerzuordnung verwendet, d.h. eine Ordnung v_1,…,v_n der Knoten von V und eine Folge von Teilmengen S_0,…,S_n von V, welche
    - die Bedingungen des Register Allocation Problems erfüllt und
    - zusätzlich die Bedingung erfüllt, dass es für alle $1 \leq j \leq k$, $1 \leq i \leq n$ höchstens ein u in S_i gibt mit $f(u)=R_j$?

  Es gilt: Das Feasible Register Assignment Problem ist NP-schwer, d.h. es kann nicht in polynomieller Zeit entschieden werden (falls P≠NP).

Code Generierung

- **Problem**: Finde zu einer DAG eine Befehlsfolge, so dass sich ein möglichst kurzes entsprechendes Programm ergibt, welches mit LOAD and STORE Operationen ergänzt ist und mit einer gegebenen Anzahl von Registern auskommt.

DAG



mögliche Befehlsfolgen:

```
t1   := a + b
t2   := c + d
t3   := e – t2
t4   := t1 – t3
```

```
t2   := c + d
t3   := e – t2
t1   := a + b
t4   := t1 – t3
```

Annahme: ein Operation der Form x := y + z wird ausgeführt als

```
MOV   y, R0
ADD   z, R0
MOV   R0, x
```


Code Generierung

- Bei Ausführung mit 2 Registern ergeben sich folgende Programme

```
t1   := a + b        →        MOV   a, R0
t2   := c + d                 ADD   b, R0
t3   := e – t2                MOV   c, R1
t4   := t1 – t3               ADD   d, R1
                              MOV   R0, t1
                              MOV   e, R0
                              SUB   R1, R0
                              MOV   t1, R1
                              SUB   R0, R1
                              MOV   R1, t4
```

```
t2   := c + d        →        MOV   c, R0
t3   := e – t2                ADD   d, R0
t1   := a + b                 MOV   e, R1
t4   := t1 – t3               SUB   R0, R1
                              MOV   a, R0
                              ADD   b, R0
                              SUB   R1, R0
                              MOV   R0, t4
```

Code Generierung

- **Code Generation for a 1-Register Machine Problem**:
  - Gegeben: Eine DAG G=(V,E), V={v_1,…,v_n} für einen Block (wobei jeder Knoten maximal 2 Nachfolger hat), eine natürliche Zahl k.
  - Frage: Gibt es ein Programm mit höchstens k Befehlen, dass alle Knoten mit Eingangsgrad 0 auf einer 1-Register-Maschine berechnet, d.h. beginnend mit den Blättern von G im Speicher, sollen nur die Befehle LOAD, STORE sowie Operationen, die den inneren Knoten entsprechen, verwendet werdet. Beachte:
    - Eine Operation für einen Knoten v kann nur dann ausgeführt, wenn für den Wert u im Register gilt: (v,u) ist in E und, falls es einen weiteren Knoten u‘ mit (v,u‘) in E gibt, dann muss u‘ bereits berechnet im Speicher stehen.
    - Nach einer entsprechenden Operation steht v im Register.
    - Ein Wert gilt erst als berechnet, wenn er im Speicher steht.

  Es gilt: Das Code Generation for a 1-Register Machine Problem ist NP-schwer, d.h. es kann nicht in polynomieller Zeit entschieden werden (falls P≠NP).

Code Generierung

- **Problem**: Optimiere Befehlsfolgen bei verschieden aufwendigen Maschinenbefehlen (CISC)

  Betrachte: Maschinenmodell für k-Register Maschine mit verschiedenen Adressierungsarten:

  - Befehle haben das Format op source, destination
    zum Beispiel: MOV R0, a ADD R0, R1

  - Jeder Befehl hat Kosten 1sowie eventuell Zusatzkosten, die von den verwendeten Adressierungen abhängen:

Adressierung:		Zusatzkosten
unmittelbar	`#c`	1
absolute Adressierung	`M`	1
register	`R`	0
indiziert	c(R)	1
indirekt register	(R)	0
indirekt indiziert	(c(R))	1

Code Generierung

- **Definition**: Eine Variable ist **aktiv** an einer Stelle innerhalb eines Basic-Blocks, wenn ihr Wert später noch benötigt wird (evtl. in einem anderen Basic-Block)

  Bestimme: Welche Variablen sind wann aktiv folgendermaßen:

  1. Gehe zum Ende eines Basic-Blocks und bestimme die Variablen, welche am Ausgang des Blocks aktiv sein müssen und den Befehl bei dem sie als nächstes verwendet werden.

