

# 64-040 Modul IP7: Rechnerstrukturen

[http://tams.informatik.uni-hamburg.de/  
lectures/2011ws/vorlesung/rs](http://tams.informatik.uni-hamburg.de/lectures/2011ws/vorlesung/rs)

## Kapitel 12

Andreas Mäder



Universität Hamburg  
Fakultät für Mathematik, Informatik und Naturwissenschaften  
Fachbereich Informatik

**Technische Aspekte Multimodaler Systeme**

Wintersemester 2011/2012

# Kapitel 12

## Schaltnetze

Definition

Schaltsymbole und Schaltpläne

Hades: Editor und Simulator

Logische Gatter

Inverter, AND, OR

XOR und Parität

Multiplexer

Einfache Schaltnetze

Siebensegmentanzeige

Schaltnetze für Logische und Arithmetische Operationen

Addierer

Multiplizierer

Prioritätsencoder

# Kapitel 12 (cont.)

Barrel-Shifter  
ALU (Arithmetisch-Logische Einheit)  
Literatur



# Schaltnetze: Definition

- **Schaltnetz** oder auch **kombinatorische Schaltung** (*combinational logic circuit*): ein digitales System mit  $n$ -Eingängen  $(b_1, b_2, \dots, b_n)$  und  $m$ -Ausgängen  $(y_1, y_2, \dots, y_m)$ , dessen Ausgangsvariablen zu jedem Zeitpunkt nur von den aktuellen Werten der Eingangsvariablen abhängen

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

- Hinweis: ein Schaltnetz darf keine Rückkopplungen enthalten
- in der Praxis können Schaltnetze nicht 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
- ▶ Verbindungen
  - ▶ Linien für Drähte (Verbindungen)
  - ▶ Lötunkte für Drahtverbindungen
  - ▶ dicke Linien für  $n$ -bit Busse, Anzapfungen, usw.
- ▶ komplexe Bausteine ggf. hierarchisch

# Schaltsymbole

DIN 40700 (ab 1976)	Schaltzeichen		Benennung
	Früher	in USA	
			UND - Glied (AND)
			ODER - Glied (OR)
			NICHT - Glied (NOT)
			Exklusiv-Oder - Glied (Exclusive-OR, XOR)
			Äquivalenz - Glied (Logic identity)
			UND - Glied mit negier- tem Ausgang (NAND)
			ODER - Glied mit negier- tem 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:

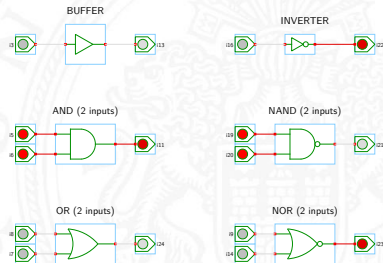
- ▶ mit Experimentierkasten oder im Logiksimulator
- ▶ interaktive Simulation erlaubt direktes Ausprobieren
- ▶ Animation und Visualisierung der logischen Werte
- ▶ „entdeckendes Lernen“
  
- ▶ Diglog: [www.cs.berkeley.edu/~lazzaro/chipmunk](http://www.cs.berkeley.edu/~lazzaro/chipmunk)
- ▶ Hades: [tams.informatik.uni-hamburg.de/applets/hades/webdemos](http://tams.informatik.uni-hamburg.de/applets/hades/webdemos)
  - ▶ Demos laufen im Browser (Java erforderlich)
  - ▶ Grundsaltungen, Gate-Level Circuits...

[tams.informatik.uni-hamburg.de/applets/hades/webdemos/toc.html](http://tams.informatik.uni-hamburg.de/applets/hades/webdemos/toc.html)

# Hades: Grundkomponenten

- ▶ Vorführung des Simulators
- ▶ Eingang: Schalter + Anzeige („Ipin“)
- ▶ Ausgang: Anzeige („Opin“)
- ▶ Taktgenerator
- ▶ PowerOnReset
- ▶ Anzeige / 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 $\Rightarrow$ keine Treiber
magenta	undefined-X $\Rightarrow$ Kurzschluss, ungültiger Wert
cyan	unknown-U $\Rightarrow$ nicht initialisiert

# Hades: Bedienung

- ▶ Menü: Anzeigeeoptionen, Edit-Befehle, usw.
- ▶ Editorfenster mit Popup-Menü für häufige Aktionen
- ▶ Rechtsklick auf Komponenten öffnet Eigenschaften/Parameter (*property-sheets*)
- ▶ optional „tooltips“ (enable im Layer-Menü)
- ▶ Simulationssteuerung: *run*, *pause*, *rewind*
- ▶ Anzeige der aktuellen Simulationszeit
- ▶ Details siehe Hades-Webseite: Kurzreferenz, Tutorial  
[tams.informatik.uni-hamburg.de/applets/hades/webdemos/docs.html](http://tams.informatik.uni-hamburg.de/applets/hades/webdemos/docs.html)

# Gatter: Verstärker, Inverter, AND, OR

**BUFFER**



**INVERTER**



**AND (2 inputs)**



**NAND (2 inputs)**



**OR (2 inputs)**

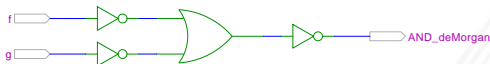


**NOR (2 inputs)**

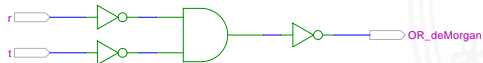


# Grundsaltungen: De'Morgan Regel

AND (2 inputs)



OR (2 inputs)

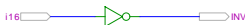


# Gatter: AND/NAND mit zwei, drei, vier Eingängen

**BUFFER**



**INVERTER**



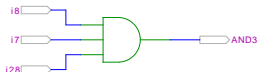
**AND (2 inputs)**



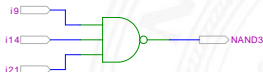
**NAND (2 inputs)**



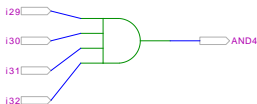
**AND (3 inputs)**



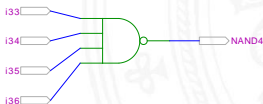
**NAND (3 inputs)**



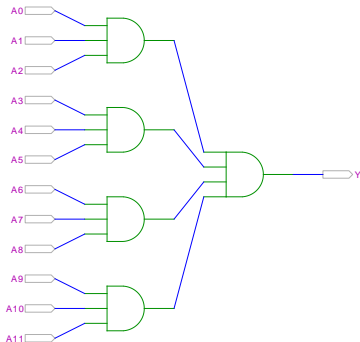
**AND (4 inputs)**



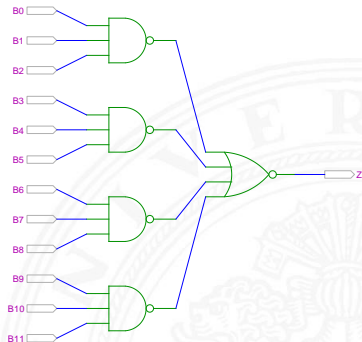
**NAND (4 inputs)**



## Gatter: AND mit zwölf Eingängen



AND3-AND4



NAND3-NOR4 (de-Morgan)

- in der Regel max. 4-Eingänge pro Gatter  
Grund: elektrotechnische Nachteile



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

**BUFFER**



**INVERTER**



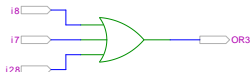
**OR (2 inputs)**



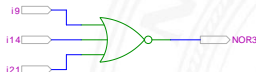
**NOR (2 inputs)**



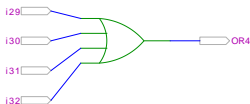
**OR (3 inputs)**



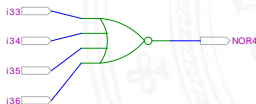
**NOR (3 inputs)**



**OR (4 inputs)**

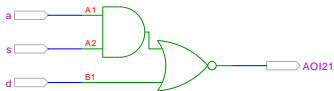


**NOR (4 inputs)**

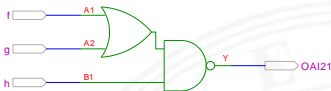


# Komplexgatter

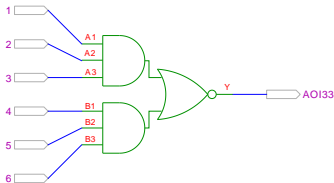
AOI21 (And-Or-Invert)



