



64-040 Modul InfB-RSB

Rechnerstrukturen und Betriebssysteme

[https://tams.informatik.uni-hamburg.de/
lectures/2021ws/vorlesung/rsb](https://tams.informatik.uni-hamburg.de/lectures/2021ws/vorlesung/rsb)

– Kapitel 9 –

Andreas Mäder



Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

Wintersemester 2021/2022

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

Barrel-Shifter

ALU (Arithmetisch-Logische Einheit)

Zeitverhalten von Schaltungen



Hazards
Literatur



- ▶ **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 $\vec{y} = F(\vec{b})$

- ▶ Bündel von Schaltfunktionen (mehrere SF)
- ▶ ein Schaltnetz darf keine Rückkopplungen enthalten

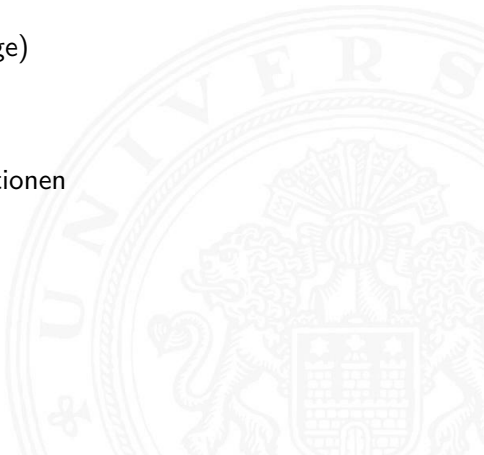
- ▶ Begriff: „Schaltnetz“
 - ▶ technische Realisierung von Schaltfunktionen / Funktionsbündeln
 - ▶ Struktur aus einfachen Gatterfunktionen:
triviale Funktionen mit wenigen (2...4) Eingängen
- ▶ in der Praxis können Schaltnetze nicht statisch betrachtet werden: Gatterlaufzeiten spielen eine Rolle



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

- ▶ Schaltnetze (mehrere Ausgänge)
- ▶ Beispiele

- ▶ Arithmetisch/Logische Operationen





- ▶ 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)
 - ▶ Anschlusspunkte für Drahtverbindungen
 - ▶ dicke Linien für n -bit Busse, Anzapfungen usw.
- ▶ komplexe Bausteine, hierarchisch zusammengesetzt

DIN 40700 (ab 1976)	Schaltzeichen		Benennung
	Früher	in USA	
			UND - Glied (AND)
			ODER - Glied (OR)
			NICHT - Glied (NOT)
			Exklusiv-Oder - Glied (Exclusive-OR, XOR)
			Aquivalenz - Glied (Logic identity)
			UND - Glied mit negier- tem Ausgang (NAND)
			ODER - Glied mit negier- tem Ausgang (NOR)
			Negation eines Eingangs
			Negation eines Ausgangs

- ▶ **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

- ▶ vollständige Basismenge erforderlich (mindestens 1 Gatter)
- ▶ in Halbleitertechnologie sind NAND/NOR besonders effizient

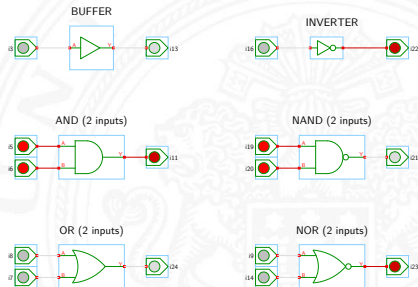
Spielerischer Zugang zu digitalen Schaltungen:

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

- ▶ Diglog: john-lazzaro.github.io/chipmunk [Laz]
- ▶ Hades: tams.informatik.uni-hamburg.de/applets/hades/webdemos [HenHA]
tams.informatik.uni-hamburg.de/applets/hades/webdemos/toc.html
 - ▶ Demos laufen im Browser (Java erforderlich)
 - ▶ Grundsaltungen, Gate-Level Circuits ...
einfache Prozessoren ...

- ▶ Vorführung des Simulators
Hades Demo: 00-intro/00-welcome/chapter
- ▶ Eingang: Schalter + Anzeige („Ipin“)
- ▶ Ausgang: Anzeige („Opin“)
- ▶ Taktgenerator
- ▶ PowerOnReset
- ▶ Anzeige / Leuchtdiode
- ▶ Siebensegmentanzeige
- ...

[HenHA] Hades Demo: 10-gates/00-gates/basic





- ▶ 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 kein Treiber (bidirektionale Busse)
magenta	undefined	X \Rightarrow Kurzschluss, ungültiger Wert
cyan	unknown	U \Rightarrow nicht initialisiert

- ▶ Menü: Anzeigoptionen, 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

Gatter: Verstärker, Inverter, AND, OR

9.4.1 Schaltnetze - Logische Gatter - Inverter, AND, OR

64-040 Rechnerstrukturen und Betriebssysteme

BUFFER



INVERTER



AND (2 inputs)



NAND (2 inputs)



OR (2 inputs)



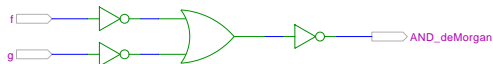
NOR (2 inputs)



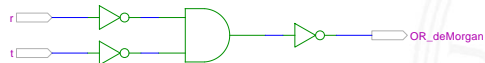
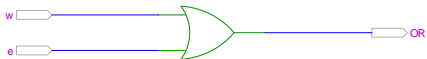
[HenHA] Hades Demo: 10-gates/00-gates/basic

Grundsaltungen: De Morgan Regel

AND (2 inputs)



OR (2 inputs)



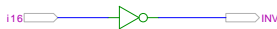
[HenHA] Hades Demo: 10-gates/00-gates/de-morgan

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

BUFFER



INVERTER



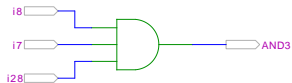
AND (2 inputs)



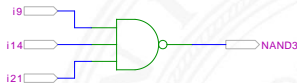
NAND (2 inputs)



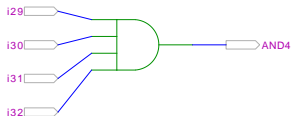
AND (3 inputs)



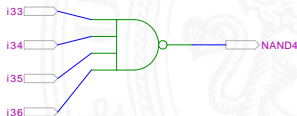
NAND (3 inputs)



AND (4 inputs)

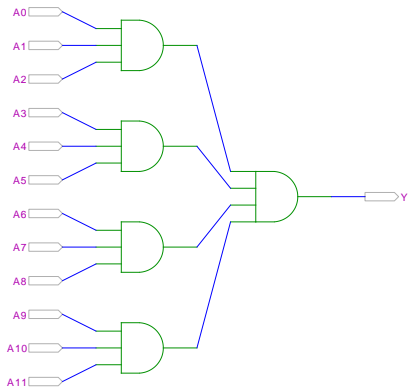


NAND (4 inputs)

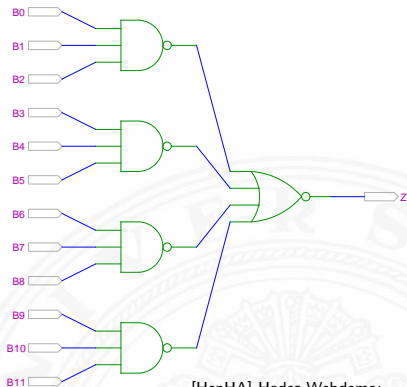


[HenHA] Hades Demo: 10-gates/00-gates/and

Gatter: AND mit zwölf Eingängen



AND3-AND4



NAND3-NOR4 (De Morgan)

