64-040 Modul IP7: Rechnerstrukturen

7. Schaltfunktionen und Schaltnetze

Norman Hendrich & Jianwei Zhang

Universität Hamburg
MIN Fakultät, Department Informatik
Vogt-Kölln-Str. 30, D-22527 Hamburg
{hendrich,zhang}@informatik.uni-hamburg.de

WS 2010/2011

Inhalt

Schaltfunktionen

- **Schaltfunktion**: eine eindeutige Zuordnungsvorschrift f, die jeder Wertekombination $(b_1, b_2, \ldots, b_n)$ von Schaltvariablen einen Wert zuweist:

$$y = f(b_1, b_2, \ldots, b_n) \in \{0, 1\}$$

- *Schaltvariable*: eine Variable, die nur endlich viele Werte annehmen kann. Typisch sind binäre Schaltvariablen.
- *Ausgangsvariable*: die Schaltvariable am Ausgang der Funktion, die den Wert y annimmt.
- bereits bekannt: *elementare Schaltfunktionen* (AND, OR, usw.)
- wichtig: wir betrachten jetzt Funktionen von n Variablen

Beschreibung von Schaltfunktionen

- textuelle Beschreibungen
  formale Notation, Schaltalgebra, Beschreibungssprachen

- tabellarische Beschreibungen
  Funktionstabelle, KV-Diagramme, ...

- graphische Beschreibungen
  Kantorovic-Baum (Datenflussgraph), Schaltbild, ...

- Verhaltensbeschreibungen („was“)
- Strukturbeschreibungen („wie“)

Funktionstabelle (1)

- Tabelle mit Eingängen x_i und Ausgangswert $y = f(x)$
- Zeilen im Binärcode sortiert
- zugehöriger Ausgangswert eingetragen

x_3	x_2	x_1	f(x)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Funktionstabelle (2)

- Kurzschreibweise: nur die Funktionswerte notiert
- $f(x_2, x_1, x_0) = \{0, 0, 1, 1, 0, 0, 1, 0\}$

- n Eingänge: Funktionstabelle umfasst 2^n Einträge
- Speicherbedarf wächst exponentiell mit n
- z.B.: 2^{33} Bit für 16-bit Addierer (16+16+1 Eingänge)
- daher nur für kleine Funktionen geeignet

- Erweiterung auf don't-care Terme: s.u.

Verhaltensbeschreibung

- Beschreibung einer Funktion als Text über ihr Verhalten
- umgangssprachliche Formulierungen oft mehrdeutig
- logische Ausdrücke in Programmiersprachen
- Einsatz spezieller (Hardware-) Beschreibungssprachen
  z.B.: Verilog, VHDL, SystemC

umgangssprachlich: Mehrdeutigkeit

„Das Schiebedach ist ok (y), wenn der Öffnungskontakt (x_0) **oder** der Schließkontakt (x_1) funktionieren **oder beide nicht** aktiv sind (Mittelstellung des Daches).“

zwei mögliche Missverständnisse:

oder: als OR oder XOR?

beide nicht: x_1 und x_0 nicht, oder x_1 nicht und x_2 nicht?

je nach Interpretation völlig unterschiedliche Schaltung

(H.D. Wuttke & K. Henke, Schaltsysteme)

Strukturbeschreibung

- **Strukturbeschreibung**: eine Spezifikation der konkreten Realisierung einer Schaltfunktion

- vollständig geklammerte algebraische Ausdrücke
  $f = x_1 \oplus (x_2 \oplus x_3)$
- Datenflußgraphen
- Schaltpläne mit Gattern (s.u.)
- PLA-Format für zweistufige AND-OR Schaltungen (s.u.)
- ...

Funktional vollständige Basismenge

- Menge M von Verknüpfungen über $GF(2)$ heisst **funktional vollständig**, wenn die Funktionen $f, g \in T_2$:

  $f(x_1, x_2) = x_1 \oplus x_2$

  $g(x_1, x_2) = x_1 x_2$

  allein mit den in M enthaltenen Verknüpfungen geschrieben werden können.

- Boole'sche Algebra: { AND, OR, NOT }
- Reed-Muller-Form: { AND, XOR, 1 }
- technisch relevant: { NAND }, { NOR }

Normalformen

- Jede Funktion kann auf beliebig viele Arten beschrieben werden

Suche nach Standardformen:

- in denen man alle Funktionen darstellen kann
- Darstellung mit universellen Eigenschaften
- eindeutige Repräsentation (einfache Überprüfung, ob gegebene Funktionen übereinstimmen)

- Beispiel: Darstellung von reellen Funktionen als Potenzreihe
  $f(x) = \sum_{i=0}^{\infty} a_i x^i$

Normalformen: Analogie zur Potenzreihe

- Darstellung von reellen Funktionen als Potenzreihe
  $f(x) = \sum_{i=0}^{\infty} a_i x^i$

Normalform einer Boole'schen Funktion:

- analog zur Potenzreihe
- als Summe über Koeffizienten $\{0, 1\}$ und Basisfunktionen

$$f = \sum_{i=1}^{2^n} \hat{f}_i \hat{B}_i, \quad \hat{f}_i \in \mathrm{GF}(2)$$

mit $\hat{B}_1, \ldots, \hat{B}_{2^n}$ einer Basis des T^n

Definition: Normalform

- funktional vollständige Menge V der Verknüpfungen von $\{0, 1\}$
- Seien $\oplus, \otimes \in V$ und assoziativ

- Wenn sich alle $f \in T^n$ in der Form

$$f = (\hat{f}_1 \otimes \hat{B}_1) \oplus \cdots \oplus (\hat{f}_{2^n} \otimes \hat{B}_{2^n})$$

schreiben lassen, so wird die Form als **Normalform** und die Menge der $\hat{B}_i$ als **Basis** bezeichnet.

Menge von 2^n Basisfunktionen $\hat{B}_i$

Menge von 2^{2^n} möglichen Funktionen f

Disjunktive Normalform (DNF)

- **Minterm**: die UND-Verknüpfung **aller** Schaltvariablen einer Schaltfunktion. Die Variablen dürfen dabei negiert oder nicht negiert auftreten.

- **Disjunktive Normalform**: die disjunktive Verknüpfung aller Minterme m mit dem Funktionswert 1.

$$f = \bigvee_{i=1}^{2^n} \hat{f}_i \cdot m(i), \quad \text{mit} \quad m(i): \ \text{Minterm}(i)$$

auch: *kanonische disjunktive Normalform*

englisch: *sum-of-products* (SOP)

Disjunktive Normalform: Minterme

- Beispiel: alle 2^3 Minterme für drei Variablen
- jeder Minterm nimmt nur für eine Belegung der Eingangsvariablen den Wert 1 an

x_3	x_2	x_1	Minterme
0	0	0	$\overline{x}_3 \wedge \overline{x}_2 \wedge \overline{x}_1$
0	0	1	$\overline{x}_3 \wedge \overline{x}_2 \wedge x_1$
0	1	0	$\overline{x}_3 \wedge x_2 \wedge \overline{x}_1$
0	1	1	$\overline{x}_3 \wedge x_2 \wedge x_1$
1	0	0	$x_3 \wedge \overline{x}_2 \wedge \overline{x}_1$
1	0	1	$x_3 \wedge \overline{x}_2 \wedge x_1$
1	1	0	$x_3 \wedge x_2 \wedge \overline{x}_1$
1	1	1	$x_3 \wedge x_2 \wedge x_1$

Disjunktive Normalform: Beispiel

x_3	x_2	x_1	f(x)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

- Zeilen der Funktionstabelle entsprechen jeweiligem Minterm
- für f sind nur drei Koeffizienten der DNF gleich 1, also:
- DNF: $f(x) = (\overline{x}_3 \wedge x_2 \wedge \overline{x}_1) \vee (\overline{x}_3 \wedge x_2 \wedge x_1) \vee (x_3 \wedge x_2 \wedge \overline{x}_1)$

Allgemeine disjunktive Form

- **disjunktive Form** (sum-of-products): die disjunktive Verknüpfung (ODER) von Termen. Jeder Term besteht aus der UND-Verknüpfung von Schaltvariablen, die entweder direkt oder negiert auftreten können.
- zum Beispiel durch Zusammenfassen („Minimierung") von Termen aus der disjunktiven Normalform. Beispiel:

DNF $\quad f(x) = (\overline{x}_3 \wedge x_2 \wedge \overline{x}_1) \vee (\overline{x}_3 \wedge x_2 \wedge x_1) \vee (x_3 \wedge x_2 \wedge \overline{x}_1)$

minimierte disjunktive Form $\quad f(x) = (\overline{x}_3 \wedge x_2) \vee (x_3 \wedge x_2 \wedge \overline{x}_1)$

- disjunktive Form ist nicht eindeutig (keine Normalform)

Konjunktive Normalform (KNF)

- **Maxterm**: die ODER-Verknüpfung **aller** Schaltvariablen einer Schaltfunktion. Wiederum dürfen die Variablen negiert und nicht negiert werden.

- **Konjunktive Normalform**: die konjunktive Verknüpfung aller Maxterme mit dem Funktionswert 0

$$f = \bigwedge_{i=1}^{2^n} \hat{f}_i \cdot \mu(i), \quad \text{mit} \quad \mu(i): \; \text{Maxterm}(i)$$

auch: *kanonische konjunktive Normalform*

englisch: *product-of-sums* (POS)

Konjunktive Normalform: Maxterme

- Beispiel: alle 2^3 Maxterme für drei Variablen
- jeder Maxterm nimmt nur für eine Belegung der Eingangsvariablen den Wert 0 an

x_3	x_2	x_1	Maxterme
0	0	0	$x_3 \vee x_2 \vee x_1$
0	0	1	$x_3 \vee x_2 \vee \overline{x}_1$
0	1	0	$x_3 \vee \overline{x}_2 \vee x_1$
0	1	1	$x_3 \vee \overline{x}_2 \vee \overline{x}_1$
1	0	0	$\overline{x}_3 \vee x_2 \vee x_1$
1	0	1	$\overline{x}_3 \vee x_2 \vee \overline{x}_1$
1	1	0	$\overline{x}_3 \vee \overline{x}_2 \vee x_1$
1	1	1	$\overline{x}_3 \vee \overline{x}_2 \vee \overline{x}_1$

Konjunktive Normalform: Beispiel

x_3	x_2	x_1	f(x)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

- Zeilen der Funktionstabelle ≈ „invertierter“ Maxterm
- für f sind fünf Koeffizienten der KNF gleich 0, also:

$$\begin{aligned} f(x) \quad = \quad & (x_3 \vee x_2 \vee x_1) \wedge (x_3 \vee x_2 \vee \overline{x}_1) \wedge (\overline{x}_3 \vee x_2 \vee x_1) \wedge \\ & (\overline{x}_3 \vee x_2 \vee \overline{x}_1) \wedge (\overline{x}_3 \vee \overline{x}_2 \vee \overline{x}_1) \end{aligned}$$

Allgemeine konjunktive Form

- **konjunktive Form** (product-of-sums): die disjunktive Verknüpfung (UND) von Termen. Jeder Term besteht aus der ODER-Verknüpfung von Schaltvariablen, die entweder direkt oder negiert auftreten können.
- zum Beispiel durch Zusammenfassen („Minimierung") von Termen aus der konjunktiven Normalform.

$$\begin{aligned} f(x) \quad = \quad & (x_3 \vee x_2 \vee x_1) \wedge (x_3 \vee x_2 \vee \overline{x}_1) \wedge (\overline{x}_3 \vee x_2 \vee x_1) \wedge \\ & (\overline{x}_3 \vee x_2 \vee \overline{x}_1) \wedge (\overline{x}_3 \vee \overline{x}_2 \vee \overline{x}_1) \end{aligned}$$

minimierte konjunktive Form
$f(x) = (x_2 \vee x_1) \wedge (x_3 \vee x_2) \wedge (\overline{x}_3 \vee \overline{x}_1)$

- konjunktive Form ist nicht eindeutig (keine Normalform)

Reed-Muller-Form

- **Reed-Muller-Form**: die additive Verknüpfung aller Reed-Muller-Terme mit dem Funktionswert 1

$$f = \bigoplus_{i=1}^{2^n} \hat{f}_i \cdot RM(i),$$

- mit den Reed-Muller Basisfunktionen $RM(i)$
- Erinnerung: Addition im GF(2) ist die XOR-Operation

Reed-Muller-Form: Basisfunktionen

- Basisfunktionen sind:

$\{1\},$ (0 Variablen)
$\{1, x_1\},$ (1 Variable)
$\{1, x_1, \quad x_2, x_2x_1\},$ (2 Variablen)
$\{1, x_1, x_2, x_2x_1, \quad x_3, x_3x_1, x_3x_2, x_3x_2x_1\},$ (3 Variablen)
...
$\{RM(n-1), \quad x_n \cdot RM(n-1)\}$ (n Variablen)

- rekursive Bildung: bei n bit alle Basisfunktionen von $(n-1)$-bit und zusätzlich das Produkt von x_n mit den Basisfunktionen von $(n-1)$-bit.