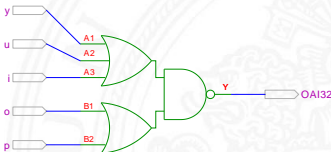
OAI21 (Or-And-Invert)



AOI33 (And-Or-Invert)



OAI32 (Or-And-Invert)



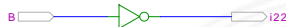
- ▶ in CMOS-Technologie besonders günstig realisierbar  
entsprechen vom Aufwand *einem* Gatter

# Gatter: XOR und XNOR

BUFFER



INVERTER



AND (2 inputs)



XOR (2 inputs)



OR (2 inputs)



XNOR (2 inputs)

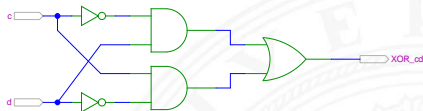


# XOR und drei Varianten der Realisierung

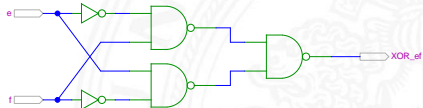
## ► Symbol



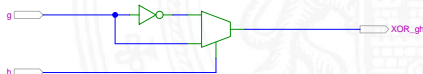
## ► AND-OR



## ► NAND-NAND



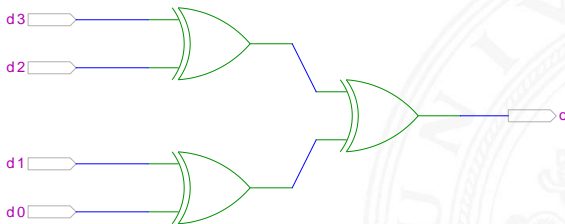
## ► mit Multiplexer



# XOR zur Berechnung der Parität

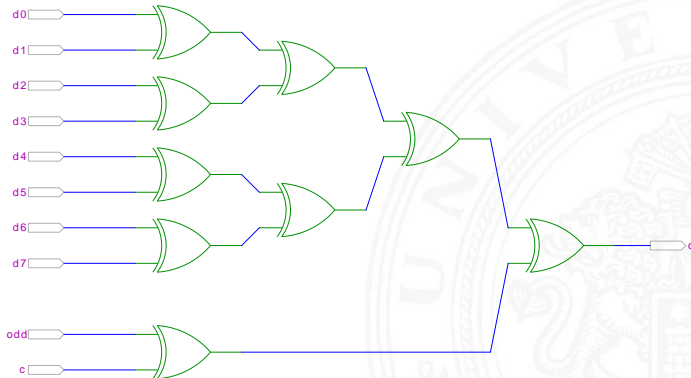
► Parität, siehe „Codierung – Fehlererkennende Codes“

► 4-bit Parität



## XOR zur Berechnung der Parität (cont.)

- 8-bit, 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  $a_0$   
ein Datenausgang:  $y$
- ▶ wenn  $s = 1$  wird  $a_1$  zum Ausgang  $y$  durchgeschaltet  
wenn  $s = 0$  wird  $a_0$  —"

$s$	$a_1$	$a_0$	$y$
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 (cont.)

- 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	$a_1$	*	$a_1$



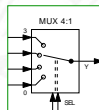
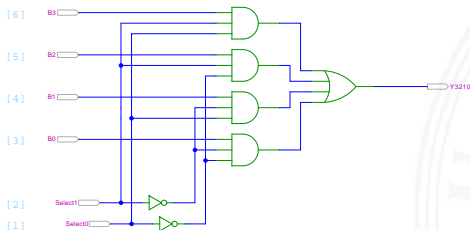
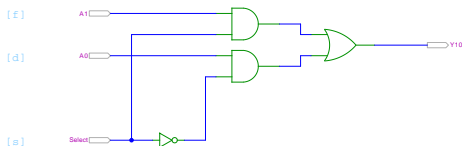
## n:1-Multiplexer

Umschalten zwischen mehreren Dateneingängen

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

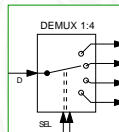
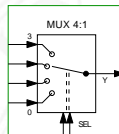
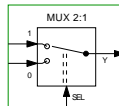
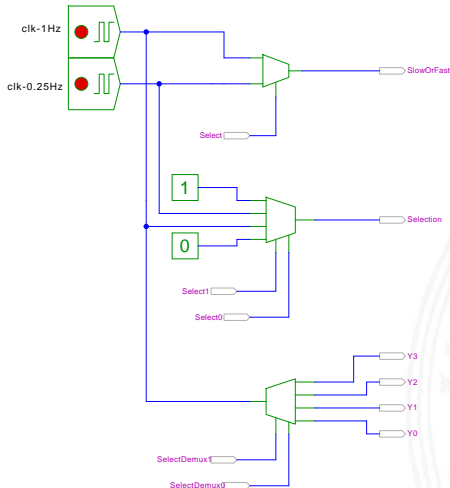
$s_1$	$s_0$	$a_3$	$a_2$	$a_1$	$a_0$	$y$
0	0	*	*	*	0	0
0	0	*	*	*	1	1
0	1	*	*	0	*	0
0	1	*	*	1	*	1
1	0	*	0	*	*	0
1	0	*	1	*	*	1
1	1	0	*	*	*	0
1	1	1	*	*	*	1

## 2:1 und 4:1 Multiplexer

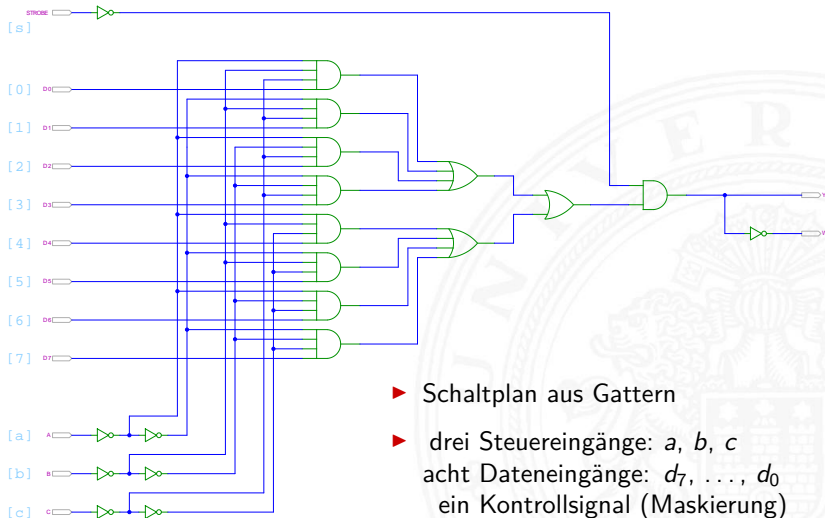


- ▶ keine einheitliche Anordnung der Dateneingänge in Schaltplänen:  
höchstwertigster Eingang manchmal oben, manchmal unten