[HenHA] Hades Webdemo:
10-gates/00-gates/andbig

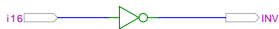
- ▶ in der Regel max. 4 Eingänge pro Gatter
Grund: elektrotechnische Nachteile

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

BUFFER



INVERTER



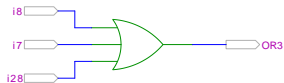
OR (2 inputs)



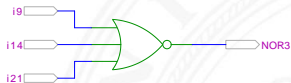
NOR (2 inputs)



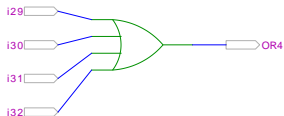
OR (3 inputs)



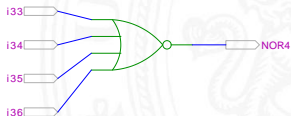
NOR (3 inputs)



OR (4 inputs)

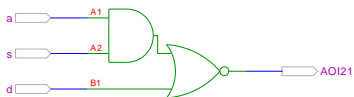


NOR (4 inputs)

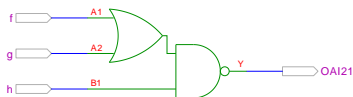


[HenHA] Hades Demo: 10-gates/00-gates/or

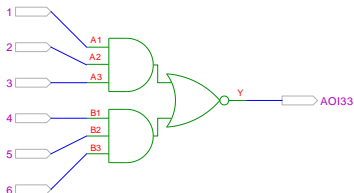
AOI21 (And-Or-Invert)



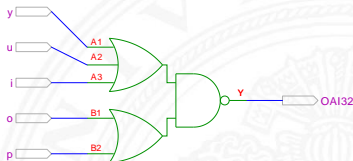
OAI21 (Or-And-Invert)



AOI33 (And-Or-Invert)



OAI32 (Or-And-Invert)



[HenHA] Hades Demo: 10-gates/00-gates/complex

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

Gatter: XOR und XNOR

BUFFER



INVERTER



AND (2 inputs)



XOR (2 inputs)



OR (2 inputs)



XNOR (2 inputs)



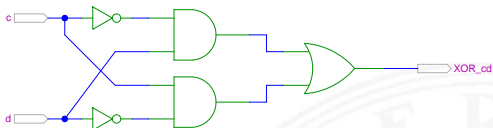
[HenHA] Hades Demo: 10-gates/00-gates/xor

XOR und drei Varianten der Realisierung

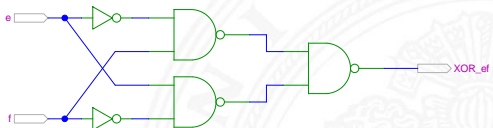
► Symbol



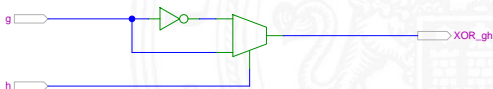
► AND-OR



► NAND-NAND



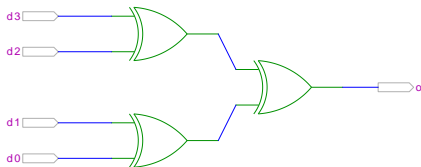
► mit Multiplexer



XOR zur Berechnung der Parität

- ▶ Parität, siehe „Codierung – Fehlererkennende Codes“

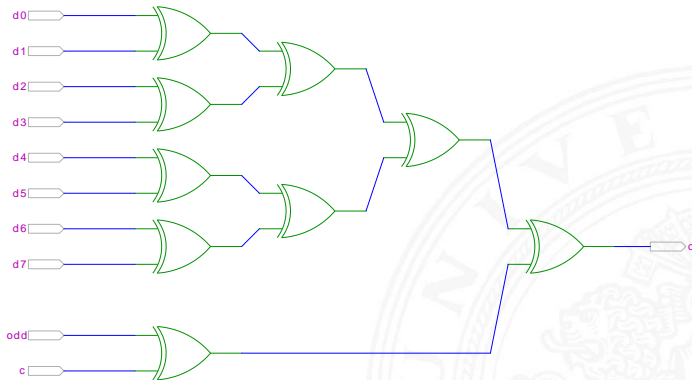
- ▶ 4-bit Parität: $d_3 \oplus d_2 \oplus d_1 \oplus d_0$



[HenHA] Hades Demo: 10-gates/12-parity/parity4

XOR zur Berechnung der Parität (cont.)

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



[HenHA] Hades Demo: 10-gates/12-parity/parity8

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

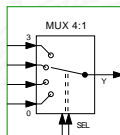
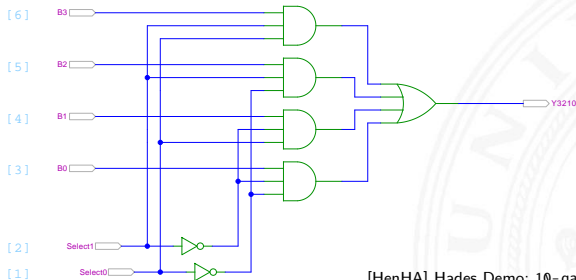
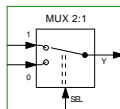
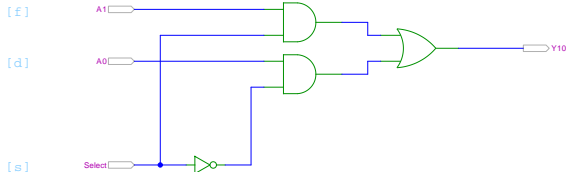
- ▶ wenn $s = 0$ hängt der Ausgangswert nur von a_0 ab
wenn $s = 1$ —"— a_1 ab

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

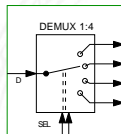
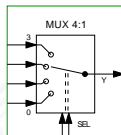
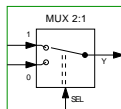
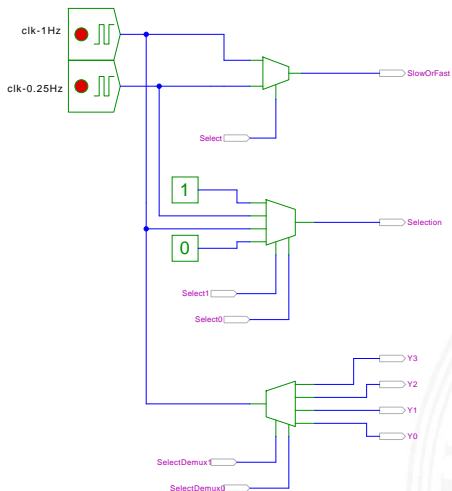
2:1 und 4:1 Multiplexer