Reed-Muller-Form: Umrechnung

Umrechnung von gegebenem Ausdruck in Reed-Muller Form?

- Ersetzen der Negation: $\overline{a} = a \oplus 1$
- Ersetzen der Disjunktion: $a \vee b = a \oplus b \oplus ab$
- Ausnutzen von: $a \oplus a = 0$

Beispiel:

$$\begin{aligned}
f(x_1, x_2, x_3) &= (\overline{x}_1 \vee x_2)x_3 \\
&= (\overline{x}_1 \oplus x_2 \oplus \overline{x}_1 x_2)x_3 \\
&= ((1 \oplus x_1) \oplus x_2 \oplus (1 \oplus x_1)x_2)x_3 \\
&= (1 \oplus x_1 \oplus x_2 \oplus x_2 \oplus x_1 x_2)x_3 \\
&= x_3 \oplus x_1 x_3 \oplus x_1 x_2 x_3
\end{aligned}$$

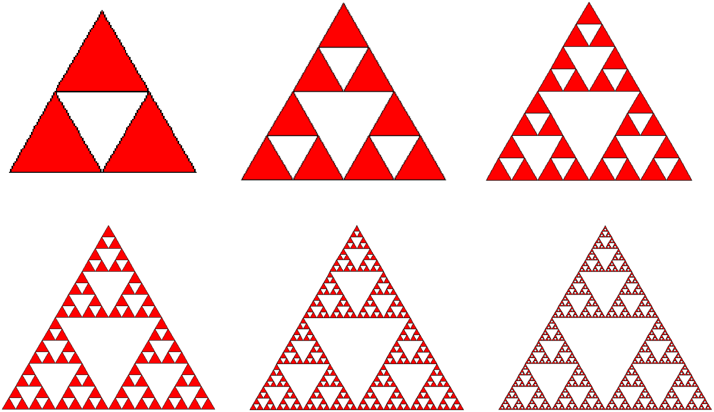
Reed-Muller-Form: Transformationsmatrix

- lineare Umrechnung zwischen Funktion f bzw. Funktionstabelle (DNF) und RMF R
- Transformationsmatrix A kann rekursiv definiert werden (wie die RMF-Basisfunktionen)
- Multiplikation von A mit f ergibt den Koeffizientenvektor r der RMF

$$r = A \cdot f, \quad \text{und} \quad f = A \cdot r$$

- weitere Details in (von der Heide: Technische Informatik T1)
- Hinweis: Beziehung zu Fraktalen (Sirpinski-Dreieck)

Exkurs: Sirpinski-Dreieck (Fraktal)



(v.d.Heide, Technische Informatik T1, `demosirpinski(n)`)

Reed-Muller-Form: Umrechnung

- $r = A \cdot f \quad$ (und $A \cdot A = I$, also $f = A \cdot r$ (!))

$$A_0 = (1)$$

$$A_1 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

$$A_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$\dots$

Reed-Muller-Form: Umrechnung

...

$$A_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

...

$$A_n = \begin{pmatrix} A_{n-1} & 0 \\ A_{n-1} & A_{n-1} \end{pmatrix}$$

Reed-Muller-Form: Beispiel

x_3	x_2	x_1	f(x)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

- Berechnung durch Rechenregeln der Boole'schen Algebra oder Aufstellen von A_3 und Ausmultiplizieren:

$$f(x) \quad = \quad x_2 \oplus x_3 x_2 x_1$$

- häufig kompaktere Darstellung als DNF oder KNF

Reed-Muller-Form: Beispiel

- $f(x_3, x_2, x_1) = \{0, 0, 1, 1, 0, 0, 1, 0\}$ (Funktionstabelle)
- Aufstellen von A_3 und Ausmultiplizieren

$$r = A_3 \cdot f = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

- gesuchte RMF ist $f(x_3, x_2, x_1) = r \cdot RM(3) = x_2 \oplus x_3 x_2 x_1$

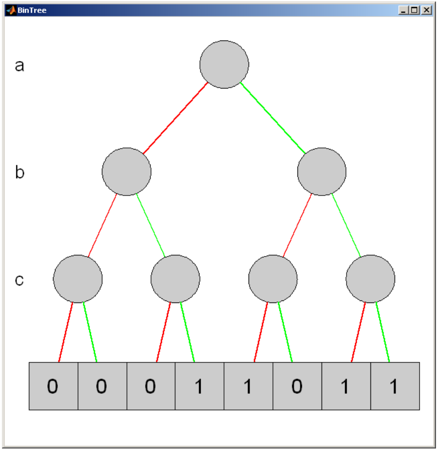
Graphische Darstellung: Entscheidungsbäume

- Darstellung einer Schaltfunktion als Baum/Graph
- jeder Knoten ist einer Variablen zugeordnet
- Verzweigung entsprechend einer `if-then-else`-Entscheidung (grün: 1-Zweig, rot: 0-Zweig)

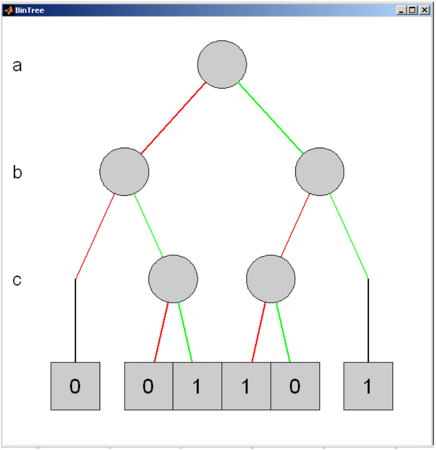
- vollständiger Baum entspricht Funktionstabelle
- Vorteil: Entfernen bzw. Zusammenfassen redundanter Knoten möglich

Entscheidungsbaum: Beispiel Multiplexer

$f(a, b, c) = (a \wedge \overline{c}) \vee (b \wedge c)$



Entscheidungsbaum: Beispiel Multiplexer



Reduced Ordered Binary-Decision Diagrams (ROBDD)

Binäres Entscheidungsdiagramm

- Variante des Entscheidungsbaums
- vorab gewählte Variablenordnung (*ordered*)
- redundante Knoten werden entfernt (*reduced*)
- ein ROBDD ist eine Normalform für eine Funktion

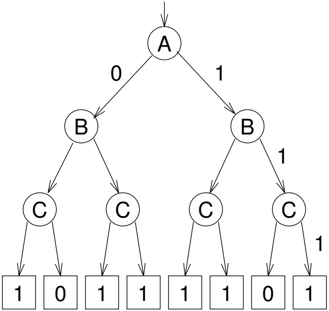
- viele praxisrelevante Funktionen sehr kompakt darstellbar, $O(n)..O(n^2)$ Knoten bei n Variablen
- wichtige Ausnahme: n-bit Multiplizierer ist $O(2^n)$
- derzeit das Standardverfahren zur Manipulation von (großen) Schaltfunktionen

R. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, IEEE Trans. Computers (1986)

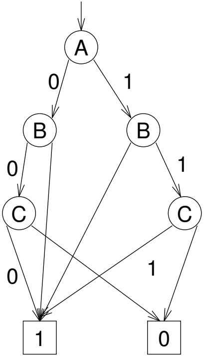
ROBDD: vs. Entscheidungsbaum

Entscheidungsbaum

$f = (a\,b\,c) \vee (a\,\overline{b}) \vee (\overline{a}\,b) \vee (\overline{a}\,\overline{b}\,\overline{c})$

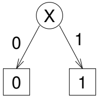


ROBDD

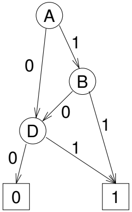


ROBDD: Beispiele

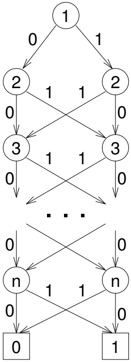
$f(x) = x$



$g = (a\, b) \vee d$



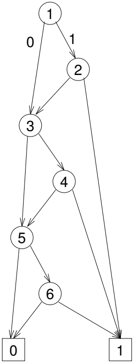
Parität $p = x_1 \oplus x_2 \oplus \ldots x_n$



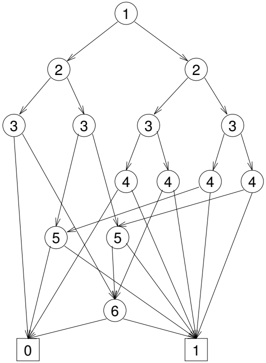
ROBDD: Problem der Variablenordnung

Anzahl der Knoten oft stark abhängig von der gewählten Variablenordnung

$f = x_1\, x_2 \vee x_3\, x_4 \vee x_5\, x_6$



$g = x_1\, x_4 \vee x_2\, x_5 \vee x_3\, x_6$



Minimierung von Schaltfunktionen

- mehrere (beliebig viele) Varianten zur Realisierung einer gegebenen Schaltfunktion bzw. eines Schaltnetzes

Minimierung des Realisierungsaufwandes:

- diverse Kriterien, technologieabhängig

- Hardwarekosten (Anzahl der Gatter)
- Hardwareeffizienz (z.B. NAND statt XOR)
- Geschwindigkeit (Anzahl der Stufen, Laufzeiten)
- Testbarkeit (Erkennung von Produktionsfehlern)
- Robustheit (z.B. ionisierende Strahlung)

Algebraische Minimierungsverfahren

- Vereinfachung der gegebenen Schaltfunktionen durch Anwendung der Gesetze der Boole'schen Algebra

- im allgemeinen nur durch Ausprobieren
- ohne Rechner sehr mühsam
- keine allgemeingültigen Algorithmen bekannt

- Heuristik: Suche nach *Primimplikanten*
- Quine-McCluskey-Verfahren und Erweiterungen

Algebraische Minimierung: Beispiel

Ausgangsfunktion in DNF:

$$\begin{aligned} y(x) &= x_3\overline{x_2}x_1\overline{x_0} \vee x_3\overline{x_2}x_1x_0 \\ &\vee x_3x_2x_1x_0 \vee x_3x_2\overline{x_1}x_0 \\ &\vee x_3\overline{x_2x_1}x_0 \vee \overline{x_3}x_2x_1x_0 \\ &\vee \overline{x_3}x_2x_1\overline{x_0} \vee x_3x_2x_1\overline{x_0} \end{aligned}$$

Zusammenfassen benachbarter Terme liefert:

$$y(x) = x_3\overline{x_2}x1 \vee x_3x_2x_0 \vee x_2x_1\overline{x_0} \vee x_3\overline{x_1}x_0 \vee \overline{x_3}x_2x_1$$

Aber bessere Lösung ist möglich:

$$y(x) = x_3x_0 \vee x_3x_1 \vee x_2x_1$$

Graphische Minimierungsverfahren

- Darstellung einer Schaltfunktion im KV-Diagramm
- Interpretation als disjunktive Normalform

- Zusammenfassen benachbarter Terme durch **Schleifen**
- alle 1-Terme mit möglichst wenigen Schleifen abdecken
- Ablesen der minimierten Funktion, wenn keine weiteren Schleifen gebildet werden können

- beruht auf der menschlichen Fähigkeit, benachbarte Flächen auf einen Blick zu „sehen“
- bei mehr als 6 Variablen nicht mehr praktikabel

Erinnerung: Karnaugh-Veitch-Diagramm

x3 x2 \ x1 x0	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

x3 x2 \ x1 x0	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

- 2D-Diagramm mit $2^n = 2^{n_y} \times 2^{n_x}$ Feldern
- gängige Größen sind 2x2 2x4 4x4
- Anordnung der Indizes ist im Gray-Code (!)
- benachbarte Felder unterscheiden sich gerade um 1 Bit

KV-Diagramm: für zwei und drei Variablen

x1 \ x0	0	1
0	00	01
1	10	11

x2 \ x1 x0	00	01	11	10
0	000	001	011	010
1	100	101	111	110

x3 x2 \ x1 x0	00	01	11	10
00	0000	0001	0011	0010
01	0100	0101	0111	0110
11	1100	1101	1111	1110
10	1000	1001	1011	1010

KV-Diagramm für Schaltfunktionen

- Funktionswerte in zugehöriges Feld im KV-Diagramm eintragen
- Werte 0 und 1, ggf. don't-care (∗)
- 2D-Äquivalent zur Funktionstabelle