  2. Gehe dann jeweils einen Befehl zurück (bis zum Eingang des Basic-Blocks) und führe dabei für jeden Befehl der Form

  (I) x := y op z

  folgende Operation aus

  IF x ist inaktiv, entferne den Befehl

  ELSE i) binde die Information (ob aktiv; wo die nächste Verwendung ist) für x, y, und z an (I);

  ii) setze x inaktiv and setze next-use=none:

  iii) setze y, z auf aktiv und next-use auf (I)

Code Generierung

Ein einfacher Code Generator:

- Behandle jeden Befehl der Form x := y op z folgendermaßen:
  - ⇨ Bestimme den Platz P (Registerplatz oder Speicherplatz) an den der Wert von x geschrieben wird durch einen Aufruf der Funktion getreg(x)
  - ⇨ Bestimme den aktuellen Platz y‘ von y. Falls y‘ nicht P ist erzeuge den Befehl

    MOV y‘, P
  - ⇨ Bestimme den aktuellen Platz z‘ von z und erzeuge

    op P, z‘

- Falls eine Variable im Speicher und in einem Register steht, bevorzuge dass Register

- Die **Funktion getreg()** arbeitet folgendermaßen: Betrachte P = getreg(x)
  - ⇨ falls y in einen Register steht und y keinen next-use hat, dann: return(R)
  - ⇨ falls es ein leeres Register R gibt: return(R)
  - ⇨ falls x einen next-use im Basic-Block hat oder op ein Operator ist, der ein Register benötigt, suche ein benutztes Register R, rette den Inhalt mit MOV R, M und return(R)
  - ⇨ anderenfalls: return(M_x)

Code Generierung

Register-Allokation:

- ⇨ Globale Register-Allokation:
  - Reserviere eine Anzahl von Registern für globale Variable, Schleifenvariablen und Variablen in Basic-Blocks
  - Benutzer-definierte Register Allokation
- ⇨ Graphen-Färbung (innerhalb von Blöcken)
- ⇨ Benutzungs-Zähler (für Schleifen)

- **Graphen-Färbung**
  - ⇨ Führe zunächst eine Register-Allokation mit unbeschränkter Anzahl von Registern durch (d.h. für jede Variable steht ein symbolisches Register zur Verfügung)
  - ⇨ Bestimme die Aktivzeiten der Variablen
  - ⇨ Konstruiere den Registerkonflikt-Graph
  - ⇨ Ordne die symbolischen Register den physikalischen Registern – verwende dabei als Hilfsmittel einer Färbung des Register-Konfliktgraphen.

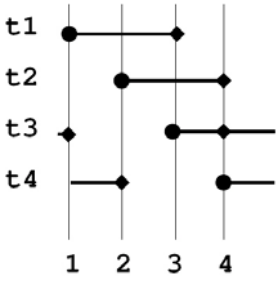
Code Generierung

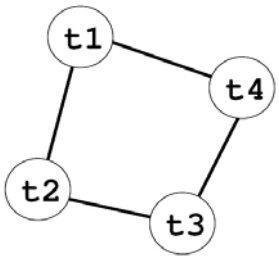
- Zwei Variablen, die überlappende Phasen haben zu denen sie aktiv sind, können nicht das gleiche Register verwenden. Man sagt auch, dass die Variablen einen **Konflikt** haben.

- Definition: Ein **Registerkonflikt-Graph** ist ein ungerichteter Graph G(V,E) wobei die Knotenmenge gleich der Variablenmenge ist und die Kanten Konflikte zwischen den entsprechenden Variablen repräsentieren.