[HenHA] Hades Demo: 10-gates/40-mux-demux/mux21-mux41

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

Multiplexer und Demultiplexer

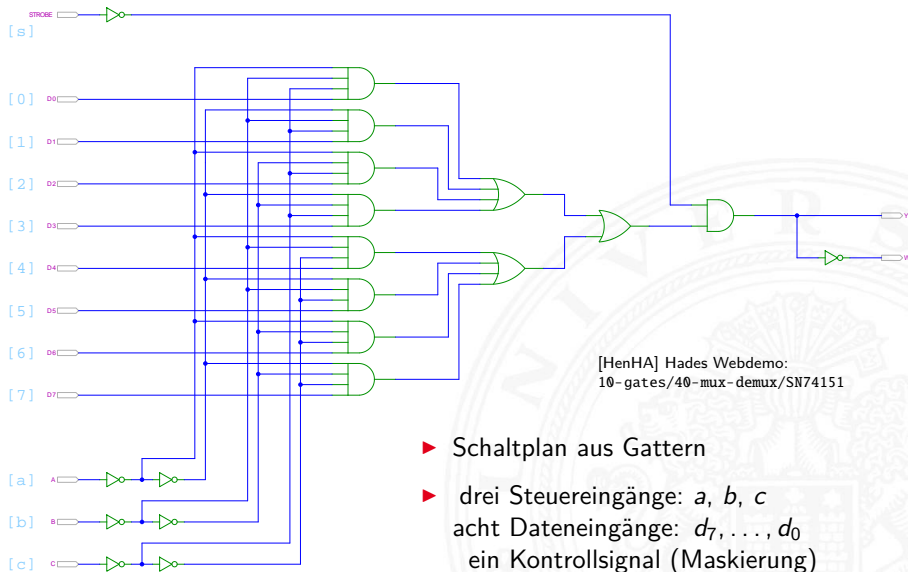


[HenHA] Hades Demo: 10-gates/40-mux-demux/mux-demux

8-bit Multiplexer: Integrierte Schaltung 74151

9.4.3 Schaltnetze - Logische Gatter - Multiplexer

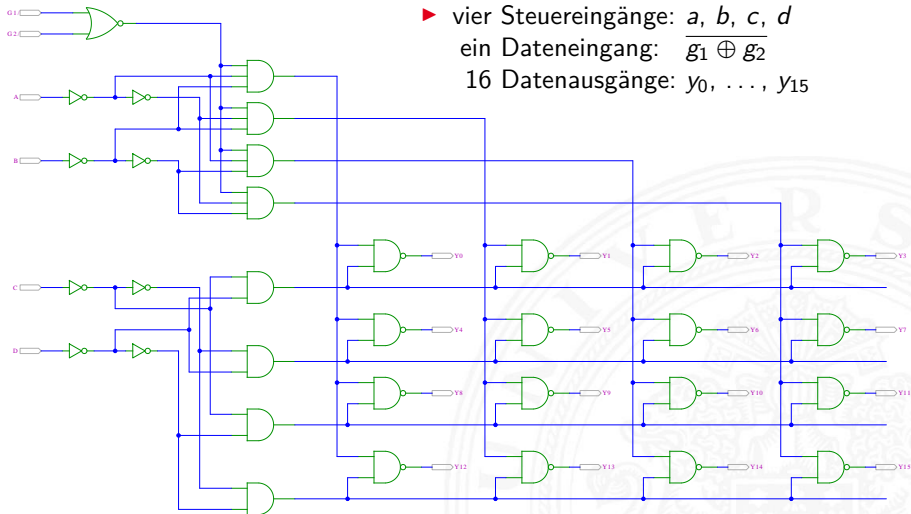
64-040 Rechnerstrukturen und Betriebssysteme



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

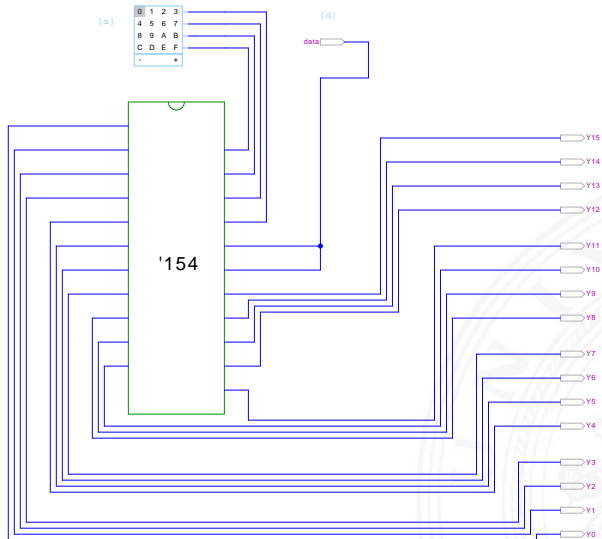
16-bit Demultiplexer: Integrierte Schaltung 74154

- vier Steuereingänge: a, b, c, d
- ein Dateneingang: $g_1 \oplus g_2$
- 16 Datenausgänge: y_0, \dots, y_{15}



[HenHA] Hades Demo: 10-gates/40-mux-demux/SN74154

16-bit Demultiplexer: 74154 als Adresdecoder



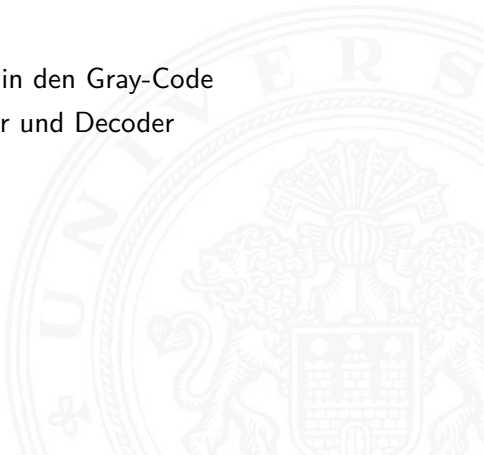
[HenHA] Hades Demo: 10-gates/40-mux-demux/demo74154



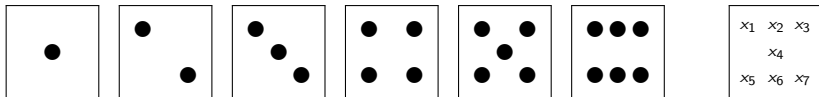
- ▶ Schaltungen mit mehreren Ausgängen
- ▶ Bündelminimierung der einzelnen Funktionen

ausgewählte typische Beispiele

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



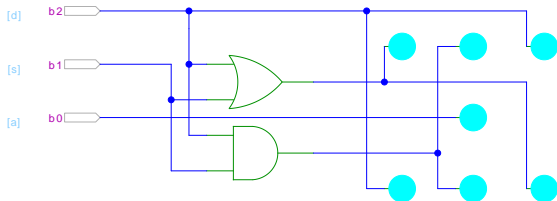
Visualisierung eines Würfels mit sieben LEDs



- ▶ Eingabewert von $0 \dots 6$
- ▶ Anzeige ein bis sechs Augen: eingeschaltet

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.)