- praktikabel für 3..6 Eingänge
- fünf Eingänge: zwei Diagramme a 4x4 Felder
- sechs Eingänge: vier Diagramme a 4x4 Felder

- viele Strukturen „auf einen Blick" erkennbar

KV-Diagramm: Zuordnung zur Funktionstabelle

x1	x0	y=f(x)
0	0	f(0 0)
0	1	f(0 1)
1	0	f(1 0)
1	1	f(1 1)

x1 \ x0	0	1
0	00	01
1	10	11

x2	x1	x0	y=f(x)
0	0	0	f(0 0 0)
0	0	1	f(0 0 1)
0	1	0	f(0 1 0)
0	1	1	f(0 1 1)
1	0	0	f(1 0 0)
1	0	1	f(1 0 1)
1	1	0	f(1 1 0)
1	1	1	f(1 1 1)

x2 \ x1 x0	00	01	11	10
0	000	001	011	010
1	100	101	111	110

KV-Diagramm: Eintragen aus Funktionstabelle

x1	x0	y=f(x)
0	0	0
0	1	1
1	0	1
1	1	0

x1 \ x0	0	1
0	0	1
1	1	0

x2	x1	x0	y=f(x)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

x2 \ x1 x0	00	01	11	10
0	0	0	1	1
1	0	0	0	1

KV-Diagramm: Beispiel

x3 x2 \ x1 x0	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

x3 x2 \ x1 x0	00	01	11	10
00	**1**	**0**	**0**	**1**
01	**0**	**0**	**0**	**0**
11	**0**	**0**	**1**	**0**
10	**0**	**0**	**1**	**0**

- Beispielfunktion in DNF mit vier Termen:
  $f(x) = (x_3 x_2 x_1 x_0) \vee (x_3 \overline{x}_2 x_1 x_0) \vee (\overline{x}_3 \overline{x}_2 \overline{x}_1 \overline{x}_0) \vee (\overline{x}_3 \overline{x}_2 x_1 \overline{x}_0)$
- Werte aus Funktionstabelle an entsprechender Stelle ins Diagramm eintragen

Schleifen: Zusammenfassen benachbarter Terme

- benachbarte Felder unterscheiden sich um 1-Bit
- falls benachbarte Terme beide 1 sind: Funktion hängt an dieser Stelle nicht von der betroffenen Variable ab
- zugehörige (Min-) Terme können zusammengefasst werden

- Erweiterung auf vier benachbarte Felder (4x1 1x4 2x2)
- Erweiterung auf acht benachbarte Felder (4x2 2x4) usw.
- aber keine Dreier- Fünfergruppen, usw.

- Nachbarschaft auch „außenherum“
- mehrere Schleifen dürfen sich überlappen

Schleifen: Ablesen der Schleifen (1)

x1	x0	y=f(x)
0	0	0
0	1	0
1	0	1
1	1	1

x1 \ x0	0	1
0	0	0
1	1	1

x2	x1	x0	y=f(x)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

x2 \ x1 x0	00	01	11	10
0	0	0	1	1
1	0	0	0	1

oben: $f(x_1, x_0) = x_1$ unten: $f(x_2, x_1, x_0) = \overline{x_2}\, x_1 \vee x_1\, \overline{x_0}$

Schleifen: Ablesen der Schleifen (2)

x3 x2 \ x1 x0	00	01	11	10
00	**1**	**0**	**0**	**1**
01	**0**	**0**	**0**	**0**
11	**0**	**0**	**1**	**0**
10	**0**	**0**	**1**	**0**

x3 x2 \ x1 x0	00	01	11	10
00	**1**	**0**	**0**	**1**
01	**0**	**0**	**0**	**0**
11	**0**	**0**	**1**	**0**
10	**0**	**0**	**1**	**0**

- insgesamt zwei Schleifen möglich
- rot entspricht $(x_3 x_1 x_0) = (x_3 x_2 x_1 x_0) \vee (x_3 \overline{x}_2 x_1 x_0)$
- grün entspricht $(\overline{x}_3 \overline{x}_2 \overline{x}_0) = (\overline{x}_3 \overline{x}_2 \overline{x}_1 \overline{x}_0) \vee (\overline{x}_3 \overline{x}_2 x_1 \overline{x}_0)$
- minimierte disjunktive Form: $f(x) = (x_3 x_1 x_0) \vee (\overline{x}_3 \overline{x}_2 \overline{x}_0)$

Schleifen: Interaktive Demonstration

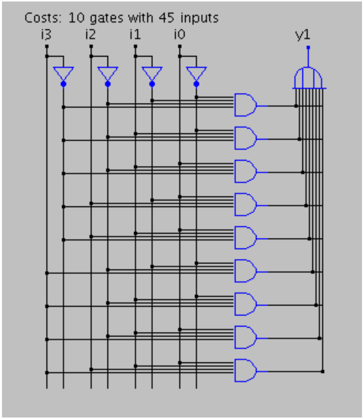
- Applet zur Minimierung mit KV-Diagrammen:
  tams-www.informatik.uni-hamburg.de/applets/kvd/kvd.html

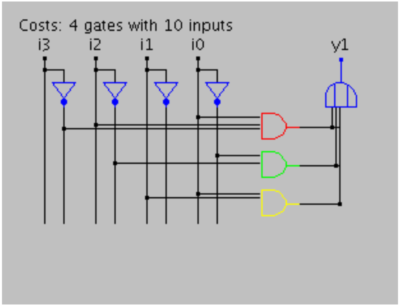
1. Auswahl oder Eingabe einer Funktion (2..6 Variablen)
2. Interaktives Setzen und Erweitern von Schleifen
   (click, shift+click, control+click)
3. Anzeige der zugehörigen Hardwarekosten und Schaltung

- Achtung: andere Anordnung der Eingangsvariablen als im Skript
- entsprechend andere Anordnung der Terme im KV-Diagramm
- Prinzip bleibt aber gleich

KV-Diagramm Applet: Screenshots

KV-Diagramm Applet: zugehörige Hardwarekosten



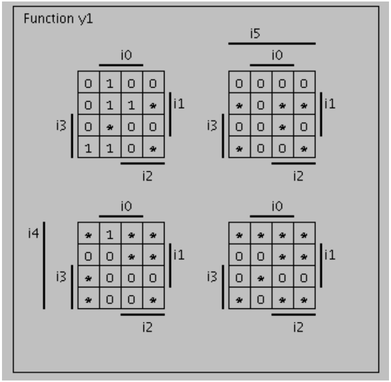


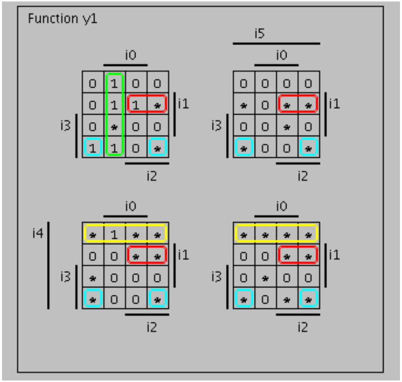
Minimierung mit Don't-Care Termen

- in der Praxis: viele Schaltfunktionen unvollständig definiert
- weil bestimmte Eingangskombinationen nicht vorkommen
- zugehörige Terme als **Don't Care** markieren
  (typisch: Sternchen $*$ in Funktionstabelle/KV-Diagramm)

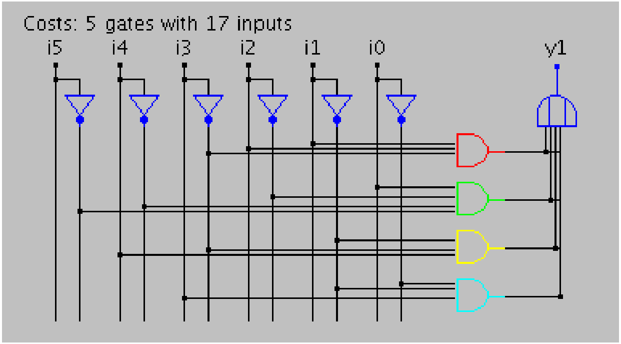
- solche Terme bei Minimierung nach Wunsch auf 0/1 setzen
- Schleifen dürfen Don't Cares enthalten
- Schleifen möglichst groß

KV-Diagramm Applet: Sechs Variablen, Don't Cares





KV-Diagramm Applet: Sechs Variablen, Don't Cares



- Schaltung und Realisierungsaufwand (Gattereingänge) zur vorigen Folie, nach der Minimierung

Quine-McCluskey-Algorithmus

- Algorithmus zur Minimierung einer Schaltfunktion
- Notation der Terme in Tabellen, n Variablen
- Prinzip entspricht der Minimierung im KV-Diagramm
- Grundlage gängiger Minimierungsprogramme („espresso“)

- Sortieren der Terme nach Hamming-Distanz
- Erkennen der unverzichtbaren Terme („Primimplikanten“)
- Aufstellen von Gruppen benachbarter Terme (mit Distanz 1)
- Zusammenfassen geeigneter benachbarter Terme

- Details: (Schiffmann & Schmitz) (Becker, Drechsler, Molitor)

Schaltnetze: Definition

- **Schaltnetz** oder
- **kombinatorische Schaltung** (*combinational logic circuit*):

  ein digitales System mit n-Eingängen $(b_1, b_2, \ldots, b_n)$ und m-Ausgängen $(y_1, y_2, \ldots, y_m)$, dessen Ausgangsvariablen zu jedem Zeitpunkt nur von den aktuellen Zuständen der Eingangsvariablen abhängen.

  Beschreibung als Vektorfunktion $\vec{y} = F(\vec{b})$

- Hinweis: ein Schaltnetz darf keine Rückkopplungen enthalten
- in der Praxis können Schaltnetze nicht rein statisch betrachtet werden: Gatterlaufzeiten spielen eine Rolle

Elementare digitale Schaltungen

- Schaltsymbole
- Grundgatter (Inverter, AND, OR, usw.)
- Kombinationen aus mehreren Gattern

- Schaltnetze (mehrere Ausgänge)
- Beispiele

- Arithmetisch/Logische Operationen

Schaltpläne (*schematics*)

- standardisierte Methode zur Darstellung von Schaltungen
- genormte Symbole für Komponenten:
  - Spannungs- und Stromquellen, Messgeräte
  - Schalter und Relais
  - Widerstände, Kondensatoren, Spulen
  - Dioden, Transistoren (bipolar, MOS)

  - **Gatter**: logische Grundoperationen (UND, ODER, usw.)
  - **Flipflops**: Speicherglieder
- Linien für Drähte (Verbindungen)
- Lötpunkte für Drahtverbindungen
- dicke Linien für *n*-bit Busse, Anzapfungen, usw.

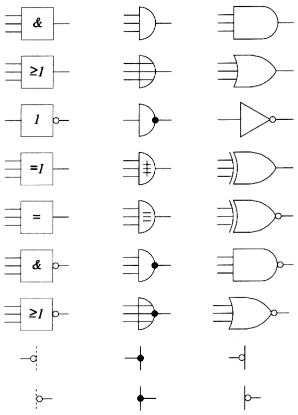
- komplexe Bausteine ggf. hierarchisch

Schaltsymbole

Schaltzeichen

DIN 40700 (ab 1976) | Früher | in USA

Benennung



- UND - Glied (AND)
- ODER - Glied (OR)
- NICHT - Glied (NOT)
- Exklusiv-Oder - Glied (Exclusive-OR, XOR)
- Äquivalenz - Glied (Logic identity)
- UND - Glied mit negiertem Ausgang (NAND)
- ODER - Glied mit negiertem Ausgang (NOR)
- Negation eines Eingangs
- Negation eines Ausgangs

(Schiffmann & Schmitz, Technische Informatik 1)

Logische Gatter

- **Logisches Gatter** (*logic gate*): die Bezeichnung für die Realisierung einer logischen Grundfunktion als gekapselte Komponente (in einer gegebenen Technologie)

- 1 Eingang: Treiberstufe/Verstärker und Inverter (Negation)
- 2 Eingänge: AND/OR, NAND/NOR, XOR, XNOR
- 3 - und mehr Eingänge: AND/OR, NAND/NOR, Parität
- Multiplexer

- mindestens Gatter für eine vollständige Basismenge erforderlich
- in Halbleitertechnologie sind NAND/NOR besonders effizient

Schaltplan-Editor und -Simulator

Spielerischer Zugang zu digitalen Schaltungen:

- mit Experimentierkasten oder im Logiksimulator
- interaktive Simulation erlaubt direktes Ausprobieren
- Animation und Visualisierung der logischen Werte
- „entdeckendes Lernen“

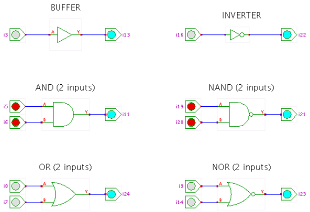
- Diglog: `www.eecs.berkeley.edu/~lazzaro/chipmunk/`
- Hades:
  `tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/`
- Kapitel: Grundschaltungen, Gate-Level Circuits
- Demos laufen im Browser (Java erforderlich)

Hades: Grundkomponenten

- Vorführung des Simulators

- Eingang und Schalter („*Ipin*“)
- Ausgang und Leuchtdiode („*Opin*“)
- Taktgenerator
- PowerOnReset
- Leuchtdiode
- Siebensegmentanzeige

...