# Multiplexer und Demultiplexer

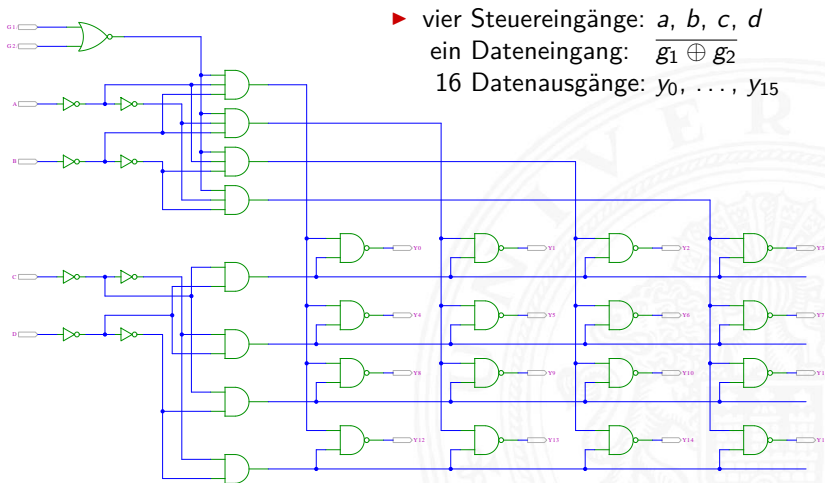


# 8-bit Multiplexer: Integrierte Schaltung 74151

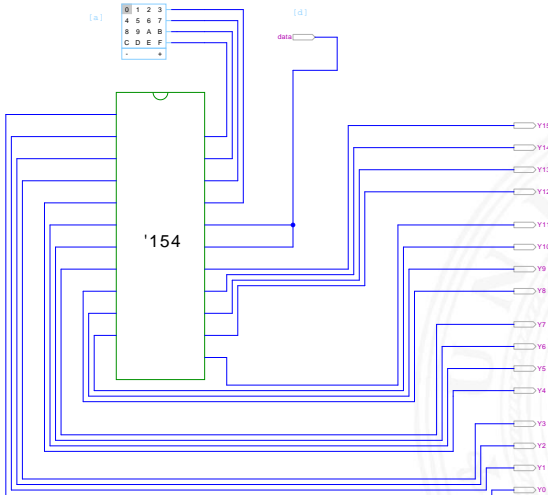


- ▶ Schaltplan aus Gattern
- ▶ drei Steuereingänge:  $a$ ,  $b$ ,  $c$   
acht Dateneingänge:  $d_7, \dots, d_0$   
ein Kontrollsignal (Maskierung)

# 16-bit Demultiplexer: Integrierte Schaltung 74154



# 16-bit Demultiplexer: 74154 als Adressdecoder



# Beispiele für Schaltnetze

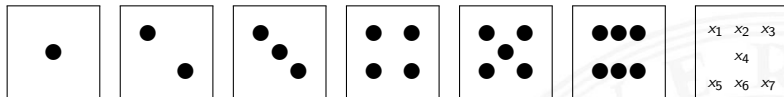
- ▶ Schaltungen mit mehreren Ausgängen
- ▶ Bündelminimierung der einzelnen Funktionen

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

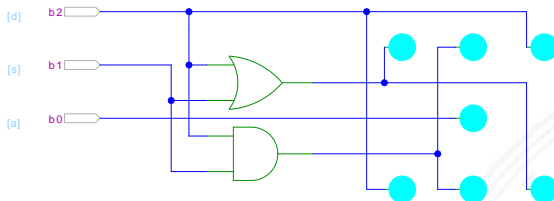


- ▶ Eingabewert von 0...6
- ▶ Anzeige als ein bis sechs Augen, bzw. ausgeschaltet

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 (cont.)



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

$$x_1 = x_7 = b_2 \vee b_1$$

$$x_2 = x_6 = b_0 \wedge b_1$$

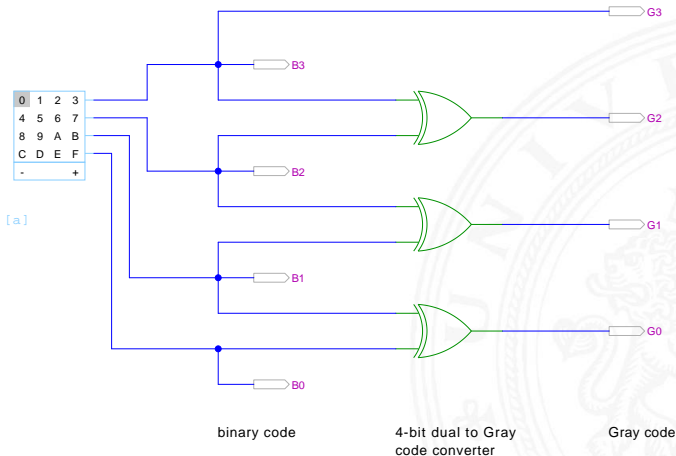
$$x_3 = x_5 = b_2$$

$$x_4 = b_0$$

links oben, rechts unten  
mitte oben, mitte unten  
rechts oben, links unten  
Zentrum

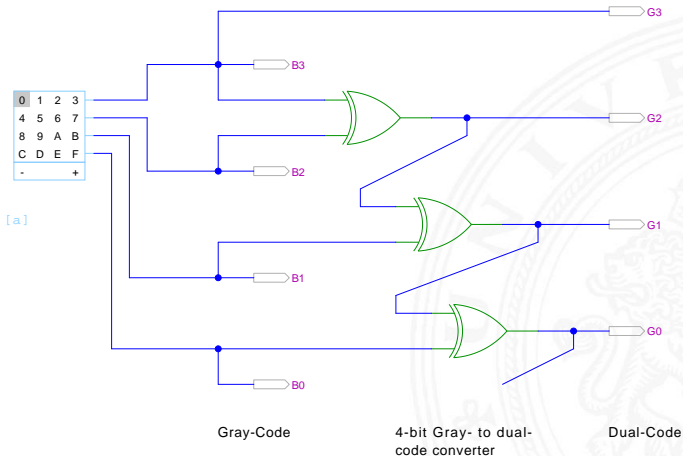
# Beispiel: Umwandlung vom Dualcode in den Graycode

## XOR benachbarter Bits



# Beispiel: Umwandlung vom Graycode in den Dualcode

## XOR-Kette



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

## ► Encoder

- vier Eingabebits
- Hamming-Encoder erzeugt drei Paritätsbits

linke Seite

## ► Übertragungskanal

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

Mitte

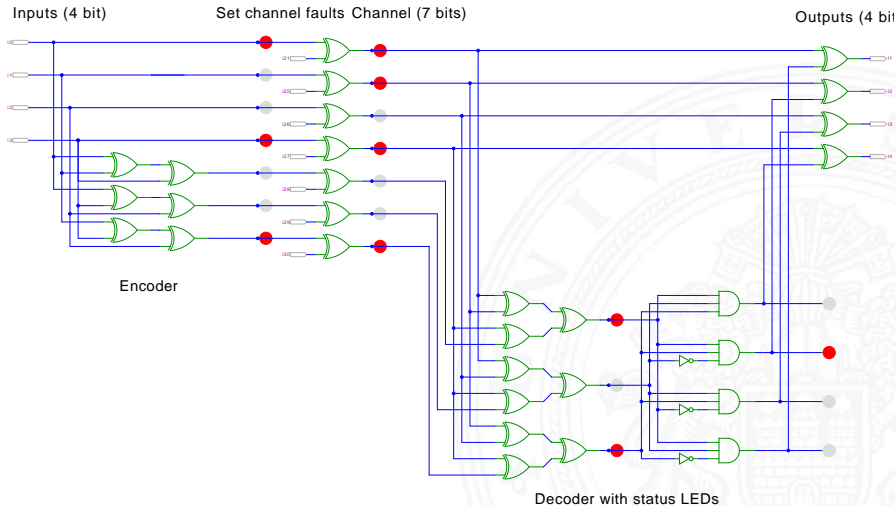
## ► Decoder und Fehlerkorrektur

- Decoder liest die empfangenen sieben Bits
- Syndrom-Berechnung mit XOR-Gattern und Anzeige erkannter Fehler
- Korrektur gekippter Bits

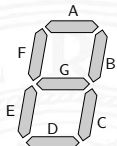
rechte Seite

rechts oben

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



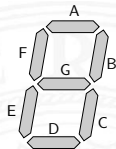
# Siebensegmentanzeige

- ▶ 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
  - ▶ Segmente im Uhrzeigersinn: A (oben) bis F, G innen
- 
- ▶ eingeschränkt auch als alphanumerische Anzeige für Ziffern und (einige) Buchstaben
    - gemischt Groß- und Kleinbuchstaben
    - Probleme mit M, N, usw.

# Siebensegmentanzeige: Funktionen

- Funktionen für Hex-Anzeige, 0...F

	0	1	2	3	4	5	6	8	9	A	B	C	D	E	F
A =	1	0	1	1	0	1	1	1	1	1	0	0	0	1	1
B =	1	1	1	1	1	0	0	1	1	1	1	0	0	1	0
C =	1	1	0	1	1	1	1	1	1	1	1	1	0	1	0
D =	1	0	1	1	0	1	1	0	1	1	0	1	1	1	0
E =	1	0	1	0	0	0	1	0	1	0	1	1	1	1	1
F =	1	0	0	0	1	1	1	1	1	1	1	0	0	1	1
G =	0	0	1	1	1	1	0	1	1	1	1	1	1	1	1



- für Ziffernanzeige mit *Don't Care*-Termen

A = 1011011111\*\*\*\*\*  
 B = usw.