[HenHA] Hades Demo: 10-gates/10-wuerfel/wuerfel

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

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

$$x_2 = x_6 = b_2 \wedge b_1$$

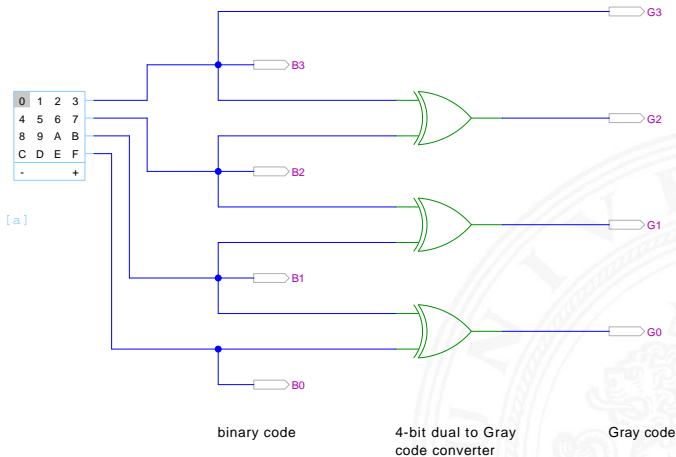
$$x_3 = x_5 = b_2$$

$$x_4 = b_0$$

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

Beispiel: Umwandlung vom Dualcode in den Graycode

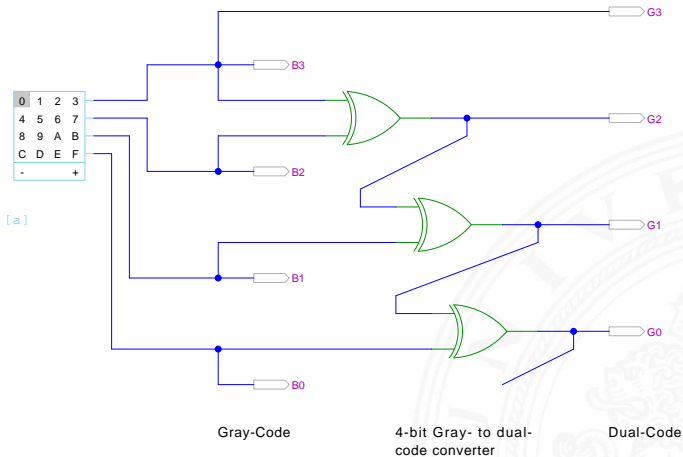
XOR benachbarter Bits



[HenHA] Hades Demo: 10-gates/15-graycode/dual2gray

Beispiel: Umwandlung vom Graycode in den Dualcode

XOR-Kette

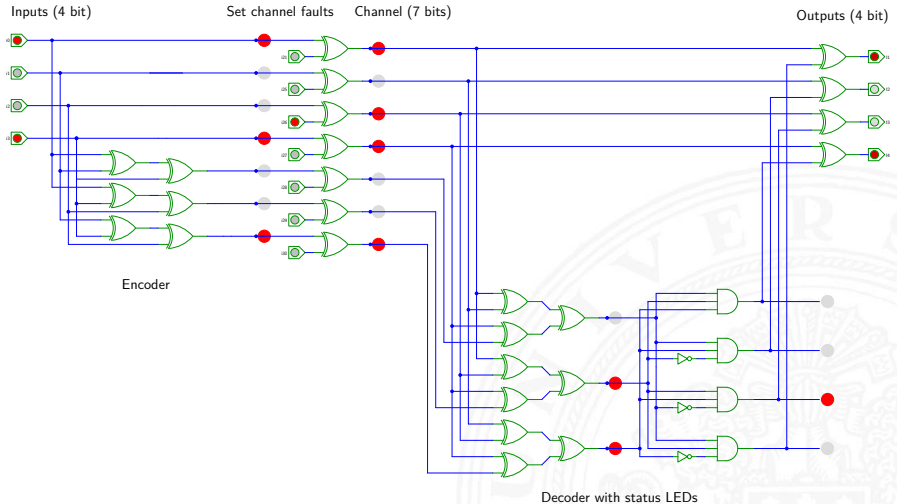


[HenHA] Hades Demo: [10-gates/15-graycode/gray2dual](#)



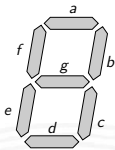
- ▶ Encoder linke Seite
 - ▶ vier Eingabebits
 - ▶ Hamming-Encoder erzeugt drei Paritätsbits
- ▶ Übertragungskanal Mitte
 - ▶ Übertragung von sieben Codebits
 - ▶ Einfügen von Übertragungsfehlern durch Invertieren von Codebits mit XOR-Gattern
- ▶ Decoder und Fehlerkorrektur rechte Seite
 - ▶ Decoder liest die empfangenen sieben Bits
 - ▶ Syndrom-Berechnung mit XOR-Gattern und Anzeige erkannter Fehler
 - ▶ Korrektur gekippter Bits rechts oben

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



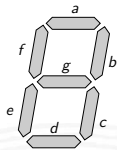
[HenHA] Hades Demo: 10-gates/50-hamming/hamming

- ▶ 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.



- ▶ Funktionen für Hex-Anzeige, 0...F

	0	1	2	3	4	5	6	7	8	9	A	b	C	d	E	F
a	1	0	1	1	0	1	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	0
c	1	1	0	1	1	1	1	1	1	1	1	1	0	1	0	0
d	1	0	1	1	0	1	1	0	1	1	0	1	1	1	1	0
e	1	0	1	0	0	0	1	0	1	0	1	1	1	1	1	1
f	1	0	0	0	1	1	1	1	1	1	1	1	0	0	1	1
g	0	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1



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

$a = 1011011111*****$
 $b = \text{usw.}$

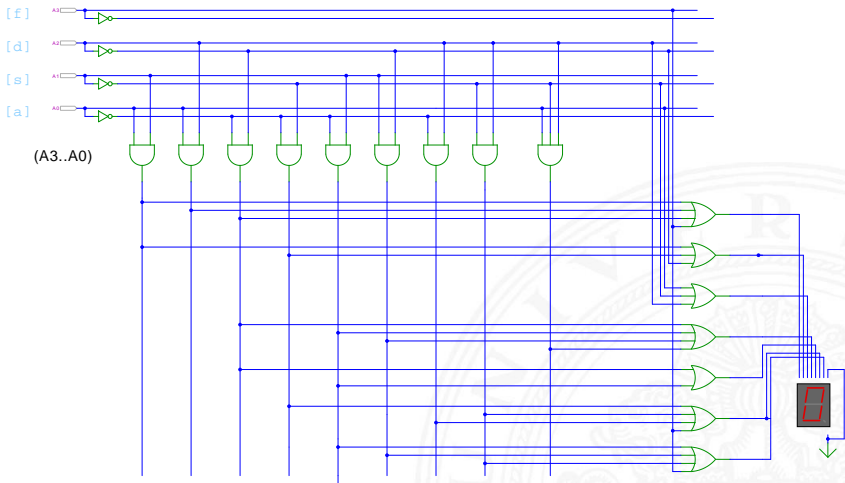


- ▶ zum Beispiel mit sieben KV-Diagrammen ...
- ▶ dabei versuchen, gemeinsame Terme zu finden und zu nutzen

Minimierung als Übungsaufgabe?