Hades: *glow-mode* Visualisierung

- Farbe einer Leitung codiert den logischen Wert
- Einstellungen sind vom Benutzer konfigurierbar

- Defaultwerte:

blau	glow-mode ausgeschaltet
hellgrau	logisch-0
rot	logisch-1
orange	tri-state-Z (keine Treiber)
magenta	undefined-X (Kurzschluss)
cyan	unknown-U (nicht initialisiert)

Hades: Bedienung

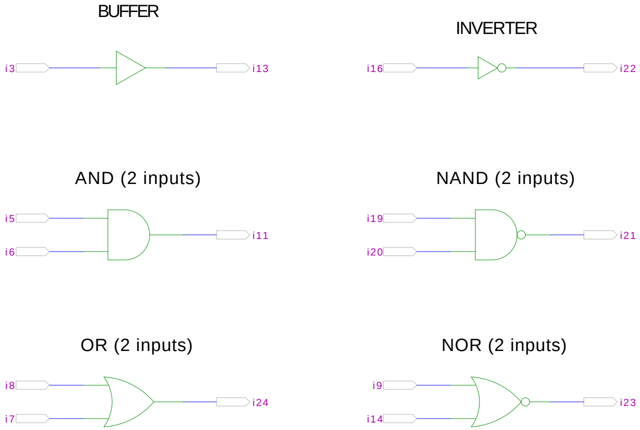
- Menü: Anzeigeoptionen, Edit-Befehlen, usw.

- Editorfenster mit Popup-Menü für häufige Aktionen
- Rechtsklick auf Komponenten öffnet *property-sheets*
- optional „tooltips“ (enable im Layer-Menü)

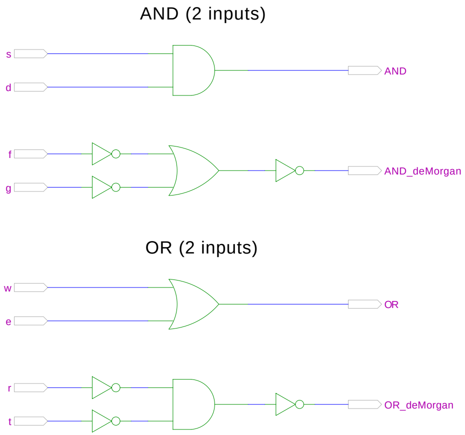
- Simulationssteuerung: *run*, *pause*, *rewind*
- Anzeige der aktuellen Simulationszeit

- Details siehe Hades-Webseite: Kurzreferenz, Tutorial

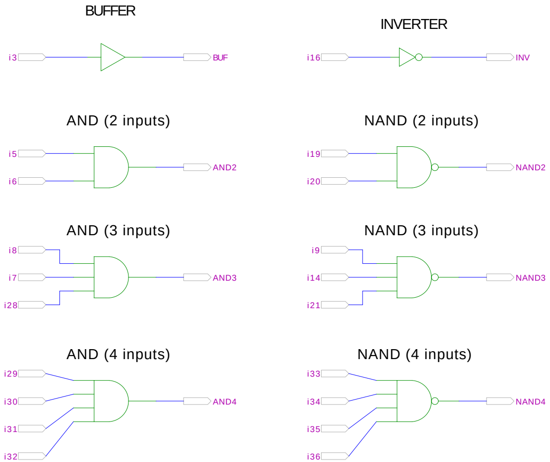
Gatter: Verstärker, Inverter, AND, OR



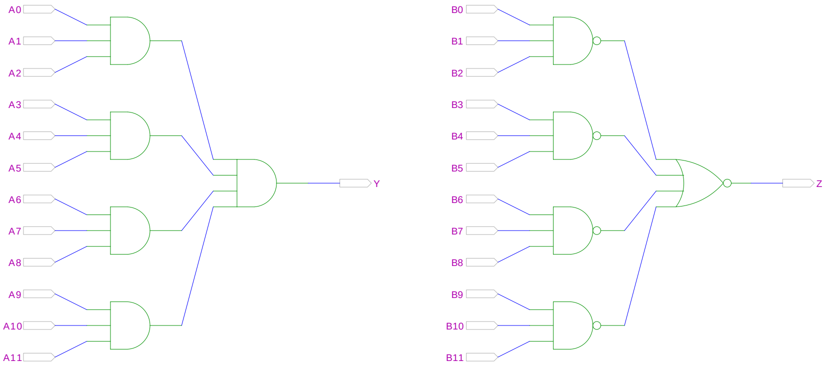
Grundschaltungen: De'Morgan Regel



Gatter: AND/NAND mit zwei, drei, vier Eingängen

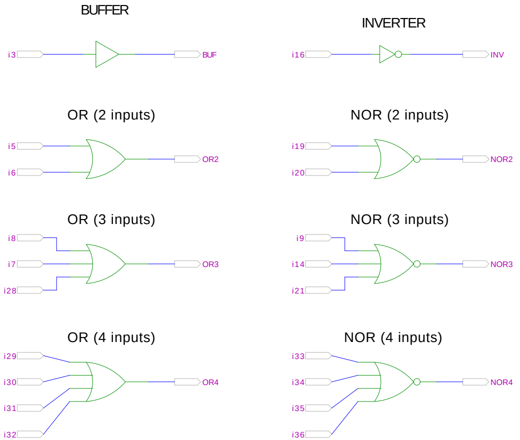


Gatter: AND mit zwölf Eingängen



links: AND3-AND4 rechts: NAND3-NOR4 (de-Morgan)

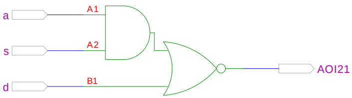
Gatter: OR/NOR mit zwei, drei, vier Eingängen



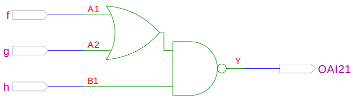
Komplexgatter

in CMOS-Technologie besonders günstig realisierbar

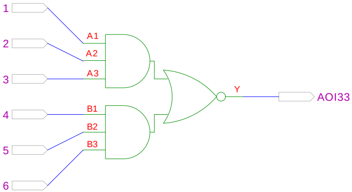
AOI21 (And-Or-Invert)



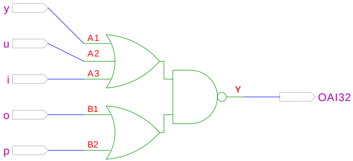
OAI21 (Or-And-Invert)



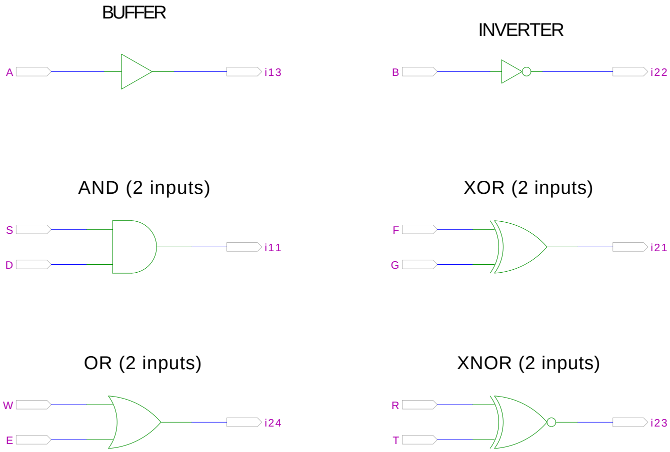
AOI33 (And-Or-Invert)



OAI32 (Or-And-Invert)

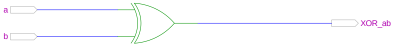


Gatter: XOR und XNOR

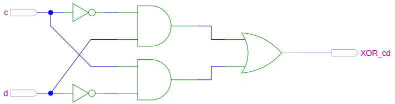


XOR und drei Varianten der Realisierung

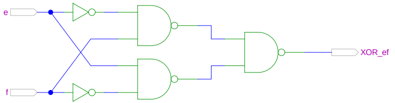
- Symbol



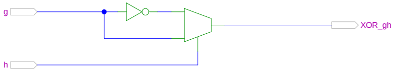
- AND-OR



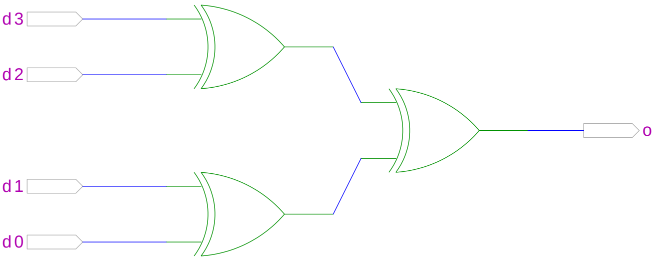
- NAND-NAND



- mit Multiplexer

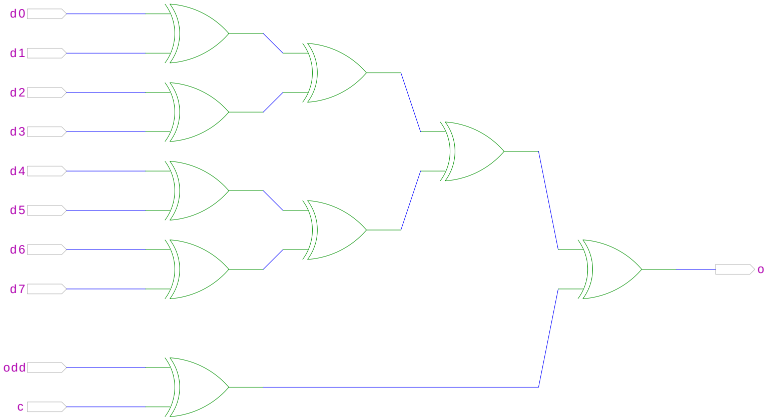


4-bit Parität mit XOR



8-bit Parität mit XOR

bzw. 10-bit: Umschaltung odd/even, Kaskadierung über c-Eingang



2:1-Multiplexer

Umschalter zwischen zwei Dateneingängen („Wechselschalter“)

- ein Steuereingang s
- zwei Dateneingänge a_1 und a_0, Datenausgang y
- wenn $s = 1$ wird a_1 zum Ausgang y durchgeschaltet,
- wenn $s = 0$ wird a_0 zum Ausgang y durchgeschaltet

s	a_1	a_0	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

2:1-Multiplexer

- kompaktere Darstellung der Funktionstabelle durch Verwendung von $*$ (don't care) Termen

s	a_1	a_0	y
0	*	0	0
0	*	1	1
1	0	*	0
1	1	*	1

s	a_1	a_0	y
0	*	a_0	a_0
1	a_1	*	a_1

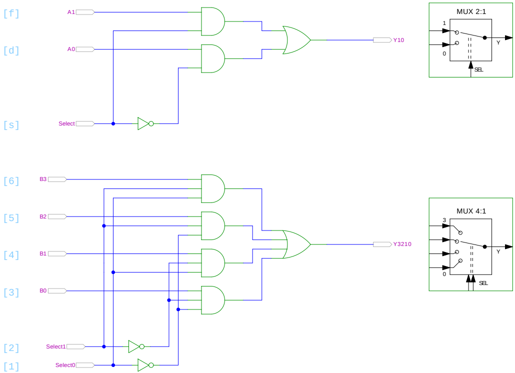
n:1-Multiplexer

Umschalten zwischen mehreren Dateneingängen

- Datenausgang y
- n Dateneingänge, $a_{n-1}, \ldots, a_1, a_0$
- $\lceil \log_2(n) \rceil$ Steuereingänge $s_m, \ldots, s_0$

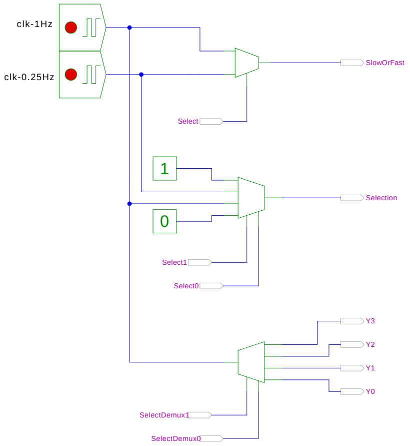
s_1	s_0	a_3	a_2	a_1	a_0	y
0	0	*	*	*	0	0
0	0	*	*	*	1	1
0	1	*	*	0	*	0
0	1	*	*	1	*	1
1	0	*	0	*	*	0
1	0	*	1	*	*	1
1	1	0	*	*	*	0
1	1	1	*	*	*	1

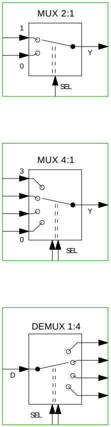
2:1 und 4:1 Multiplexer



- Hinweis: die Anordnung der Dateneingänge ist in Schaltplänen nicht einheitlich: höchstwertigster Eingang manchmal oben, manchmal unten.