- **Beispiel**:

```
(1)  t1 := t3 + 10
(2)  t2 := t4 + 20
(3)  t3 := t1 +5
(4)  t4 := t2 + t3
```





Code Generierung

- **Ziel**: Färbe die Knoten des Graphen mit möglichst wenigen Farben so, dass zwei benachbarte Knoten nie die gleiche Farbe haben.

- **Heuristik**: Prüfe, ob sich ein Graph G mit k Farben färben lässt

1. IF für alle Knoten v_i in G gilt $deg(v_i) \geq k$, dann STOP (k-Färbung evtl. nicht möglich)
   ELSE wähle einen Knoten v_i in G mit $deg(v_i) < k$

2. Entferne v_i und alle adjazenten Kanten aus G

3. IF G hat keine Knoten mehr hat, ist eine k-Färbung möglich; return „ja“; STOP
   ELSE gehe zu (1)

Färbung: Falls die Antwort ja ist, kann eine Färbung gefunden werden, indem die Knoten in der inversen Ordnung gefärbt werden. Jeder Knoten erhält eine der k Farben, die ungleich den Farben seiner bereits gefärbten Nachbarknoten ist.

Code Generierung

- **Ziel**: Treffe die Entscheidung darüber welche Variablen in Registern stehen so dass in einer Schleife, die aus mehreren Basic Blocks besteht, Kosten gespart werden

Dazu Verwendung von **Benutzungs-Zählern (Usage-Counter)**

Falls eine Variable während der gesamten Ausführung der Schleife in einem Register steht, spart man:

- 1 Kosteneinheit für jede Verwendung von a:
  z.B. verwende ADD R_0, R_1 (Kosten 1) statt ADD a R_2 (Kosten 2)

- 2 Kosteneinheiten am Ende eines Basic-Blocks, wenn die Variable im Block definiert wurde und am Ende aktiv ist
  Begründung: man spart einen Befehl (z.B. MOV R_1 a) um den Wert von a in den Speicher zu bringen

Code Generierung

Entscheide welche Variablen während der gesamten Schleife in Registern bleiben sollen

Annahme: Jeder Basic-Block gleich oft ausgeführt wird und die Schleife wird oft durchlaufen

Approximativ ergeben sich folgende Kosteneinsparungen in der Schleife L für a:

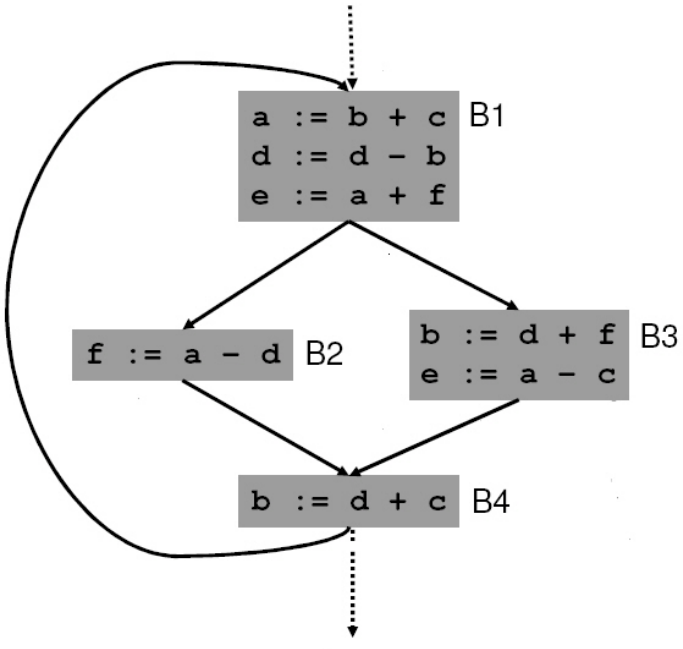
$$\Sigma_{B \in L} used(a,B) + 2 * active(a,B)$$

used(a,B): zählt die Anzahl der Verwendungen von a im Block B bevor a definiert wird

active(a,B): ist 1, wenn a im Block B definiert wurde und am Ende des Blocks aktiv (d.h. dieser Wert wird später noch benötigt) ist, ansonsten ist der Wert 0