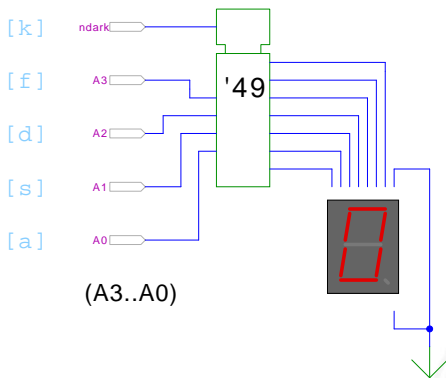
- ▶ nächste Folie zeigt Lösung aus Schiffmann, Schmitz [SS04]
- ▶ als mehrstufige Schaltung ist günstigere Lösung möglich
Knuth: *AoCP, Volume 4, Fascicle 0*, 7.1.2, Seite 112ff [Knu08]

Siebensegmentdecoder: Ziffern 0...9



[HenHA] Hades Demo: 10-gates/20-sevensegment/sevensegment

Siebensegmentdecoder: Integrierte Schaltung 7449



[HenHA] Hades Demo: 10-gates/20-sevenssegment/SN7449-demo

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

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

$$x_5 = x_2 \oplus x_3$$

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

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

$$x_8 = x_1 \oplus x_2$$

$$x_9 = x_4 \oplus x_5$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

D. E. Knuth: *AoCP, Volume 4, Fascicle 0*, Kap 7.1.2, Seite 113 [Knu08]



- ▶ Halb- und Volladdierer
- ▶ Addierertypen
 - ▶ Ripple-Carry
 - ▶ Carry-Lookahead

- ▶ Multiplizierer
- ▶ Quadratwurzel

- ▶ Barrel-Shifter
- ▶ ALU



- ▶ **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:** berechnet 1-bit Summe s und Übertrag c_o (*carry-out*) von zwei Eingangsbits a , b sowie Eingangsübertrag c_i (*carry-in*)

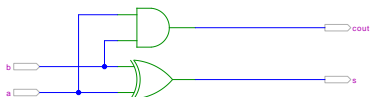
a	b	c_i	c_o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

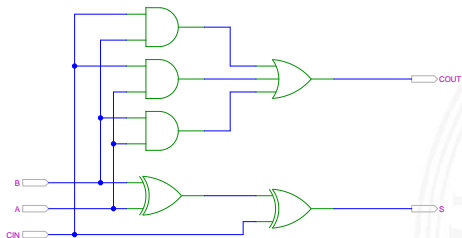
$$s = a \oplus b \oplus c_i$$

Schaltbilder für Halb- und Volladdierer

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



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



[HenHA] Hades Demo: 20-arithmetic/10-adders/halfadd- fulladd

► 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



Diverse gängige Alternativen für Addierer

- ▶ Ripple-Carry
 - ▶ lineare Struktur
 - + klein, einfach zu implementieren
 - langsam, Laufzeit $\mathcal{O}(n)$
- ▶ 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
- ...

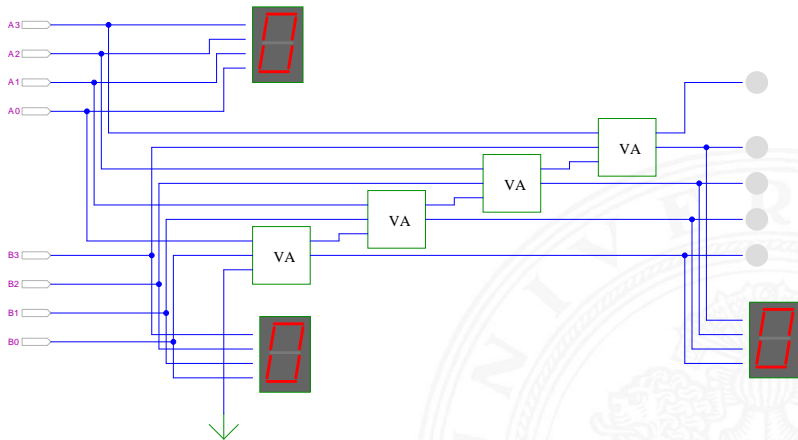
Ripple-Carry Adder

- ▶ 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 $\mathcal{O}(n)$

- ▶ Addierer in Prozessoren häufig im *kritischen Pfad*
- ▶ möglichst hohe Performanz ist essenziell
- ▶ Ripple-Carry in CMOS-Technologie bis ca. 10-bit geeignet
- ▶ bei größerer Wortbreite gibt es effizientere Schaltungen

- ▶ Überlauf-Erkennung: $c_o(n) \neq c_o(n - 1)$

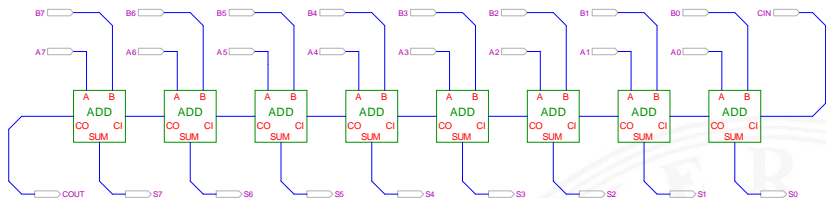
Ripple-Carry Adder: 4-bit



Schiffmann, Schmitz: *Technische Informatik I* [SS04]

Ripple-Carry Adder: Hades-Beispiel mit Verzögerungen

► Kaskade aus acht einzelnen Volladdierern



[HenHA] Hades Demo: 20-arithmetic/10-adders/ripple

- 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

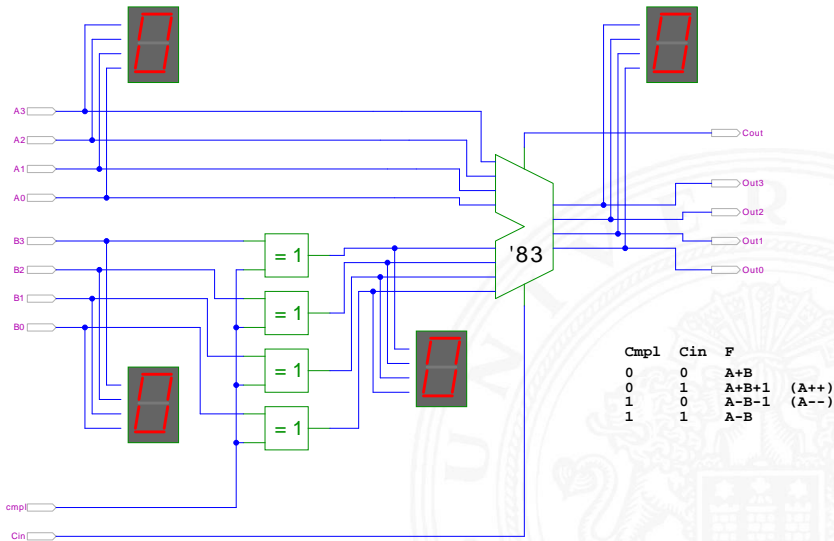
Zweierkomplement

- ▶ $(A - B)$ 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 $A + \overline{B} + 1 = A - B$

Subtrahierer: Beispiel 7483 – 4-bit Addierer



- ▶ Addierer in Prozessoren häufig im *kritischen Pfad*
 - ▶ möglichst hohe Performanz ist essenziell
- ⇒ 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
 - ▶ Übersicht beispielsweise auf www.aoki.ecei.tohoku.ac.jp/arith/mg/algorithm.html