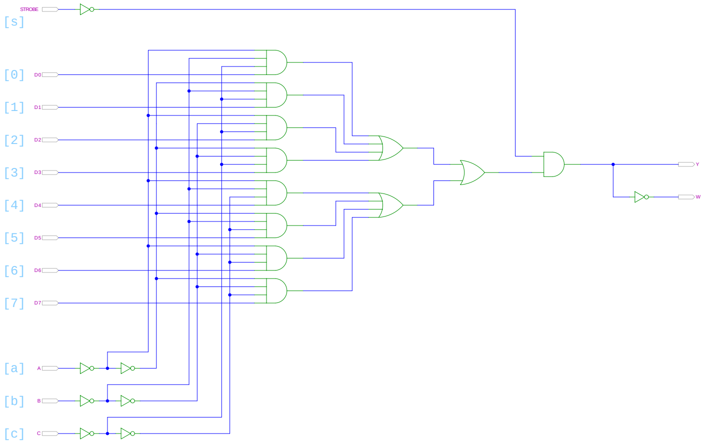
Multiplexer und Demultiplexer





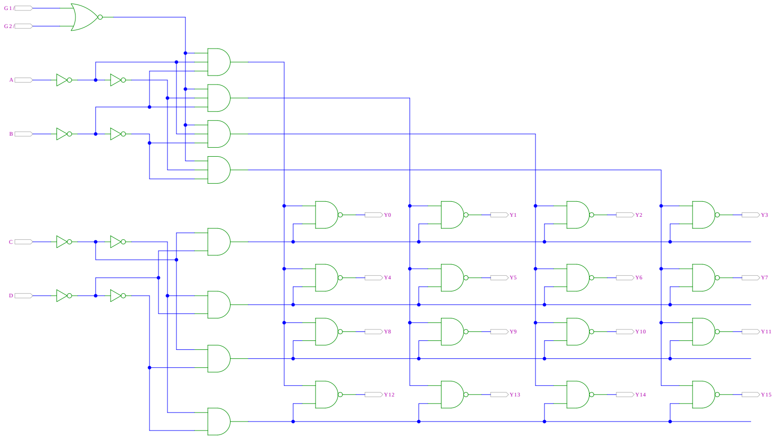
8-bit Multiplexer

Steuereingänge (a, b, c) aktivieren einen der acht Dateneingänge ($d_0, \dots d_7$)

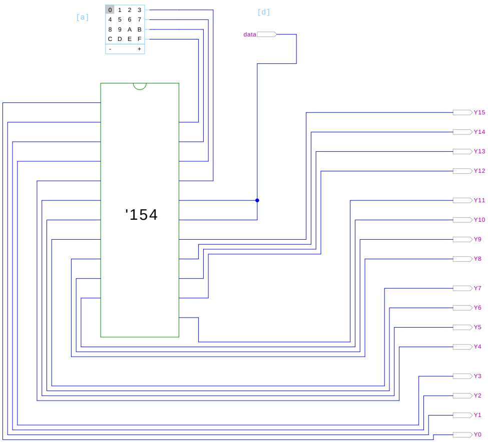


16-bit Demultiplexer: Integrierte Schaltung SN 74154

Dateneingang g wird auf einen der 16 Datenausgänge $y_0, \ldots y_{15}$ ausgegeben



16-bit Demultiplexer: SN 74154 als Adressdecoder



Beispiele für Schaltnetze

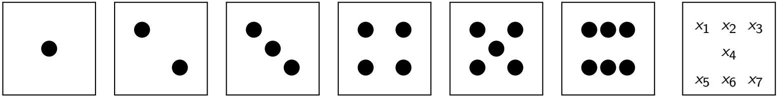
- Schaltungen mit mehreren Ausgängen
- Bündelminimierung der einzelnen Funktionen

ausgewählte typische Beispiele:

- Würfel-Decoder
- Umwandlung vom Dual-Code in den Gray-Code
- (7,4)-Hamming-Code: Encoder und Decoder
- Siebensegmentanzeige

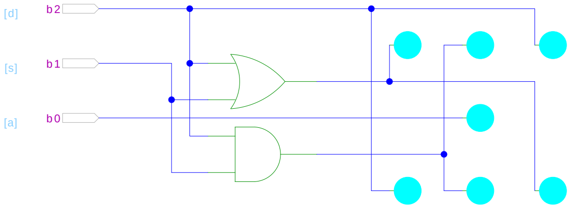
Beispiel: „Würfel"-Decoder

Visualisierung eines Würfels mit sieben LEDs



Wert	b_2	b_1	b_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	1	0	0	0
2	0	1	0	1	0	0	0	0	0	1
3	0	1	1	1	0	0	1	0	0	1
4	1	0	0	1	0	1	0	1	0	1
5	1	0	1	1	0	1	1	1	0	1
6	1	1	0	1	1	1	0	1	1	1

Beispiel: „Würfel“-Decoder

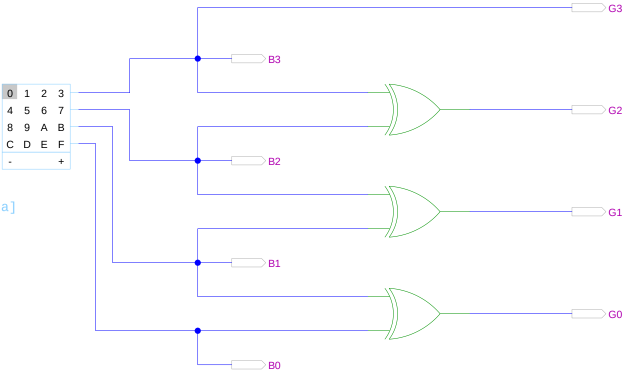


- Anzeige wie beim Würfel: ein bis sechs Augen
- Minimierung ergibt:

$x_1 = x_7 = b_2 \vee b_1$	(links oben, rechts unten)
$x_2 = x_6 = b_0 \wedge b_1$	(oben und unten Mitte)
$x_3 = x_5 = b_2$	(rechts oben, links unten)
$x_4 = b_0$	(Zentrum)

Umwandlung vom Dualcode in den Graycode

XOR benachbarter Bits



[a]

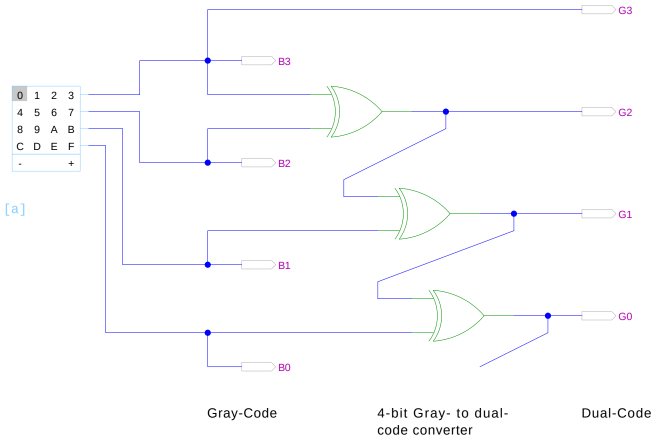
binary code

4-bit dual to Gray code converter

Gray code

Umwandlung vom Graycode in den Dualcode

XOR-Kette



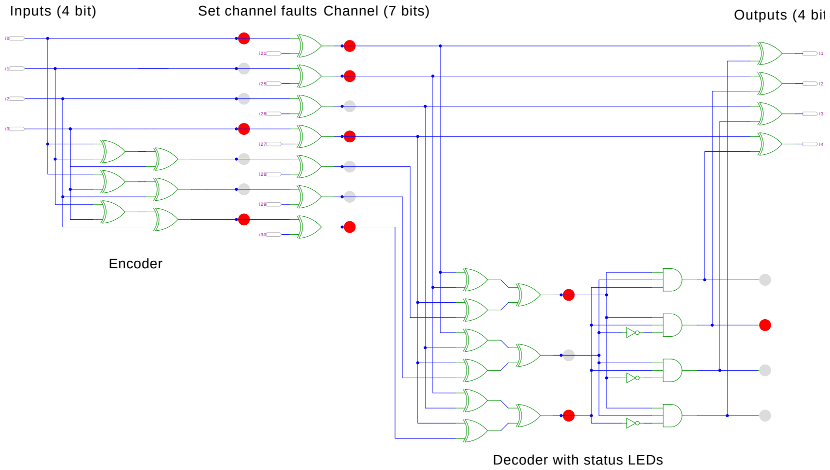
(7,4)-Hamming-Code: Encoder und Decoder

- vier Eingabebits (ganz links)
- Hamming-Encoder erzeugt drei Paritätsbits (links unten)

- Übertragung von sieben Codebits (Mitte)
- Einfügen von Übertragungsfehlern durch
  Invertieren von Codebits mit XOR-Gattern

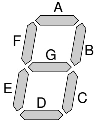
- Decoder liest die empfangenen sieben Bits (rechts unten)
  Syndrom-Berechnung mit XOR-Gattern
  Anzeige erkannter Fehler
- Korrektur gekippter Bits (rechts oben)

(7,4)-Hamming-Code: Encoder und Decoder



Siebensegmentanzeige

- sieben einzelne Leuchtsegmente (z.B. Leuchtdioden)
- Anzeige stilisierter Ziffern von `0` bis `9`
- auch für Hex-Ziffern: `A`, `b`, `C`, `d` `E`, `F`



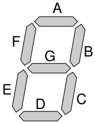
- sieben Schaltfunktionen, je eine pro Ausgang
- Umcodierung von 4-bit Dualwerten in geeignete Ausgangswerte

- eingeschränkt auch als alphanumerische Anzeige für Ziffern und Buchstaben (gemischt Groß- und Kleinbuchstaben. Natürlich Probleme mit `M`, `N`, usw.)

Siebensegmentanzeige: Funktionen

- Funktionen für Hex-Anzeige, `0` .. `F`

$a = 1011\ 0111\ 1110\ 0011$
$b = 1111\ 1001\ 1110\ 0100$
$c = 1101\ 1111\ 1111\ 0100$
$d = 1011\ 0110\ 1101\ 1110$
$e = 1010\ 0010\ 1011\ 1111$
$f = 1000\ 1111\ 1111\ 0011$
$g = 0011\ 1110\ 1111\ 1111$



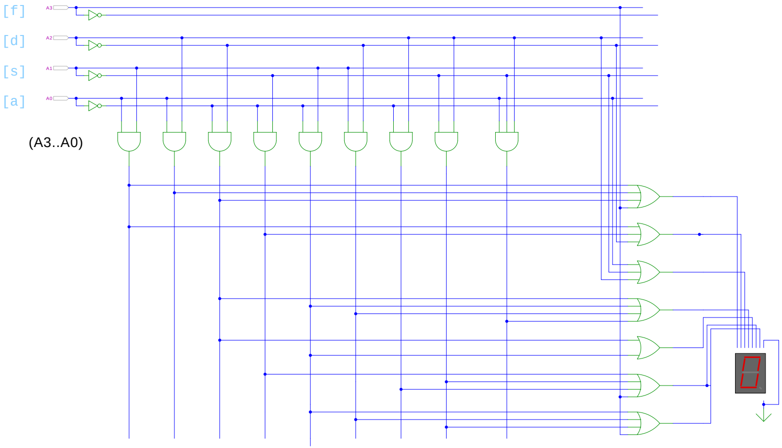
- für Ziffernanzeige mit *Don't Care*-Termen:
  $a = 1011\ 0111\ 11{*}{*}\ \ {*}{*}{*}{*}$, usw.