# Siebensegmentanzeige: Bündelminimierung

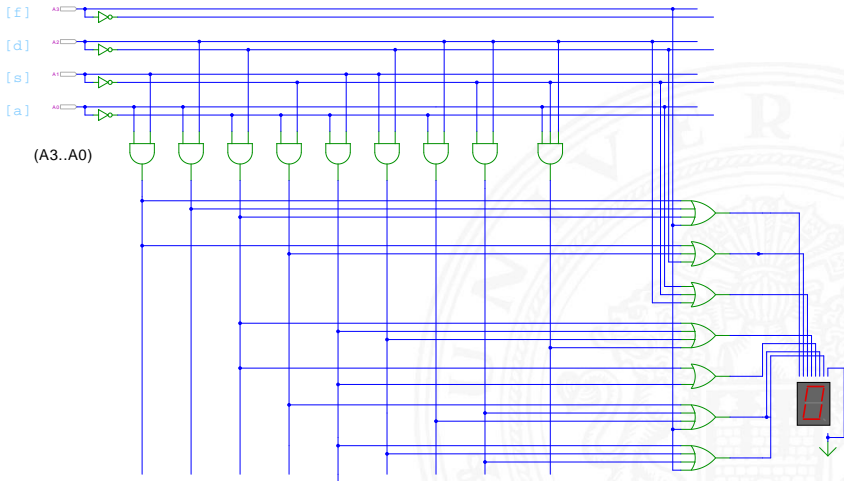
- ▶ zum Beispiel mit sieben KV-Diagrammen...
- ▶ dabei versuchen, gemeinsame Terme zu finden und zu nutzen

Minimierung als Übungsaufgabe?

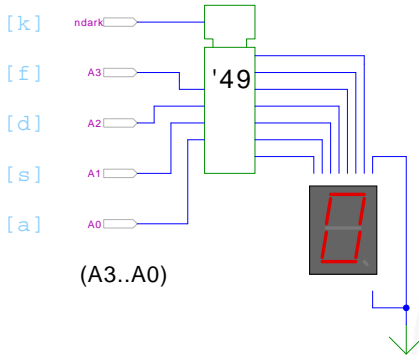
- ▶ nächste Folie zeigt Lösung aus Schiffmann, Schmitz
- ▶ als mehrstufige Schaltung ist günstigere Lösung möglich  
siehe Knuth: *AoCP, Volume 4, Fascicle 0*, 7.1.2 (Seite 112ff)



# Siebensegmentdecoder: Ziffern 0..9

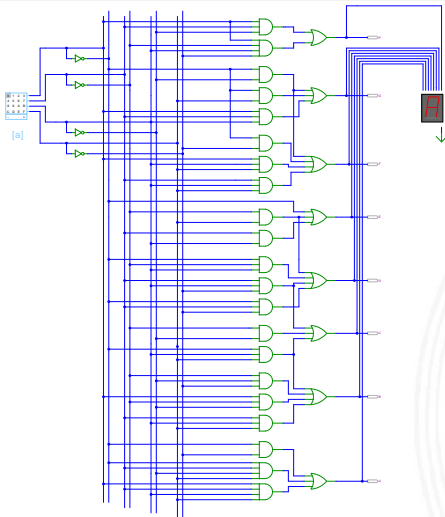


# Siebensegmentdecoder: Integrierte Schaltung 7449



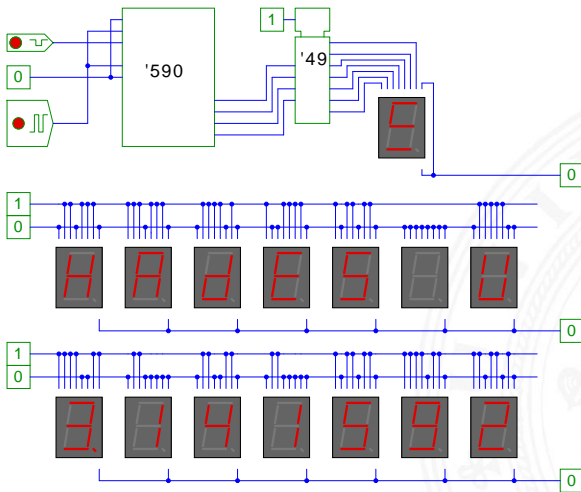
- ▶ Beispiel für eine integrierte Schaltung (IC)
- ▶ Anzeige von 0..9, Zufallsmuster für A..F, „Dunkeltastung“

# Siebensegmentanzeige: Hades-Beispiele



► Buchstaben A...P

# Siebensegmentanzeige: Hades-Beispiele (cont.)



# 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  $a, \dots, g$

$$x_5 = x_2 \oplus x_3$$

$$x_6 = \overline{x_1} \wedge x_4$$

$$x_7 = x_3 \wedge \overline{x_6}$$

$$x_8 = x_1 \oplus x_2$$

$$x_9 = x_4 \oplus x_5$$

$$x_{10} = \overline{x_7} \wedge x_8$$

$$x_{11} = x_9 \oplus x_{10}$$

$$x_{12} = x_5 \wedge x_{11}$$

$$x_{13} = x_1 \oplus x_7$$

$$x_{14} = x_5 \oplus x_6$$

$$x_{15} = x_7 \vee x_{12}$$

$$x_{16} = x_1 \vee x_5$$

$$x_{17} = x_5 \vee x_6$$

$$x_{18} = x_9 \wedge x_{10}$$

$$x_{19} = x_3 \wedge x_9$$

$$\overline{a} = x_{20} = x_{14} \wedge \overline{x_{19}}$$

$$\overline{b} = x_{21} = x_7 \oplus x_{12}$$

$$\overline{c} = x_{22} = \overline{x_8} \wedge x_{15}$$

$$\overline{d} = x_{23} = x_9 \wedge \overline{x_{13}}$$

$$\overline{e} = x_{24} = x_6 \vee x_{18}$$

$$\overline{f} = x_{25} = \overline{x_8} \wedge x_{17}$$

$$g = x_{26} = x_7 \vee x_{16}$$

# Logische und arithmetische Operationen

- ▶ Halb- und Volladdierer
- ▶ Addierertypen
  - ▶ Ripple-Carry
  - ▶ Carry-Lookahead
- ▶ 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  $c_o$  (*carry-out*) von zwei Eingangsbits  $a$ ,  $b$  sowie Eingangsübertrag  $c_i$  (*carry-in*)

$a$	$b$	$c_i$	$c_o$	$s$
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

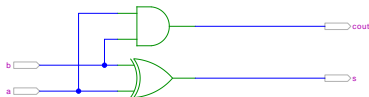
$$c_o = ab \vee ac_i \vee bc_i = (ab) \vee (a \vee b)c_i$$

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

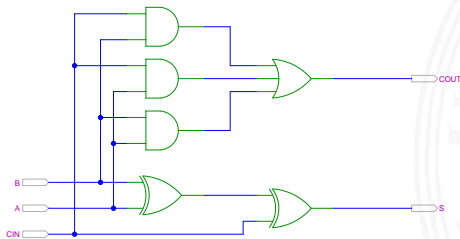


# Schaltbilder für Halb- und Volladdierer

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



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



## $n$ -bit Addierer

- Summe:  $s_n = a_n \oplus b_n \oplus c_n$

$$s_0 = a_0 \oplus b_0$$

$$s_1 = a_1 \oplus b_1 \oplus c_1$$

$$s_2 = a_2 \oplus b_2 \oplus c_2$$

...

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

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

$$c_1 = (a_0 b_0)$$

$$c_2 = (a_1 b_1) \vee (a_1 \vee b_1) c_1$$

$$c_3 = (a_2 b_2) \vee (a_2 \vee b_2) c_2$$

...