Carry-Select Adder: Prinzip

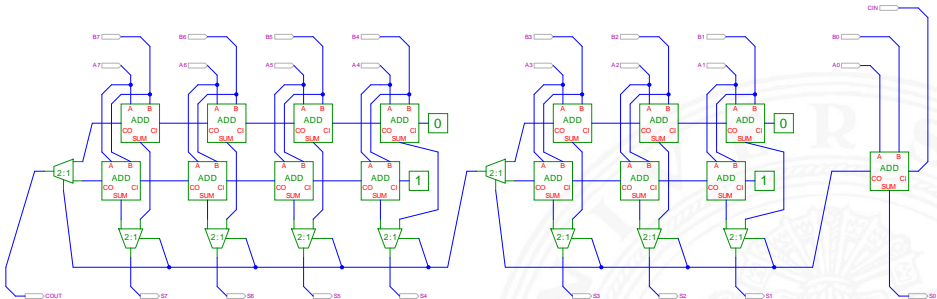
- Ripple-Carry Addierer muss auf die Überträge warten ($\mathcal{O}(n)$)
- ▶ Aufteilen des n -bit Addierers in mehrere Gruppen mit je m_i -bits
 - ▶ für jede Gruppe
 - ▶ jeweils zwei m_i -bit Addierer
 - ▶ einer rechnet mit $c_i = 0$ ($a + b$), der andere mit $c_i = 1$ ($a + b + 1$)
 - ▶ 2:1-Multiplexer 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

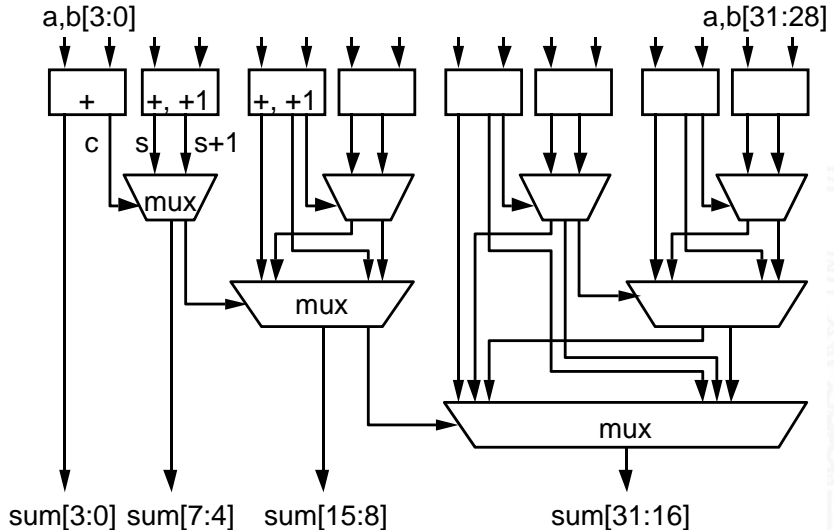
3-bit Carry-Select Adder block



[HenHA] Hades Demo: 20-arithmetic/20-carryselect/adder_carryselect

- ▶ 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

$$g_n = (a_n b_n)$$

„generate carry“

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

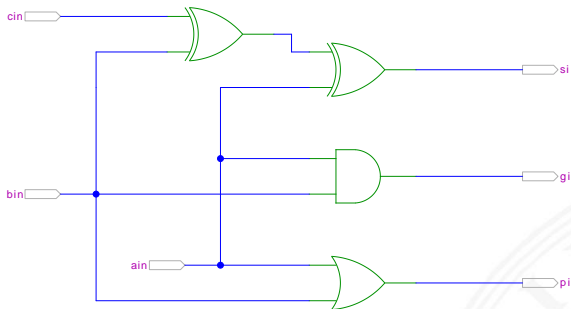
„propagate carry“

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

- ▶ *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

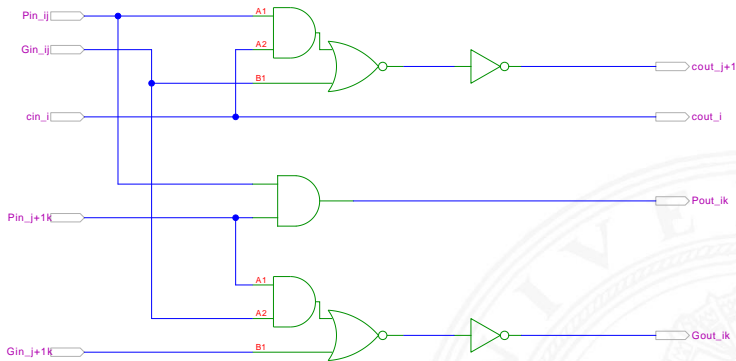
Carry-Lookahead Adder: SUM-Funktionsblock



[HenHA] Hades Demo: 20-arithmetic/30-cla/sum

- ▶ 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*
 $p_i = a_i \vee b_i$ – "– *propagate carry*

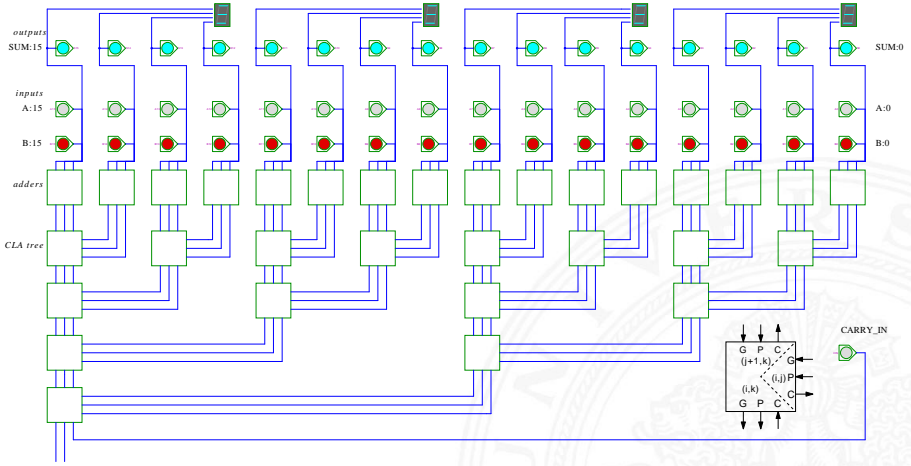
Carry-Lookahead Adder: CLA-Funktionsblock



[HenHA] Hades Demo: 20-arithmetic/30-cla/cla

- ▶ **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

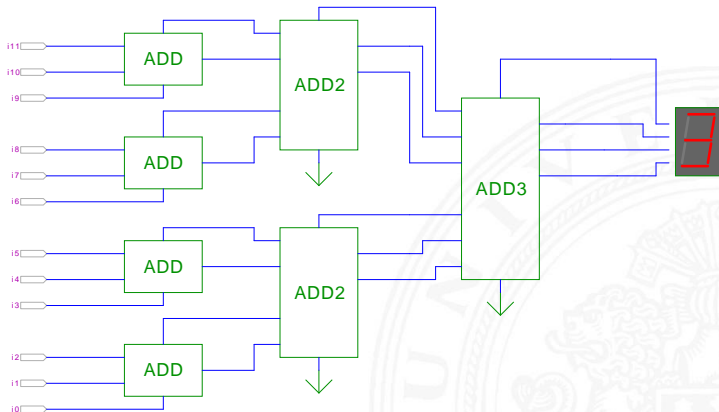
Carry-Lookahead Adder: 16-bit Addierer



[HenHA] Hades Demo: 20-arithmetic/30-cla/adder16

Addition mehrerer Operanden

- ▶ Addierer-Bäume
- ▶ Beispiel: Bitcount



[HenHA] Hades Demo: 20-arithmetic/80-bitcount/bitcount

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

- ▶ Ripple-Carry
 - ▶ Kaskade aus Volladdierern, einfach und billig
 - ▶ aber manchmal zu langsam, Verzögerung: $\mathcal{O}(n)$
- ▶ Carry-Select Prinzip
 - ▶ Verzögerung $\mathcal{O}(\sqrt{n})$
- ▶ Carry-Lookahead Prinzip
 - ▶ Verzögerung $\mathcal{O}(\ln n)$