Siebensegmentanzeige: Bündelminimierung

- zum Beispiel mit sieben KV-Diagrammen…
- dabei versuchen, gemeinsame Terme zu finden und zu nutzen
- Minimierung als Übungsaufgabe?
- nächste Folie zeigt Lösung aus Schiffmann

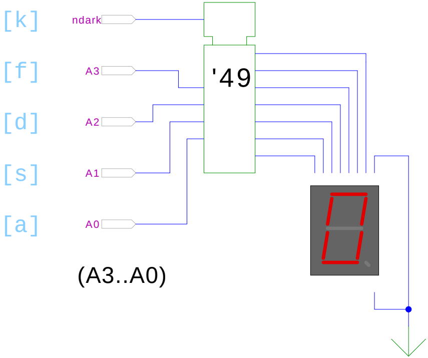
- als mehrstufige Schaltung ist günstigere Lösung möglich
- siehe Knuth: *AoCP, Volume 4.0*, 7.1.2 (p.112ff)

Siebensegmentdecoder: Ziffern 0..9



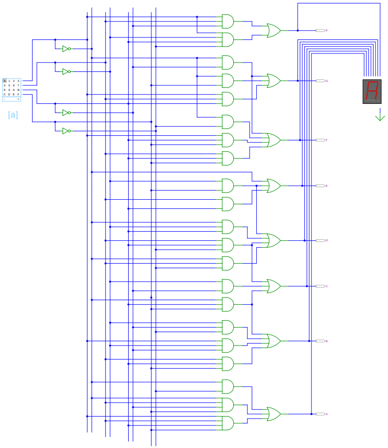
(Schiffmann & Schmitz, Technische Informatik 1)

Siebensegmentdecoder: SN 7449

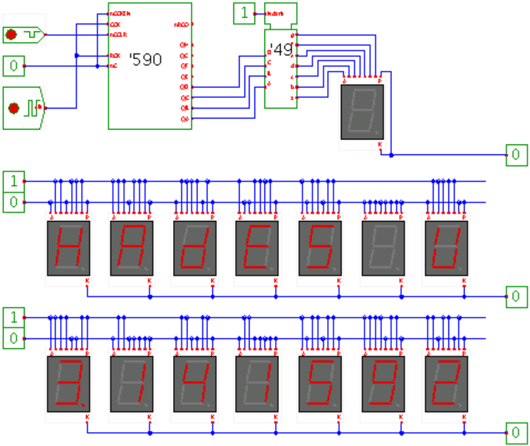


- Beispiel für eine integrierte Schaltung (IC)
- Anzeige von 0..9, Zufallsmuster für A..F, „Dunkeltastung“

Siebensegmentdecoder: Buchstaben A..P



Siebensegmentanzeige: Hades-Beispiel



Siebensegmentanzeige: mehrstufige Realisierung

- minimale Anzahl der Gatter für die Schaltung?!
- Problem vermutlich nicht optimal lösbar (nicht *tractable*)
- Heuristik basierend auf „häufig" verwendeten Teilfunktionen
- Eingänge x_1, x_2, x_3, x_4, Ausgänge $a, \ldots, g$

$$
\begin{array}{lll}
x_5 = x_2 \oplus x_3, & x_{13} = x_1 \oplus x_7, & \overline{a} = x_{20} = x_{14} \wedge \overline{x_{19}}, \\
x_6 = \overline{x_1} \wedge x_4, & x_{14} = x_5 \oplus x_6, & \overline{b} = x_{21} = x_7 \oplus x_{12}, \\
x_7 = x_3 \wedge \overline{x_6}, & x_{15} = x_7 \vee x_{12}, & \overline{c} = x_{22} = \overline{x_8} \wedge x_{15}, \\
x_8 = x_1 \oplus x_2, & x_{16} = x_1 \vee x_5, & \overline{d} = x_{23} = x_9 \wedge \overline{x_{13}}, \\
x_9 = x_4 \oplus x_5, & x_{17} = x_5 \vee x_6, & \overline{e} = x_{24} = x_6 \vee x_{18}, \\
x_{10} = \overline{x_7} \wedge x_8, & x_{18} = x_9 \wedge x_{10}, & \overline{f} = x_{25} = \overline{x_8} \wedge x_{17}, \\
x_{11} = x_9 \oplus x_{10}, & x_{19} = x_3 \wedge x_9, & g = x_{26} = x_7 \vee x_{16}, \\
x_{12} = x_5 \wedge x_{11} & &
\end{array}
$$

(Knuth, AoCP 4.0, 7.1.2, 2008)

Logische und arithmetische Operationen

- Halb- und Volladdierer
- Carry-Ripple-Addierer
- Carry-Lookahead-Addierer

- Multiplizierer
- Quadratwurzel

- Barrel-Shifter
- ALU

Halbaddierer

- **Halbaddierer**: berechnet 1-bit Summe s und Übertrag c_o (*carry-out*) von zwei Eingangsbits a und b

a	b	c_o	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$c_o = a \wedge b$$

$$s = a \oplus b$$

Volladdierer

- **Volladdierer**: berechnet 1-bit Summe s und Übertrag c_o von zwei Eingangsbits a und b und Carry-in c_i

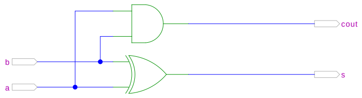
a	b	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$c_o = ab \vee ac_i \vee bc_i = (ab) \vee (a \vee b)c_i$$

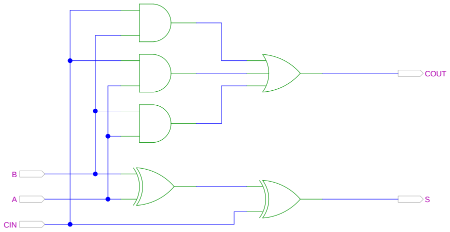
$$s = a \oplus b \oplus c_i$$

Schaltbilder Halb- und Volladdierer

1-bit half-adder: (COUT,S) = (A+B)



1-bit full-adder: (COUT,S) = (A+B+Cin)



n-bit Addierer

$$s_0 = a_0 \oplus b_0$$
$$s_1 = a_1 \oplus b_1 \oplus c_1$$
$$s_2 = a_2 \oplus b_2 \oplus c_2$$
$$\ldots$$

- $s_n \quad = a_n \oplus b_n \oplus c_n$

$$c_1 = (a_0 b_0)$$
$$c_2 = (a_1 b_1) \vee (a_1 \vee b_1) c_1$$
$$c_3 = (a_2 b_2) \vee (a_2 \vee b_2) c_2$$

- $c_{n+1} = (a_n b_n) \vee (a_n \vee b_n) c_n$

n-bit Addierer

- n-bit Addierer theoretisch als zweistufige Schaltung realisierbar
- direkte und negierte Eingänge, dann AND-OR Netzwerk
- Aufwand steigt aber sehr schnell mit n an
- für Ausgang n sind $2^{(2n-1)}$ Minterme erforderlich
- nicht praktikabel

Diverse gängige Alternativen:

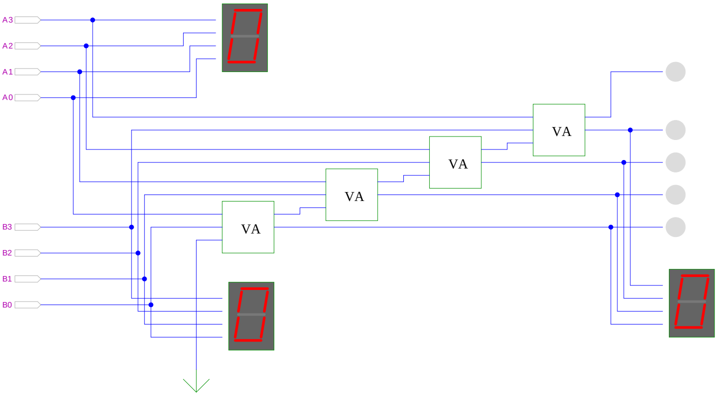
- Ripple-Carry Adder (mehrstufig, billig, langsam $O(n)$))
- Carry-Lookahead-Adder (Baumstruktur, teuer, schnell)
- Zwischenformen

Ripple-Carry Adder

- Kaskade aus n einzelnen Volladdierern
- Carry-out von Stufe i treibt Carry-in von Stufe $i+1$
- Gesamtverzögerung wächst mit der Anzahl der Stufen als $O(n)$

- Addierer in Prozessoren häufig im *kritischen Pfad*
- möglichst hohe Performance ist essentiell
- ripple-carry in CMOS-Technologie bis ca. 10-bit geeignet
- bei größerer Wortbreite gibt es effizientere Schaltungen

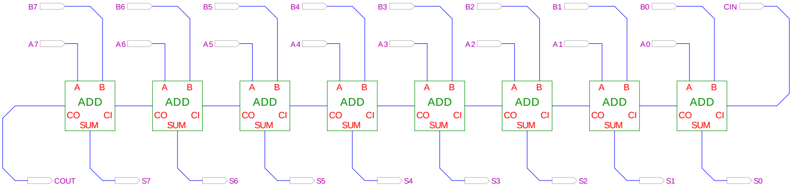
Ripple-Carry Adder: 4-bit



(Schiffmann & Schmitz, Technische Informatik Online)

Ripple-Carry Adder: Demo mit Verzögerungen

- Kaskade aus acht einzelnen Volladdierern



- Gatterlaufzeiten in der Simulation bewusst groß gewählt
- Ablauf der Berechnung kann interaktiv beobachtet werden
- alle Addierer arbeiten parallel
- aber Summe erst fertig, wenn alle Stufen durchlaufen sind

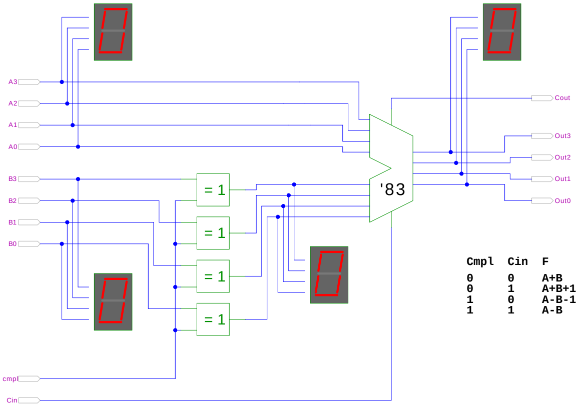
Subtrahierer: Zweierkomplement

- $(A - B)$ ersetzt durch Addition des 2-Komplements von B
- 2-Komplement: Invertieren aller Bits und Addition von Eins
- Carry-in Eingang des Addierers bisher nicht benutzt

Subtraktion quasi „gratis“ realisierbar:

- normalen Addierer verwenden
- Invertieren der Bits von B (1-Komplement)
- Carry-in Eingang auf 1 setzen (Addition von 1)
- Resultat ist $A + (\neg B) + 1 = A - B$

Subtrahierer



Cmpl	Cin	F	
0	0	A+B	
0	1	A+B+1	(A++)
1	0	A-B-1	(A--)
1	1	A-B	

(7483: 4-bit Addierer) (Schiffmann & Schmitz, Technische Informatik Online)

Schnelle Addierer

- Addierer in Prozessoren häufig im *kritischen Pfad*
- möglichst hohe Performance ist essentiell

- Carry-Select Adder: Gruppen von ripple-carry
- Carry-Lookahead Adder: Baumstruktur
- technologieabhängige Kombinationen

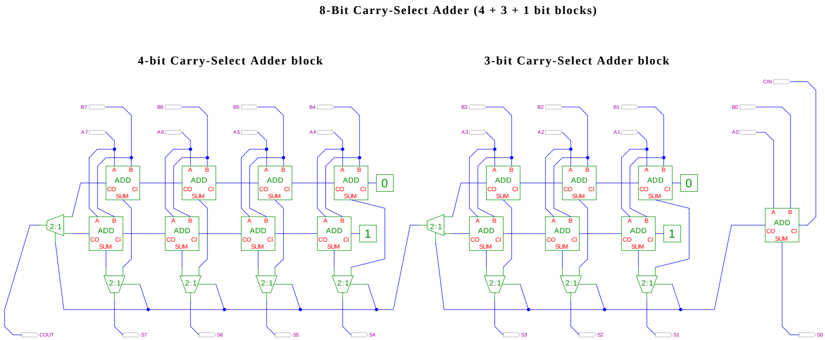
Carry-Select Adder: Prinzip

- Aufteilen des n-bit Addierers in mehrere Gruppen mit je m_i-bits
- für jede Gruppe:
  - jeweils zwei m_i-bit Addierer
  - einer rechnet mit $c_i = 0$, der andere mit $c_i = 1$
  - 2:1-Multiplexer mit m_i-bit wählt die korrekte Summe aus