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

## *n*-bit Addierer (cont.)

- ▶ *n*-bit Addierer theoretisch als zweistufige Schaltung realisierbar
- ▶ direkte und negierte Eingänge, dann AND-OR Netzwerk
- ▶ Aufwand steigt exponentiell mit *n* an,  
für Ausgang *n* sind  $2^{(2n-1)}$  Minterme erforderlich
- ⇒ nicht praktikabel
- ▶ Problem: Übertrag (*carry*)  

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

## $n$ -bit Addierer (cont.)

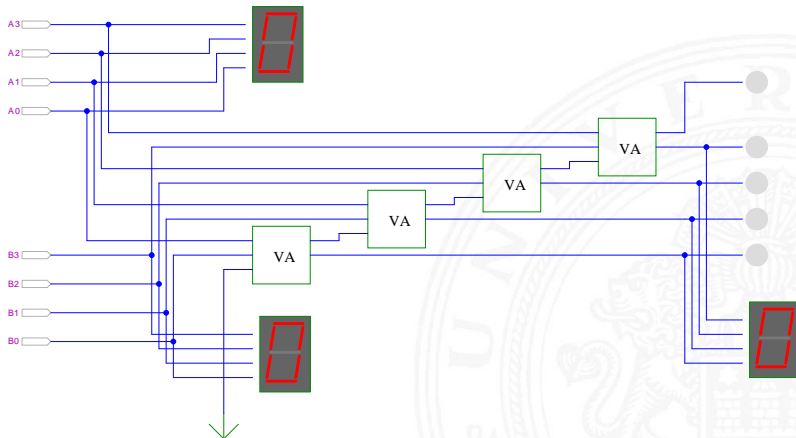
Diverse gängige Alternativen für Addierer

- ▶ Ripple-Carry
  - ▶ lineare Struktur
  - + klein, einfach zu implementieren
  - langsam, Laufzeit  $O(n)$
- ▶ Carry-Lookahead (CLA)
  - ▶ Baumstruktur
  - + schnell
  - teuer (Flächenbedarf der Hardware)
- ▶ Mischformen: Ripple-block CLA, Block CLA, Parallel Prefix
- ▶ andere Ideen: Carry Select, Conditional Sum, Carry Skip
- ...

# Ripple-Carry Adder

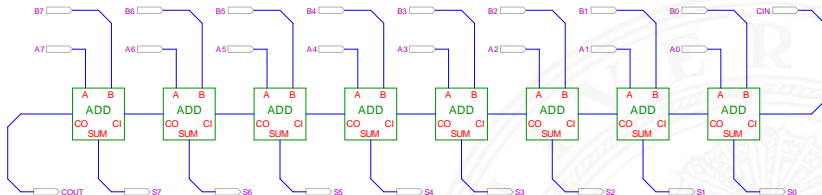
- ▶ Kaskade aus  $n$  einzelnen Volladdierern
- ▶ Carry-out von Stufe  $i$  treibt Carry-in von Stufe  $i + 1$
- ▶ Gesamtverzögerung wächst mit der Anzahl der Stufen als  $O(n)$
- ▶ Addierer in Prozessoren häufig im *kritischen Pfad*
- ▶ möglichst hohe Performance ist essentiell
- ▶ ripple-carry in CMOS-Technologie bis ca. 10-bit geeignet
- ▶ bei größerer Wortbreite gibt es effizientere Schaltungen

# Ripple-Carry Adder: 4-bit



# Ripple-Carry Adder: Hades-Beispiel 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

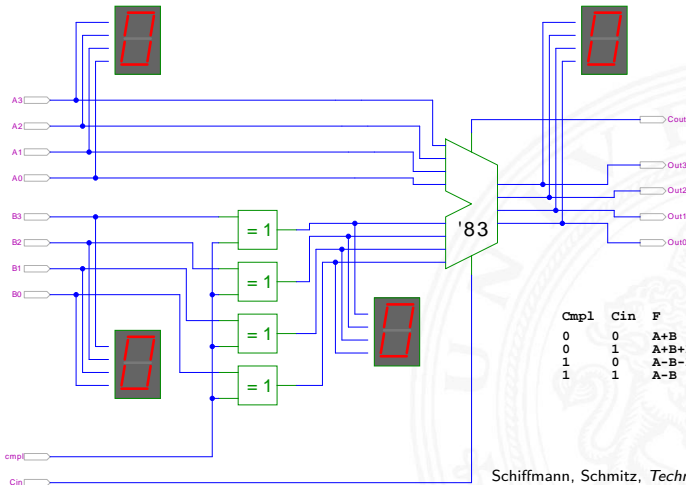
- ▶  $(A - B)$  ersetzt durch Addition des 2-Komplements von  $B$
- ▶ 2-Komplement: Invertieren aller Bits und Addition von Eins
- ▶ Carry-in Eingang des Addierers bisher nicht benutzt

## 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: Beispiel (7483 – 4-bit Addierer)



Schiffmann, Schmitz, *Technische Informatik I*

# Schnelle Addierer

- ▶ Addierer in Prozessoren häufig im *kritischen Pfad*
- ▶ möglichst hohe Performance ist essentiell
- ⇒ bestimmt Taktfrequenz
- ▶ Carry-Select Adder: Gruppen von Ripple-carry
- ▶ Carry-Lookahead Adder: Baumstruktur zur Carry-Berechnung
- ▶ ...
- ▶ über 10 Addierer „Typen“ (für 2 Operanden)
- ▶ Addition mehrerer Operanden
- ▶ Typen teilweise technologieabhängig

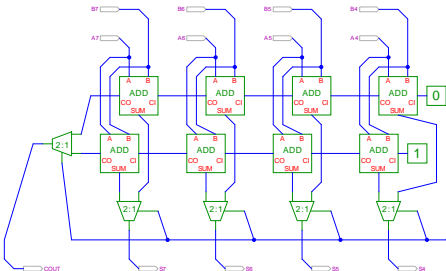
## Carry-Select Adder: Prinzip

- ▶ 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  $c_i = 0$  ( $a + b$ ), der andere mit  $c_i = 1$  ( $a + b + 1$ )
    - ▶ 2:1-Multiplexer mit  $m_i$ -bit wählt die korrekte Summe aus
  - ▶ Sobald der Wert von  $c_i$  bekannt ist (Ripple-Carry), wird über den Multiplexer die benötigte Zwischensumme ausgewählt
  - ▶ Das berechnete Carry-out  $c_o$  der Gruppe ist das Carry-in  $c_i$  der folgenden Gruppe
- ⇒ Verzögerung reduziert sich auf die Verzögerung eines  $m$ -bit Addierers plus die Verzögerungen der Multiplexer

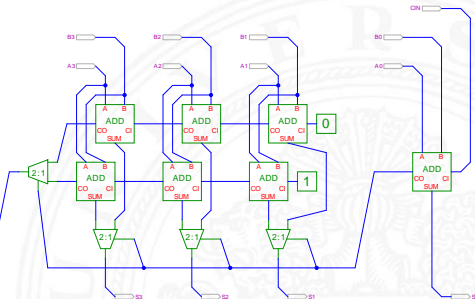
# Carry-Select Adder: Beispiel

## 8-Bit Carry-Select Adder (4 + 3 + 1 bit blocks)

### 4-bit Carry-Select Adder block

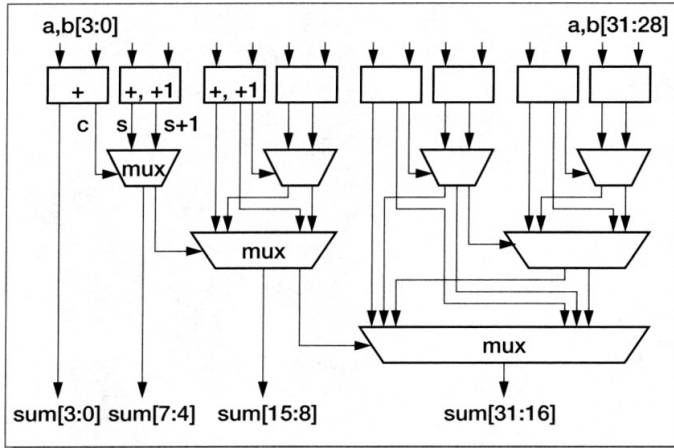


### 3-bit Carry-Select Adder block



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

# Carry-Select Adder: Beispiel ARM v6



# Carry-Lookahead Adder: Prinzip

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

- Einführung von Hilfsfunktionen

$$g_n = (a_n b_n)$$

$$p_n = (a_n \vee b_n)$$

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

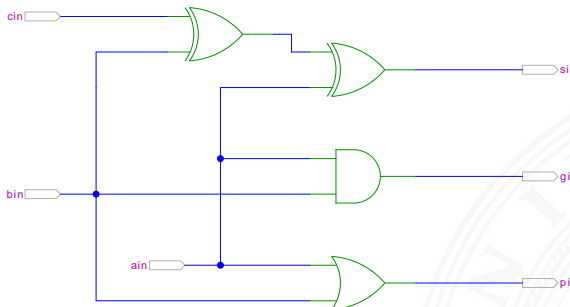
„generate carry“

„propagate carry“

- *generate*: Carry out erzeugen, unabhängig von Carry-in  
 ► *propagate*: Carry out weiterleiten / Carry-in maskieren