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

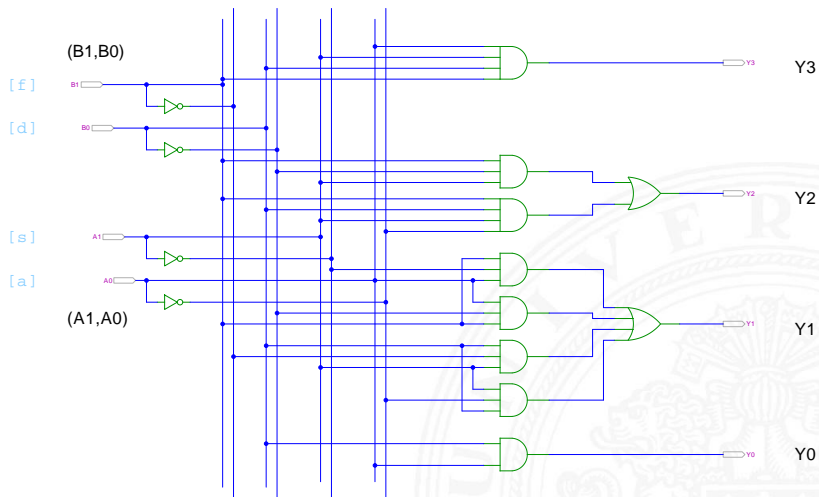


- ▶ Teilprodukte als UND-Verknüpfung des Multiplikators mit je einem Bit des Multiplikanden
- ▶ Aufaddieren der Teilprodukte mit Addierern
- ▶ Realisierung als Schaltnetz erfordert:
 - n^2 UND-Gatter (bitweise eigentliche Multiplikation)
 - n^2 Volladdierer (Aufaddieren der Teilprodukte)
- ▶ abschließend 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

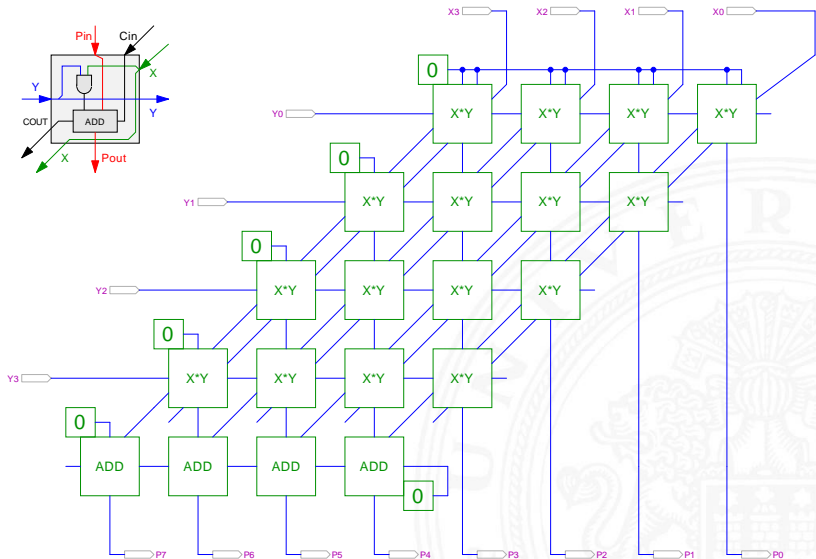
9.7.2 Schaltnetze - Schaltnetze für Logische und Arithmetische Operationen - Multiplizierer 64-040 Rechnerstrukturen und Betriebssysteme



[HenHA] Hades Demo: 10-gates/13-mult2x2/mult2x2

4x4-bit Multiplizierer – Array

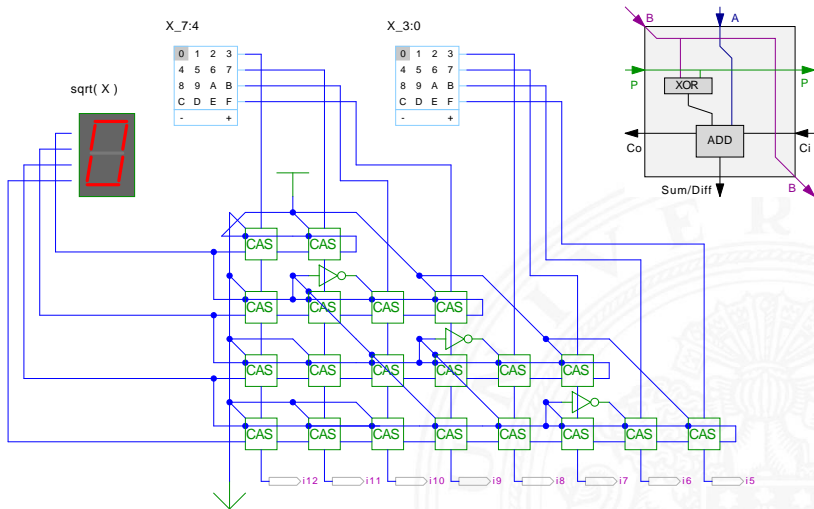
9.7.2 Schaltnetze - Schaltnetze für Logische und Arithmetische Operationen - Multiplizierer 64-040 Rechnerstrukturen und Betriebssysteme



[HenHA] Hades Demo: 20-arithmetic/60-mult/mult4x4

4x4-bit Quadratwurzel

9.7.2 Schaltnetze - Schaltnetze für Logische und Arithmetische Operationen - Multiplizierer 64-040 Rechnerstrukturen und Betriebssysteme



[HenHA] Hades Demo: 20-arithmetic/90-sqrt/sqrt4



weitere wichtige Themen aus Zeitgründen nicht behandelt

- ▶ *Carry-Save Adder* zur Summation der Teilprodukte
- ▶ *Booth-Codierung* (effiziente Multiplikation)
- ▶ Multiplikation von Zweierkomplementzahlen
- ▶ Multiplikation von Gleitkommazahlen

- ▶ *CORDIC-Algorithmen*
 - ▶ Multiplikation, Division
 - ▶ iterative Berechnung höherer Funktionen: Exponentialfunktion, Logarithmus, trigonometrische Funktionen

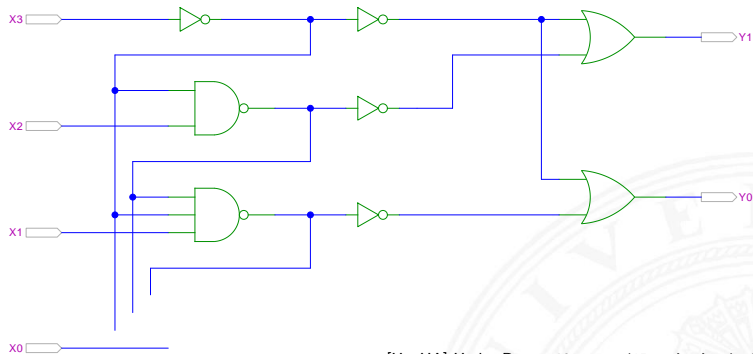
- ▶ bei Interesse: Literatur anschauen [Omo94, Kor01, Spa76]