- Sobald der Wert von c_i bekannt ist, wird über den Multiplexer die benötigte Zwischensumme ausgewählt, und das berechnete Carry-out c_o der Gruppe als Carry-in c_i der folgenden Gruppe verwendet

- Verzögerung reduziert sich auf die Verzögerung eines m-bit Addierers plus die Verzögerungen der Multiplexer

Carry-Select Adder: Demo



- drei Gruppen: 1-bit, 3-bit, 4-bit
- Gruppengrößen so wählen, dass Gesamtverzögerung minimal

Carry-Lookahead Addierer: Prinzip

- $c_{n+1} = (a_n b_n) \vee (a_n \vee b_n) c_n$

- Einführung von Hilfsfunktionen

  $g_n = (a_n b_n)$ („generate carry“)

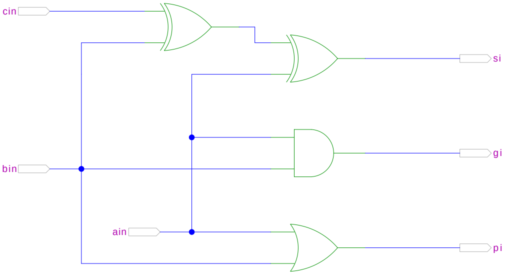
  $p_n = (a_n \vee b_n)$ („propagate carry“)

  $c_{n+1} = g_n + p_n c_n$

  Berechnung der g_n und p_n in einer Baumstruktur

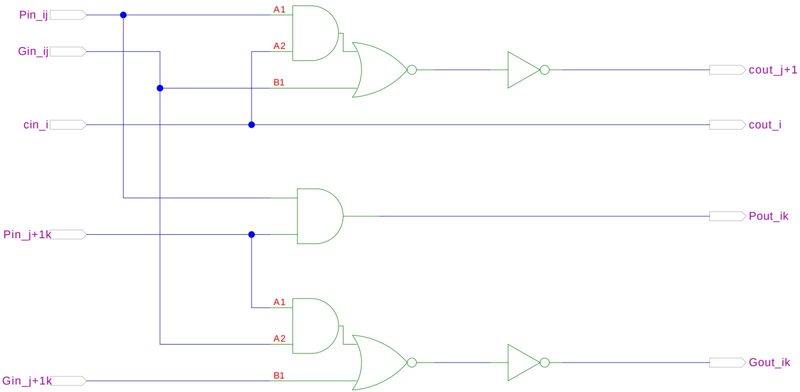
  Tiefe des Baums ist $\log_2 N$, entsprechend schnell

Carry-Lookahead Adder: SUM-Funktionsblock

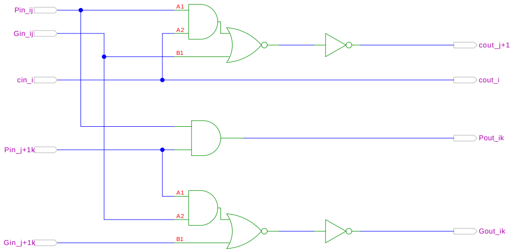


- 1-bit Addierer, $s = a_i \oplus b_i \oplus c_i$
- keine Berechnung des Carry-Out
- Ausgang $g_i = a_i \wedge b_i$ liefert *generate-carry* Signal
- Ausgang $p_i = a_i \vee b_i$ liefert *propagate-carry* Signal

Carry-Lookahead Adder: CLA-Funktionsblock



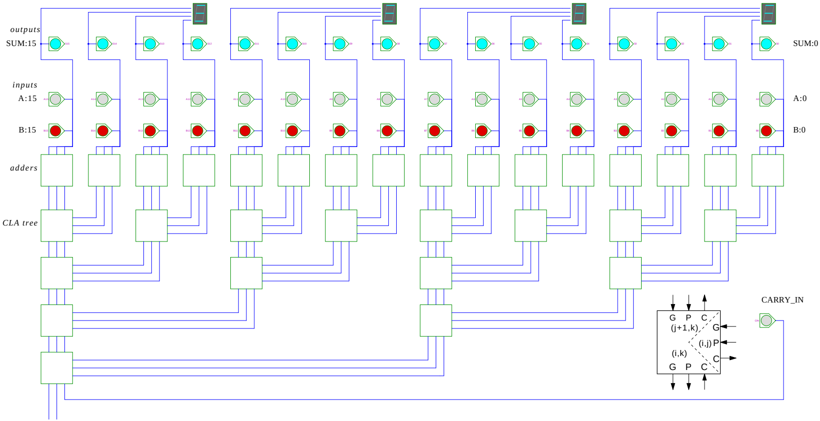
Carry-Lookahead Adder: CLA-Funktionsblock



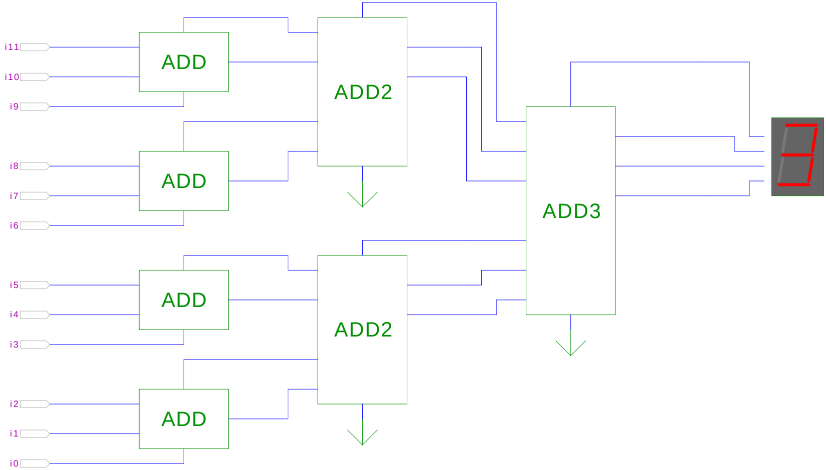
- Eingänge: propagate/generate Signale von zwei Stufen
- Eingänge: carry-in Signal

- Ausgang: propagate/generate Signale zur nächsthöheren Stufe
- Ausgänge: carry-out Signale zur nächsthöheren Stufe

Carry-Lookahead Adder: 16-bit Addierer



Bitcount: Addierer-Baum



Addierer: Zusammenfassung

- Halbaddierer ($a \oplus b$)
- Volladdierer ($a \oplus b \oplus c_i$)

- ripple-carry: Kaskade aus Volladdierern
- einfach und billig, aber manchmal zu langsam: $O(n)$

- carry-select Prinzip
- carry-lookahead Prinzip: Verzögerung $O(\ln n)$

- Subtraktion durch Zweierkomplementbildung
- erlaubt auch Inkrement (`A++`) und Dekrement (`A--`)

Multiplizierer

- Teilprodukte als UND-Verknüfung des Multiplikators mit je einem Bit des Multiplikanden
- Aufaddiereren der Teilprodukte mit Addierern
- Realisierung als Schaltnetz erfordert:

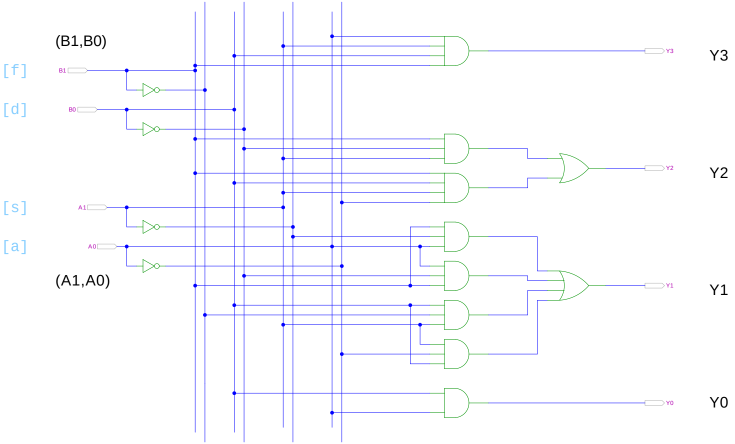
  n^2 UND-Gatter (bitweise eigentliche Multiplikation)

  n^2 Volladdierer (Aufaddieren der Teilprodukte)
- abschließend ein n-bit Addierer für die Überträge
- in heutiger CMOS-Technologie kein Problem

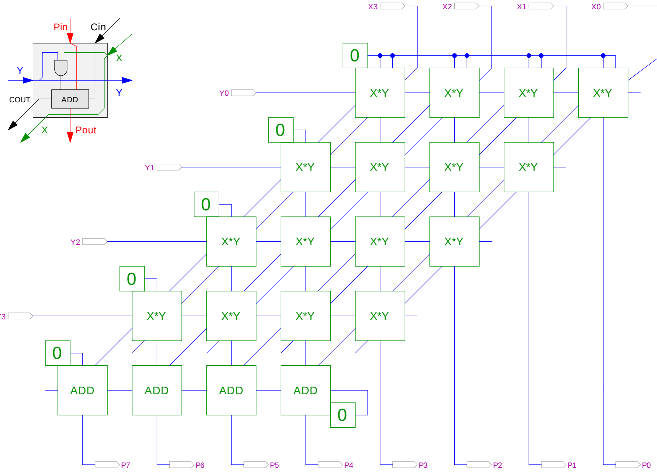
- alternativ: Schaltwerke mit sukzessiver Berechnung des Produkts in mehreren Takten

2x2-bit Multiplizierer

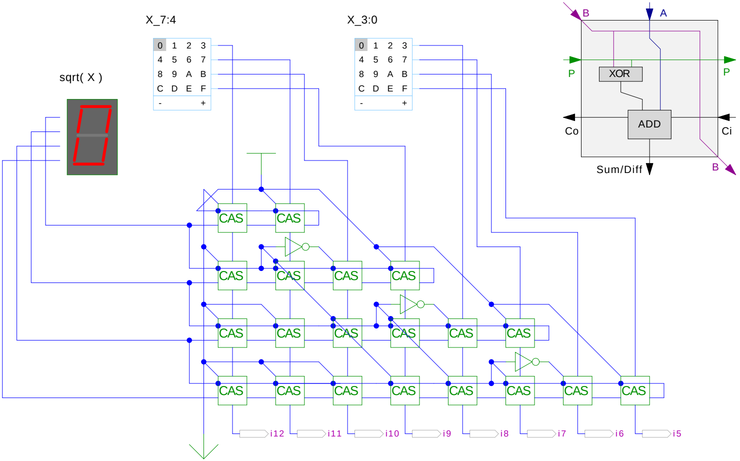
als zweistufiges Schaltnetz



4x4-bit Multiplizierer



4x4-bit Quadratwurzel



Multiplizierer

weitere wichtige Themen aus Zeitgründen nicht behandelt:

- *Booth*-Codierung
- *Carry-Save Adder* zur Summation der Teilprodukte
- Multiplikation von Zweierkomplementzahlen
- Multiplikation von Gleitkommazahlen

- CORDIC-Algorithmen
- bei Interesse: Literatur anschauen

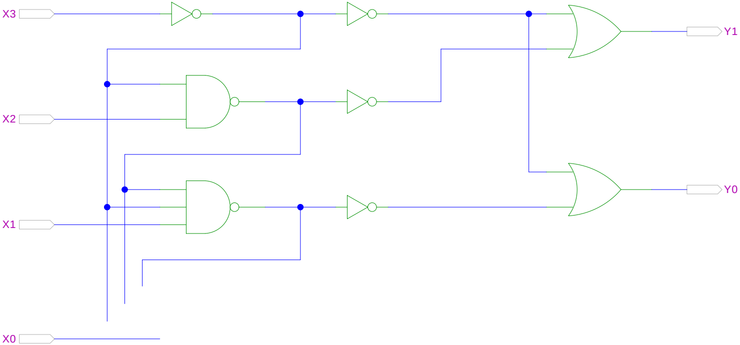
Priority Encoder

- Anwendung u.a. für Interrupt-Priorisierung
- Schaltung konvertiert *n*-bit Eingabe in eine Dualcodierung
- Wenn Bit *n* aktiv ist, werden alle niedrigeren Bits $(n-1),\ldots,0$ ignoriert:

x_3	x_2	x_1	x_0	y_1	y_0
1	*	*	*	1	1
0	1	*	*	1	0
0	0	1	*	0	1
0	0	0	*	0	0

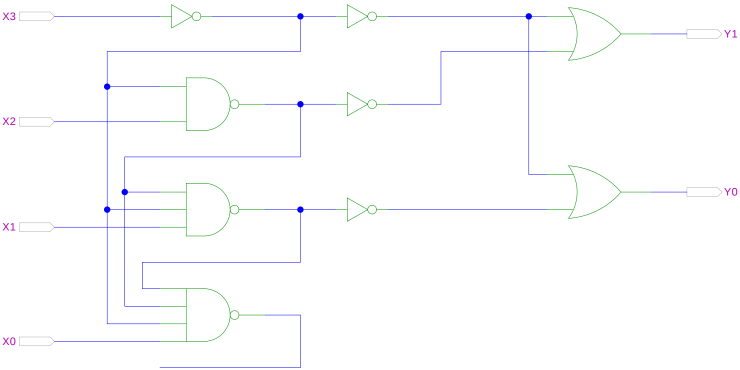
(unabhängig von niederwertigstem Bit...)