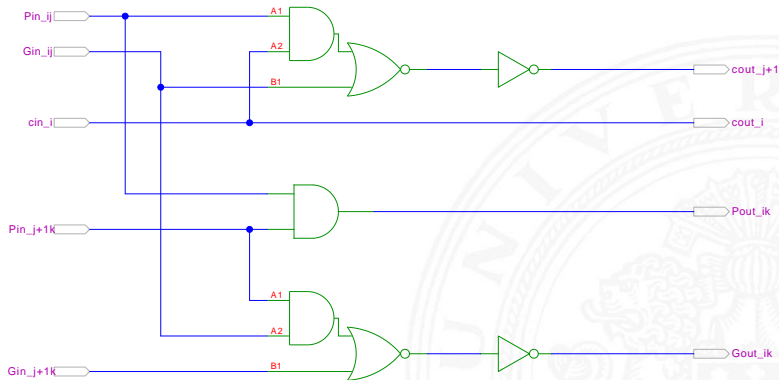
- Berechnung der  $g_n$  und  $p_n$  in einer Baumstruktur  
 Tiefe des Baums ist  $\log_2 N \Rightarrow$  entsprechend schnell

# Carry-Lookahead Adder: SUM-Funktionsblock



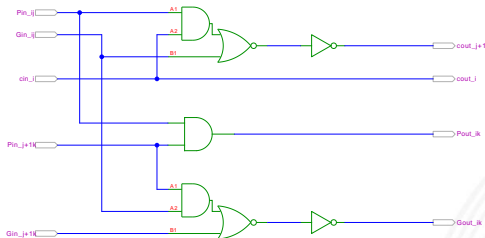
- ▶ 1-bit Addierer,  $s = a_i \oplus b_i \oplus c_i$
- ▶ keine Berechnung des Carry-Out
- ▶ Ausgang  $g_i = a_i \wedge b_i$  liefert *generate-carry* Signal
- ▶ Ausgang  $p_i = a_i \vee b_i$  liefert *propagate-carry* Signal

# Carry-Lookahead Adder: CLA-Funktionsblock





## Carry-Lookahead Adder: CLA-Funktionsblock (cont.)



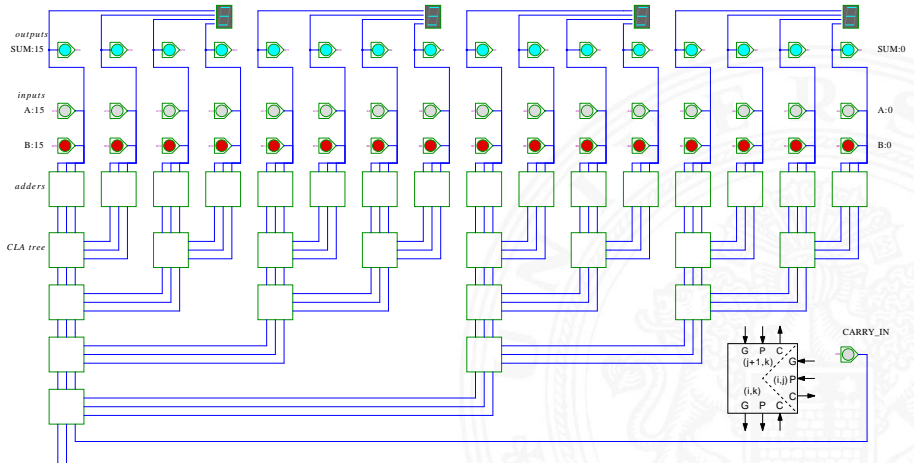
### ► Eingänge

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

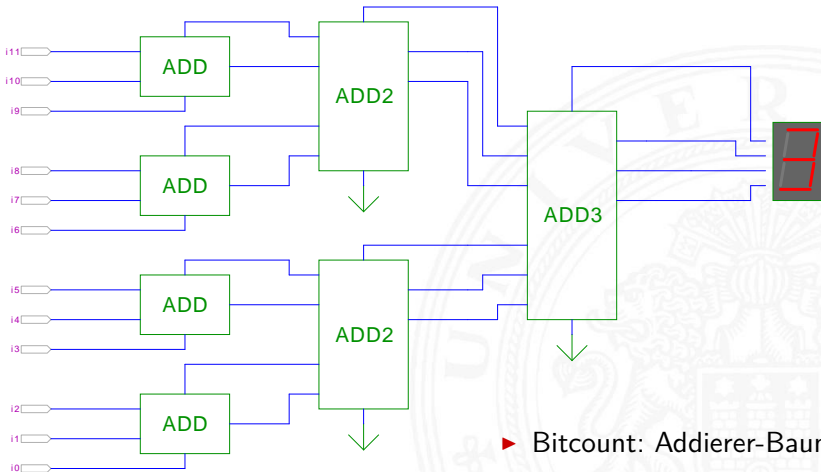
### ► Ausgänge

- propagate/generate Signale zur nächsthöheren Stufe
- carry-out Signale: Durchleiten und zur nächsthöheren Stufe

# Carry-Lookahead Adder: 16-bit Addierer



# Addition mehrerer Operanden



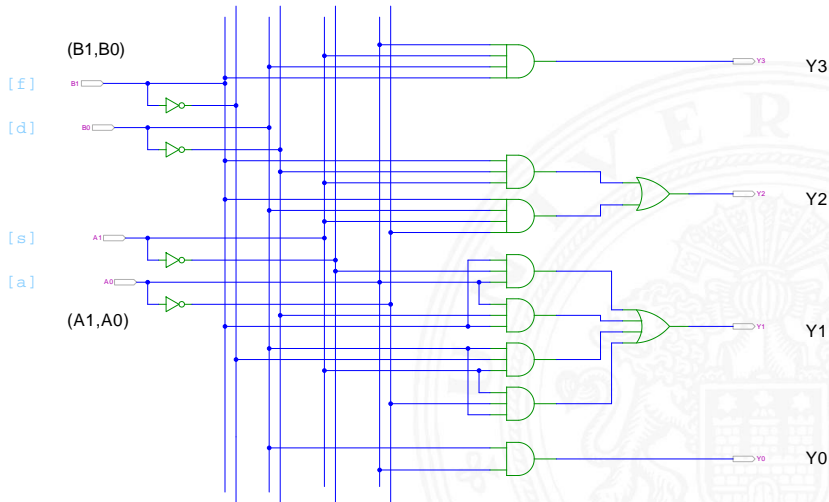
## Addierer: Zusammenfassung

- ▶ Halbaddierer ( $a \oplus b$ )
- ▶ Volladdierer ( $a \oplus b \oplus c_i$ )
- ▶ Ripple-carry
  - ▶ Kaskade aus Volladdierern, einfach und billig
  - ▶ aber manchmal zu langsam, Verzögerung:  $O(n)$
- ▶ Carry-select Prinzip
  - ▶ Verzögerung  $O(\sqrt{n})$
- ▶ Carry-lookahead Prinzip
  - ▶ Verzögerung  $O(\ln n)$
- ▶ Subtraktion durch Zweierkomplementbildung erlaubt auch Inkrement ( $A++$ ) und Dekrement ( $A--$ )