Code Generierung

Beispiel:



active(a, B1) = 1
used(a, B2) = 1
used(a, B3) = 1

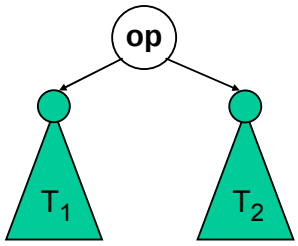
cost savings

a :	4
b :	4
c :	3
d :	6
e :	4
f :	4

for 3 registers:
d ➠ R0, b ➠ R1, a ➠ R2

Code Generierung

- **Code Generierung für Bäume:** **Verwende das Prinzip der dynamischen Programmierung**



Berechne den optimalen (d.h. mit minimalen Kosten) Code für E := (T_1 op T_2)

i) optimaler Code für T_1 und T_2

ii) optimaler Code für E ergibt sich dann

indem der bessere der Codes T_1, T_2 , op und T_2 ,T_1, op gewählt wird

Die Methode hat drei Phasen:

1. Berechne Kostenvektoren für die Knoten des Baumes
2. Berechne daraus die optimale Befehlsfolge
3. Generiere den Code

Der **Kostenvektor** C für einen Knoten v gibt an:

C[0] optimale Kosten zur Berechnung von v, wenn das Resultat im Speicher steht

C[i] optimale Kosten zur Berechnung von v, wenn maximal i Register benutzt werden und das Resultat in einem Register steht

Code Generierung

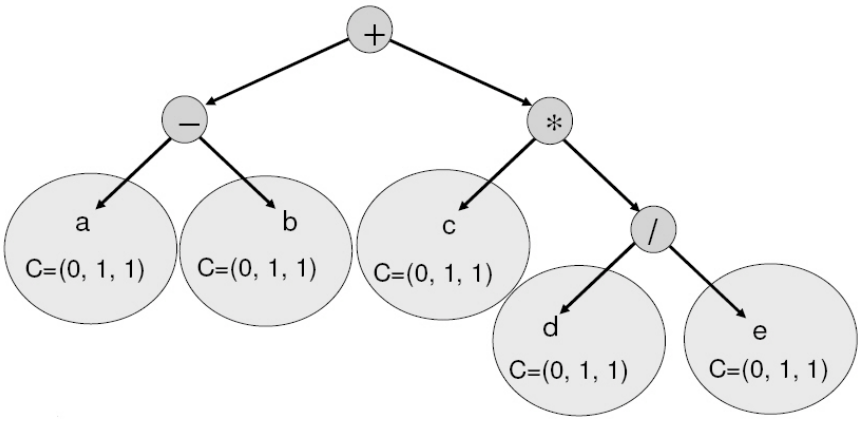
- **Maschinenmodell:**
  - ⇨ n Register R_1, …, R_n
  - ⇨ Befehle der Form R_i := E, wobei E ein Ausdruck ist, der beliebig viele Register und Speicherplätze benutzt
  - ⇨ Falls E mehr als ein Register benutzt, muss R_i eines davon sein
  - ⇨ LOAD: R_i := M, STORE: M := R_i, COPY: R_i := R_i
  - ⇨ Vereinfachung für das folgende Beispiel: Zusätzlich zu LOAD, STORE, COPY nur die Befehle:

$R_i := R_i$ op R_j

$R_i :=$ op M

Code Generierung

- **Beispiel:**



Kosten zur Berechnung von a:

i) in den Speicher: C[0]=0, da a bereits im Speicher ist

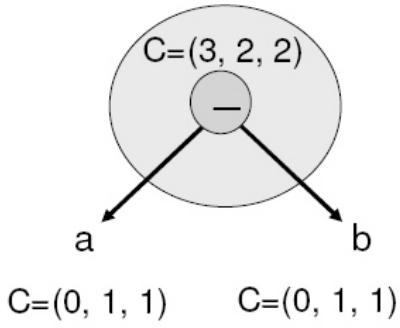
ii) mit einem Register: C[1]=1 mittels $R_i := M$

iii) mit zwei Registern: C[2]=1 mittels $R_i := M$

Analog: Kosten zur Berechung von b, …, e

Code Generierung

Kosten zur Berechnung von „-“:



i) mit einem Register

$R_i := R_i – M$: berechne den linken Teilbaum mit einem Register
und den rechten Teilbaum in den Speicher: C[1]=2

ii) mit zwei Registern

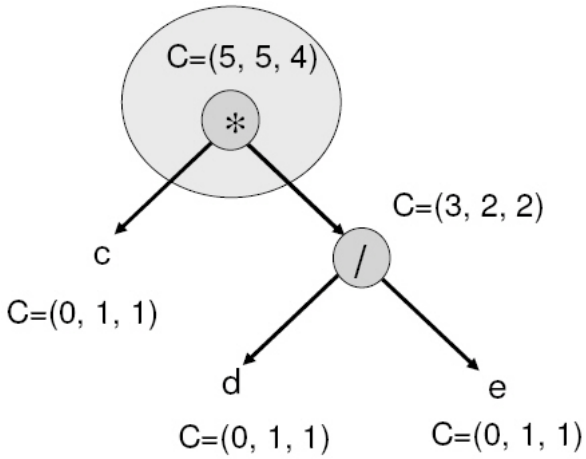
$R_i := R_i – M$: wie bei i)

$R_i := R_i – Rj$: berechne den linken Teilbaum mit zwei Registern
und den rechten Teilbaum mit einem Register: Kosten 3
oder berechne den linken Teilbaum mit einem Register
und den rechten Teilbaum mit zwei Registern: Kosten 3
➔ C[2]=2

iii) in den Speicher: C[0]=3

Code Generierung

Kosten zur Berechnung von „*“:



i) mit einem Register
$R_i := R_i * M$: berechne den linken Teilbaum mit einem Register und den rechten Teilbaum in den Speicher: C[1]=5

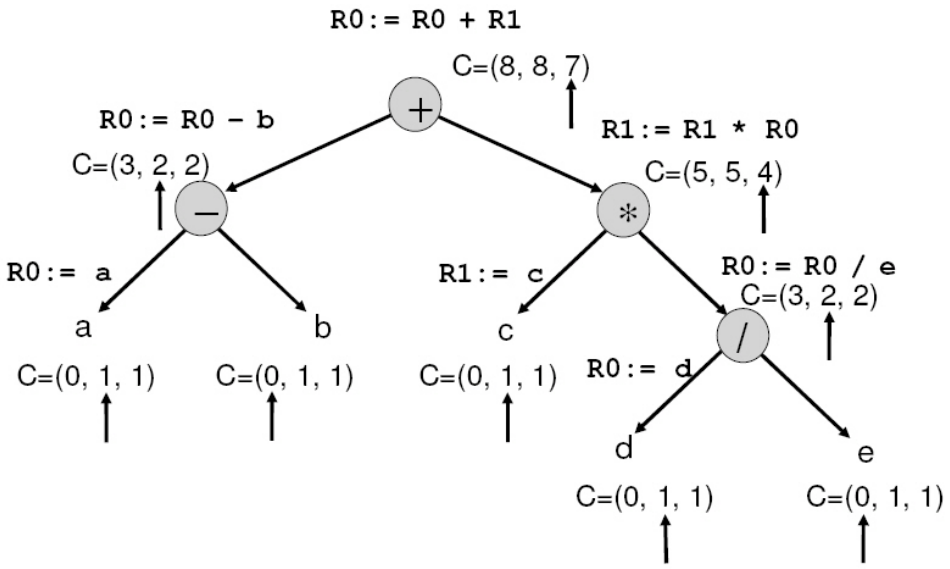
ii) mit zwei Registern
$R_i := R_i * M$: wie bei i)

$R_i := R_i * Rj$: berechne den linken Teilbaum mit zwei Registern und den rechten Teilbaum mit einem Register: Kosten 4
oder berechne den linken Teilbaum mit einem Register und den rechten Teilbaum mit zwei Registern: Kosten 4
➔ C[2]=4

iii) in den Speicher: C[0]=5

Code Generierung

Befehle für kostenminimale Lösung:



Zur Codegenerierung vgl. Vorlesung M. Platzner, Paderborn

Vielen Dank für Ihre Aufmerksamkeit
und viel Erfolg bei Klausur!