4:2 Prioritätsencoder

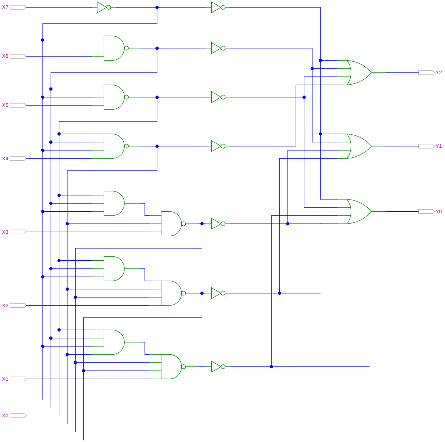


- zweistufige Realisierung
- aktive höhere Stufe blockiert alle niedrigeren Stufen

4:2 Prioritätsencoder: Kaskadierung



8:3 Prioritätsencoder

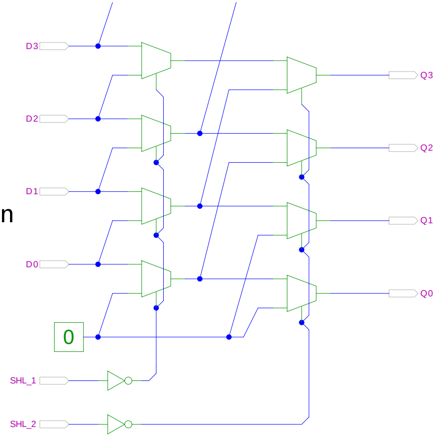


Shifter: zweistufig, shift-left um 0..3 Bits

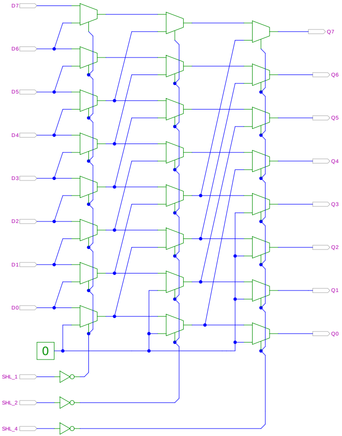
- *n*-Dateneingänge
- *n*-Datenausgänge

- Kaskade von 2:1 Multiplexern
  Stufe 0: benachbarte Bits
  Stufe 1: übernächste Bits
  usw.

  von rechts 0 nachschieben



8-bit „Barrel-Shifter“



Shift-Right, Rotate & Co.

- Prinzip der oben vorgestellten Schaltungen gilt auch für die übrigen Shift- und Rotate-Operationen

- logic-shift right: von links Nullen nachschieben
- arithmetic shift right: oberstes Bit nachschieben

- rotate left und right: außen herausgeschobene Bits auf der anderen Seite wieder hereinschieben

- alle Operationen typischerweise in einem Takt realisierbar

Arithmetisch-Logische Einheit (ALU)

- **Arithmetisch-logische Einheit** (engl. *arithmetic/logic unit*): kombiniertes Schaltwerk für arithmetische und logische Operationen
- das zentrale Rechenwerk in Prozessoren
- Funktionsumfang variiert von Typ zu Typ

- Addition und Subtraktion (2-Komplement)
- bitweise logische Operationen (Negation, UND, ODER, XOR)
- Schiebeoperationen (shift, rotate)
- evtl. Multiplikation

- Integer-Division selten verfügbar (separates Rechenwerk)

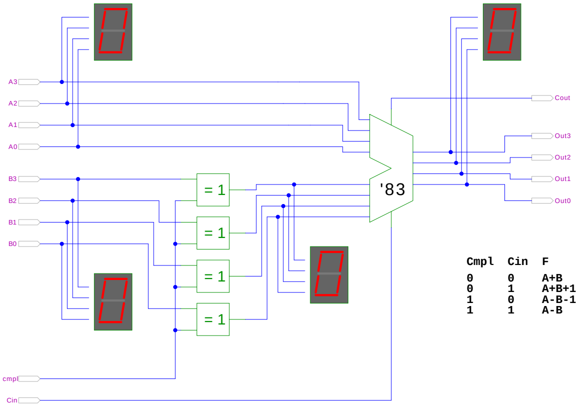
ALU: Addierer und Subtrahierer

- Addition $(A + B)$ mit normalem Addierer

- XOR-Gatter zum Invertieren von Operand B
- Steuerleitung *sub* aktiviert das Invertieren und den Carry-in c_i
- wenn aktiv, Subtraktion als $(A - B) = A + \neg B + 1$
- ggf. auch Increment $(A + 1)$ und Decrement $(A - 1)$

- folgende Folie: 7483 ist IC mit 4-bit Addierer

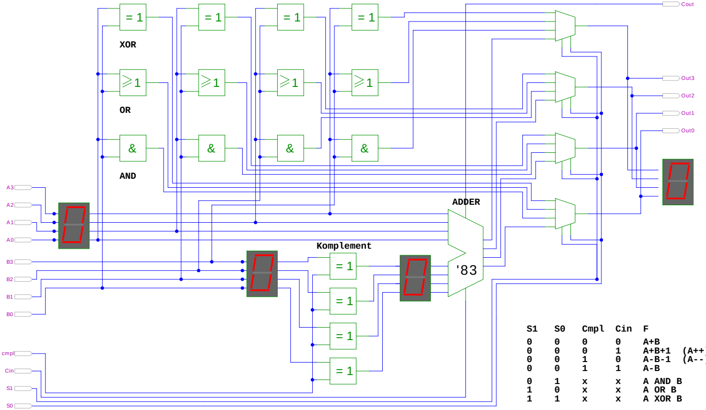
ALU: Addierer und Subtrahierer



Cmpl	Cin	F	
0	0	A+B	
0	1	A+B+1	(A++)
1	0	A-B-1	(A--)
1	1	A-B	

(Schiffmann & Schmitz, Technische Informatik Online)

ALU: Addierer und bitweise Operationen



S1	S0	Cmpl	Cin	F
0	0	0	0	A+B
0	0	0	1	A+B+1 (A++)
0	0	1	0	A-B-1 (A--)
0	0	1	1	A-B
0	1	x	x	A AND B
1	0	x	x	A OR B
1	1	x	x	A XOR B

(Schiffmann & Schmitz, Technische Informatik Online)

ALU: Prinzip

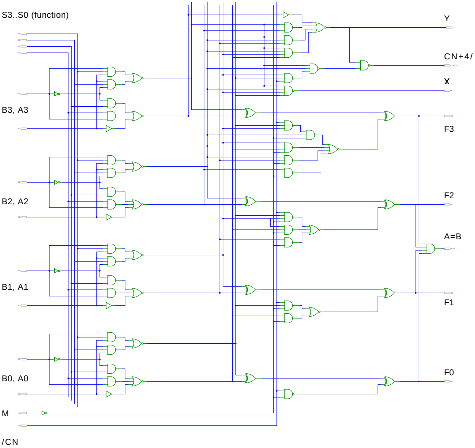
vorige Folie zeigt die „triviale“ Realisierung einer ALU:

- mehrere parallele Rechenwerke für die m einzelnen Operationen
  n-bit Addierer, n-bit Komplement, n-bit OR, usw.
- Auswahl des Resultats über n-bit $m:1$-Multiplexer

nächste Folie: Realisierung in der Praxis (74181 IC):

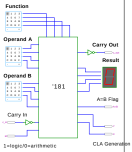
- erste Stufe für bitweise logische Operationen und Komplement
- zweite Stufe als Carry-lookahead Addierer
- weniger Gatter und schneller

ALU: 74181



selection S3 S2 S1 S0	logic functions M = H	arithmetic functions M = L, Cn=H (no carry)
L L L L	F = !A	F = A
L L L H	F = !(A or B)	F = A or B
L L H L	F = !A * B	F = A or !B
L L H H	F = !A * B	F = MINUS 1
L H L L	F = 0	F = A PLUS (A*!B)
L H L H	F = !B	F = (A or B) PLUS (A * !B)
L H H L	F = A xor B	F = A MINUS B MINUS 1
L H H H	F = A * !B	F = (A * !B) MINUS 1
H L L L	F = !A or B	F = A PLUS (A*B)
H L L H	F = A xnor B	F = A PLUS B
H L H L	F = B	F = (A or !B) PLUS (A*B)
H L H H	F = A * B	F = (A*B) MINUS 1
H H L L	F = 1	F = A PLUS A
H H L H	F = A or !B	F = (A or B) PLUS A
H H H L	F = A or B	F = (A or !B) PLUS A
H H H H	F = A	F = A MINUS 1
		Cn=L: PLUS 1

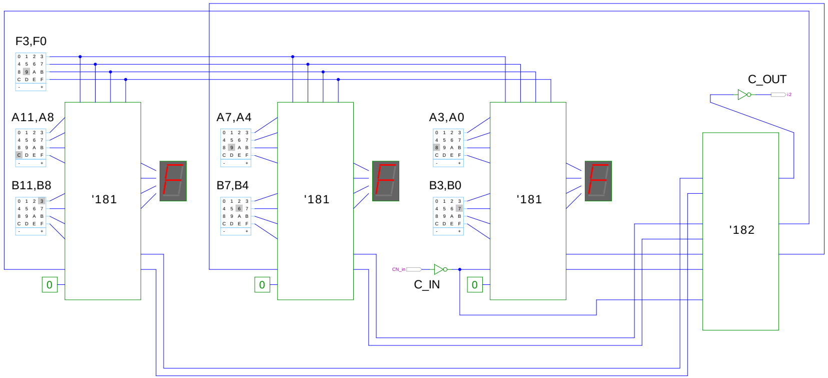
ALU: 74181 (Funktionstabelle und Demo)



selection S3 S2 S1 S0	logic functions M = H	arithmetic functions M = L, Cn=H (no carry)
L L L L	F = !A	F = A
L L L H	F = !(A or B)	F = A or B
L L H L	F = !A * B	F = A or !B
L L H H	F = 0	F = MINUS 1
L H L L	F = !(A*B)	F = A PLUS (A*!B)
L H L H	F = !B	F = (A or B) PLUS (A * !B)
L H H L	F = A xor B	F = A MINUS B MINUS 1
L H H H	F = A * !B	F = (A * !B) MINUS 1
H L L L	F = !A or B	F = A PLUS (A*B)
H L L H	F = A xnor B	F = A PLUS B
H L H L	F = B	F = (A or !B) PLUS (A*B)
H L H H	F = A * B	F = (A*B) MINUS 1
H H L L	F = 1	F = A PLUS A
H H L H	F = A or !B	F = (A or B) PLUS A
H H H L	F = A or B	F = (A or !B) PLUS A
H H H H	F = A	F = A MINUS 1
		Cn=L: PLUS 1

ALU: 74181 und 74182 CLA

12-bit ALU mit Carry-Lookahead Generator 74182



Literatur: Vertiefung

- Donald E. Knuth, *The Art of Computer Programming*:
  Volume 4 Fascicle 0: Boolean Functions
  Volume 4 Fascicle 1: Bitwise Tricks and Techniques, Binary Decision Diagrams Addison-Wesley, 2006-2009
- Ingo Wegener, *The Complexity of Boolean Functions*,
  ls2-www.cs.uni-dortmund.de/monographs/bluebook/
- B. Becker, R. Drechsler & P. Molitor,
  Technische Informatik — Eine Einführung,
  Pearson Studium, 2005
  Besonderheit: Einführung von BDDs/ROBDDs

Material: Interaktives Lehrmaterial

- Klaus von der Heide,
  Vorlesung Technische Informatik T1,(Matlab)
  Universität Hamburg, FB Informatik, 2004
- Norman Hendrich,
  Hamburg Design System,
  tams-www.informatik.uni-hamburg.de/applets/hades/
  KV-Diagram Simulation,
  tams-www.informatik.uni-hamburg.de/applets/kvd/
- John Lazarro,
  Chipmunk design tools (AnaLog, DigLog)
  www.eecs.berkeley.edu/~lazzaro/chipmunk/