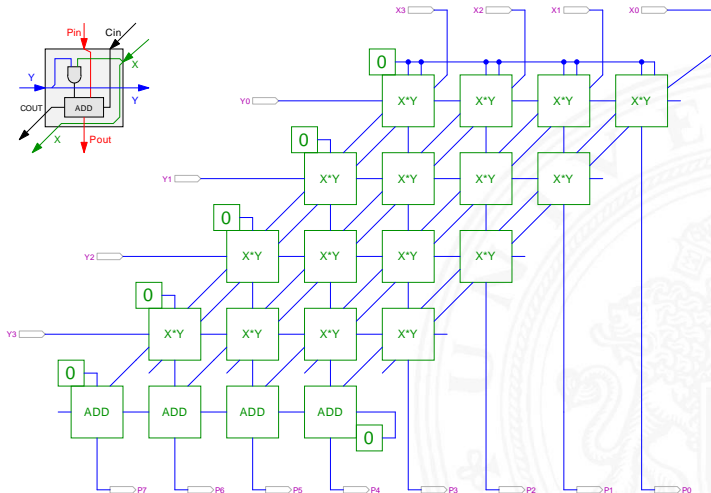
# Multiplizierer

- ▶ Teilprodukte als UND-Verknüpfung des Multiplikators mit je einem Bit des Multiplikanden
- ▶ Aufaddieren 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 (Automaten) mit sukzessiver Berechnung des Produkts in mehreren Takten durch Addition und Schieben

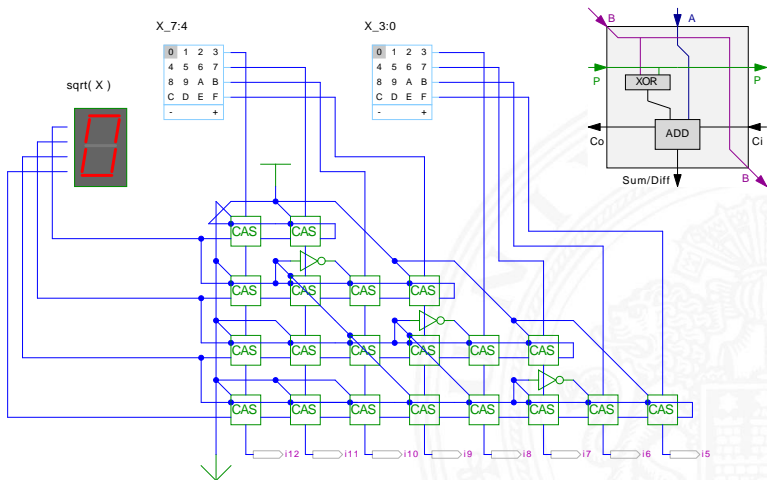
## 2x2-bit Multiplizierer – als zweistufiges Schaltnetz



# 4x4-bit Multiplizierer – Array



# 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

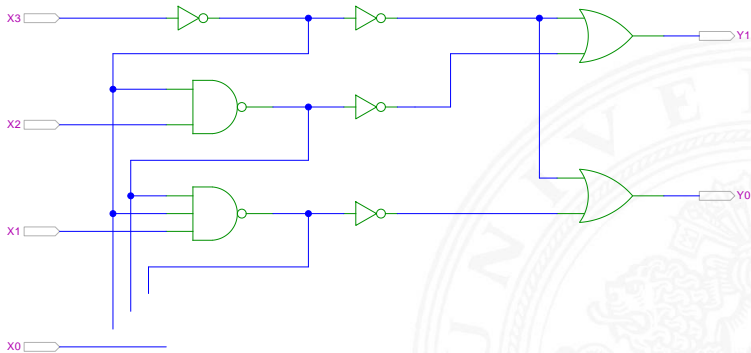
# Priority Encoder

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

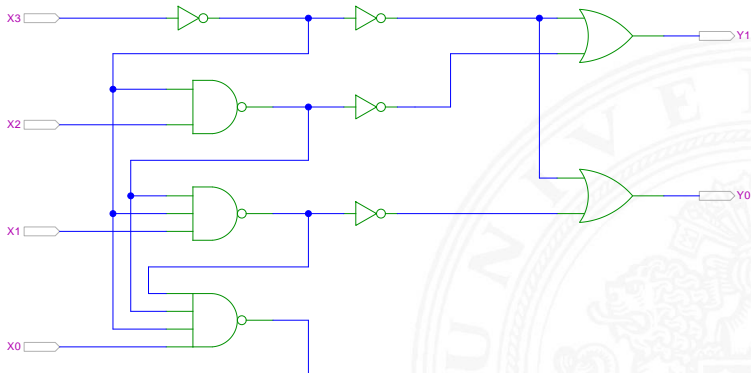
- ▶ unabhängig von niederwertigstem Bit,  $x_0$  kann entfallen

## 4:2 Prioritätsencoder

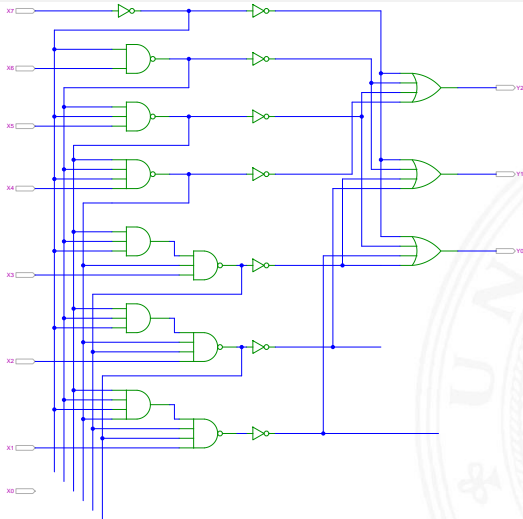


- ▶ zweistufige Realisierung
- ▶ aktive höhere Stufe blockiert alle niedrigeren Stufen

## 4:2 Prioritätsencoder: Kaskadierung

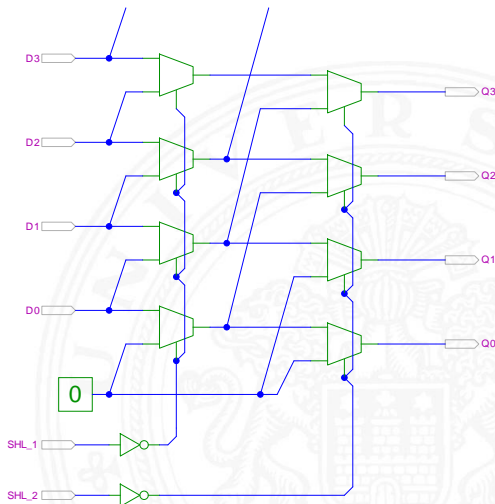


## 8:3 Prioritätsencoder

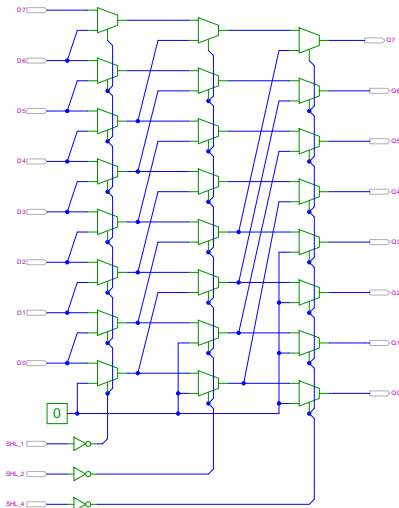


## Shifter: zweistufig, shift-left um 0...3 Bits

- ▶  $n$ -Dateneingänge  $D_i$   
 $n$ -Datenausgänge  $Q_i$
- ▶ 2:1 Multiplexer Kaskade
  - ▶ Stufe 0: benachbarte Bits
  - ▶ Stufe 1: übernächste Bits
  - ▶ usw.
- ▶ von rechts 0 nachschieben



# 8-bit Barrel-Shifter



## Shift-Right, Rotate etc.

- ▶ Prinzip der oben vorgestellten Schaltungen gilt auch für alle übrigen Shift- und Rotate-Operationen
- ▶        Logic shift right: von links Nullen nachschieben  
Arithmetic shift right: oberstes Bit nachschieben
- ▶ Rotate left / right: außen herausgeschobene Bits auf der anderen Seite wieder hereinschieben
- + alle Operationen typischerweise in einem Takt realisierbar
- Problem: Hardwareaufwand bei großen Wortbreiten und beliebigem Schiebe-/Rotate-Argument



# Arithmetisch-Logische Einheit (ALU)

## Arithmetisch-logische Einheit ALU (*Arithmetic Logic Unit*)

- ▶ kombiniertes Schaltnetz für arithmetische und logische Operationen
- ▶ das zentrale Rechenwerk in Prozessoren

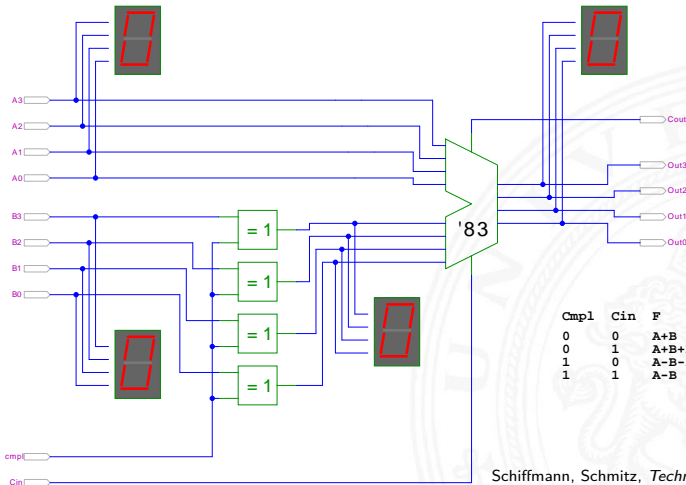
Funktionsumfang variiert von Typ zu Typ

- ▶ 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

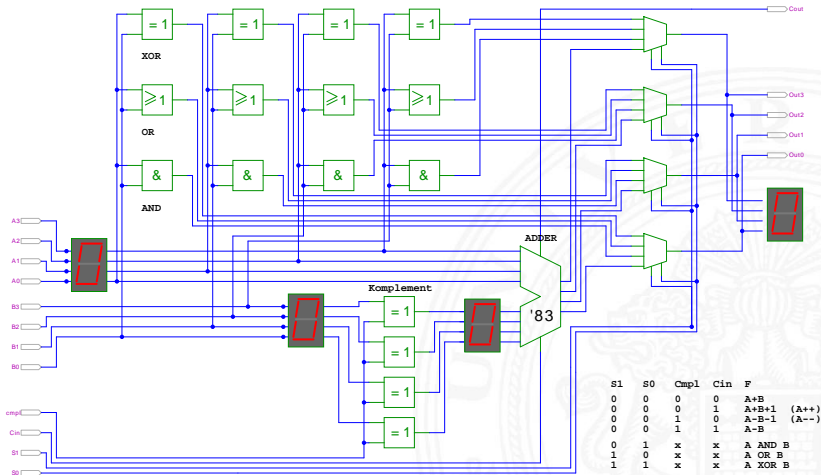
- ▶ Addition ( $A + B$ ) mit normalem Addierer
- ▶ XOR-Gatter zum Invertieren von Operand  $B$
- ▶ Steuerleitung *sub* aktiviert das Invertieren und den Carry-in  $c_i$
- ▶ wenn aktiv, Subtraktion als  $(A - B) = A + \neg B + 1$
- ▶ ggf. auch Inkrement ( $A + 1$ ) und Dekrement ( $A - 1$ )
- ▶ folgende Folien: 7483 ist IC mit 4-bit Addierer

# ALU: Addierer und Subtrahierer



Schiffmann, Schmitz, *Technische Informatik I*

# ALU: Addierer und bitweise Operationen



# ALU: Prinzip

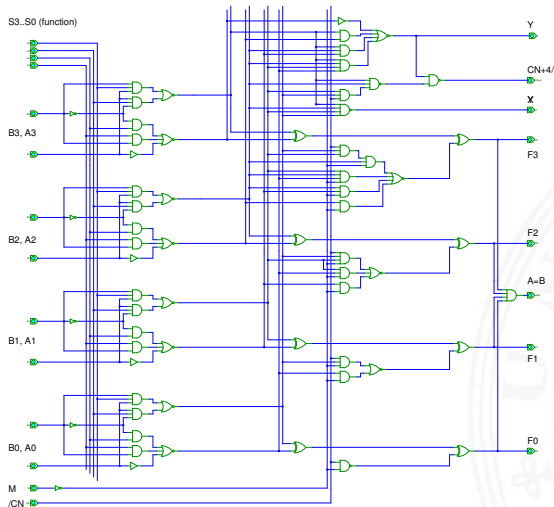
vorige Folie zeigt die „triviale“ Realisierung einer ALU

- ▶ 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 (IC 74181)

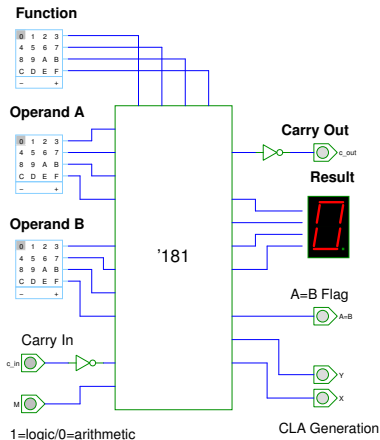
- ▶ erste Stufe für bitweise logische Operationen und Komplement
- ▶ zweite Stufe als Carry-lookahead Addierer
- ▶ weniger Gatter und schneller

# ALU: 74181 – Aufbau



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

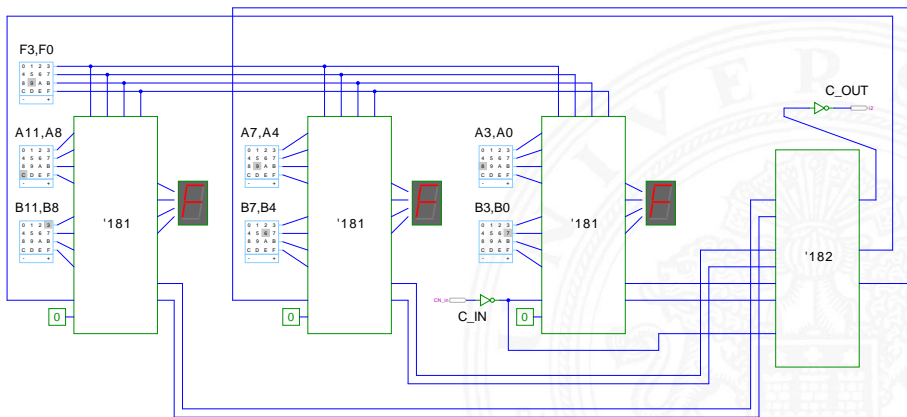
# ALU: 74181 – Funktionstabelle



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

# 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: Introduction to Combinatorial Algorithms and Boolean Functions*, Addison-Wesley, 2008
- ▶ Donald E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques, Binary Decision Diagrams*, Addison-Wesley, 2009
- ▶ Ingo Wegener, *The Complexity of Boolean Functions*, Wiley, 1987     [ls2-www.cs.uni-dortmund.de/monographs/bluebook](http://ls2-www.cs.uni-dortmund.de/monographs/bluebook)
- ▶ Bernd Becker, Rolf Drechsler, Paul Molitor, *Technische Informatik: Eine Einführung*, Pearson Studium, 2005  
Besonderheit: Einführung von BDDs/ROBDDs

# Interaktives Lehrmaterial

- ▶ Klaus von der Heide,  
*Vorlesung: Technische Informatik 1 — interaktives Skript*  
[tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1](http://tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1)
  
- ▶ Norman Hendrich,  
*HADES — HAmBurg DEsign System*  
[tams.informatik.uni-hamburg.de/applets/hades](http://tams.informatik.uni-hamburg.de/applets/hades)  
  
*KV-Diagram Simulation*  
[tams.informatik.uni-hamburg.de/applets/kvd](http://tams.informatik.uni-hamburg.de/applets/kvd)
  
- ▶ John Lazzaro,  
*Chipmunk design tools (AnaLog, DigLog)*  
[www.cs.berkeley.edu/~lazzaro/chipmunk](http://www.cs.berkeley.edu/~lazzaro/chipmunk)