- ▶ 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 $\Rightarrow x_0$ kann entfallen

4:2 Prioritätsencoder



[HenHA] Hades Demo: 10-gates/45-priority/priority42

- ▶ zweistufige Realisierung (Inverter ignoriert)
- ▶ aktive höhere Stufe blockiert alle niedrigeren Stufen

4:2 Prioritätsencoder: Kaskadierung

9.7.3 Schaltnetze

Schaltnetze

für

Logische

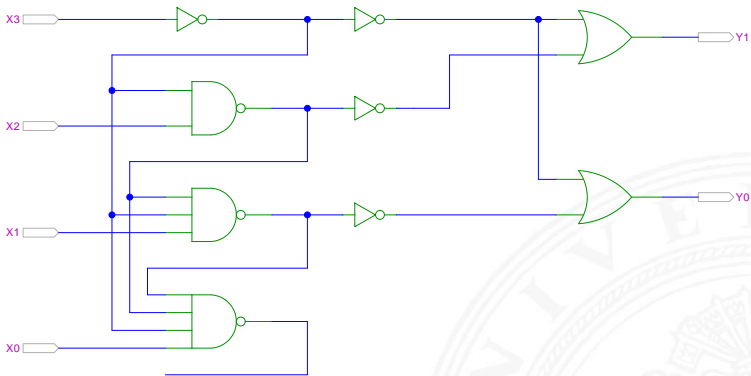
und

Arithmetische

Operationen

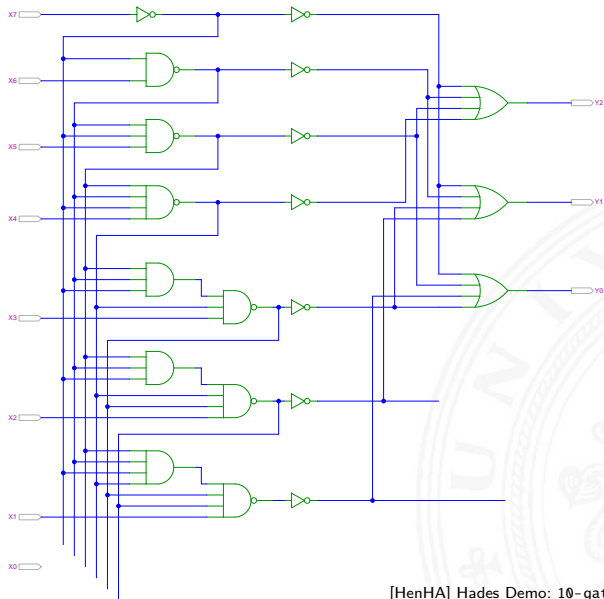
Prioritätsencoder

64-040 Rechnerstrukturen und Betriebssysteme





8:3 Prioritätsencoder

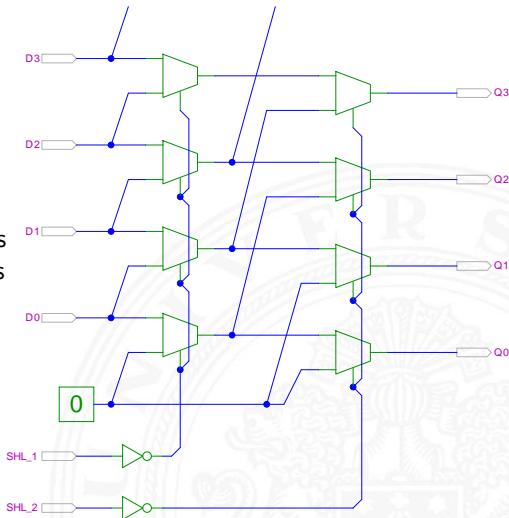


[HenHA] Hades Demo: 10-gates/45-priority/priority83

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

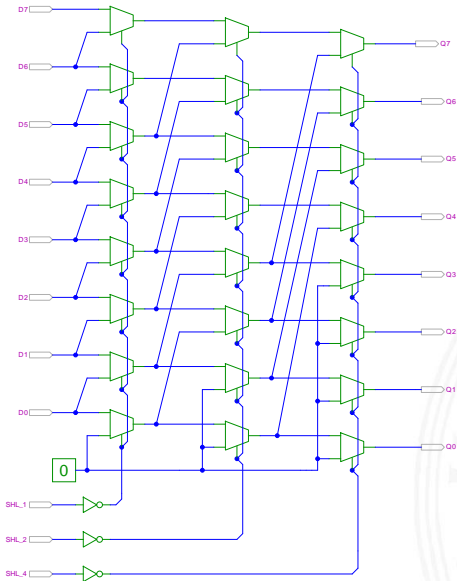
9.7.4 Schaltnetze - Schaltnetze für Logische und Arithmetische Operationen - Barrel-Shifter 64-040 Rechnerstrukturen und Betriebssysteme

- ▶ 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

9.7.4 Schaltnetze - Schaltnetze für Logische und Arithmetische Operationen - Barrel-Shifter 64-040 Rechnerstrukturen und Betriebssysteme



[HenHA] Hades Webdemo:
10-gates/60-barrel/shifter8

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 hineinschieben
- + alle Operationen typischerweise in einem Takt realisierbar
- Problem: Hardwareaufwand bei großen Wortbreiten und beliebigem Schiebe-/Rotate-Argument

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

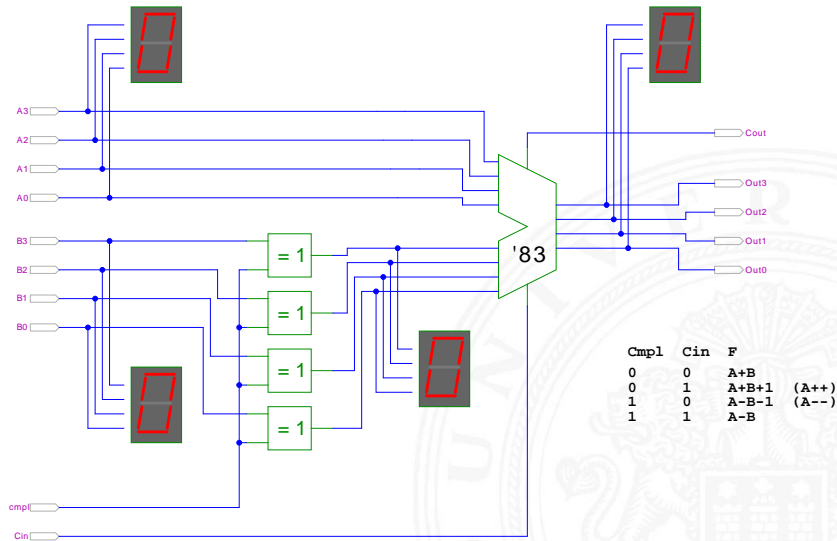
- ▶ Addition und Subtraktion 2-Komplement
- ▶ bitweise logische Operationen Negation, UND, ODER, XOR
- ▶ Schiebeoperationen shift, rotate
- ▶ evtl. Multiplikation

- ▶ Integer-Division selten verfügbar (separates Rechenwerk)

- ▶ 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 $(A - B) = A + \bar{B} + 1$
- ▶ ggf. auch Inkrement ($A + 1$) und Dekrement ($A - 1$)

- ▶ folgende Folien: 7483 ist IC mit 4-bit Addierer

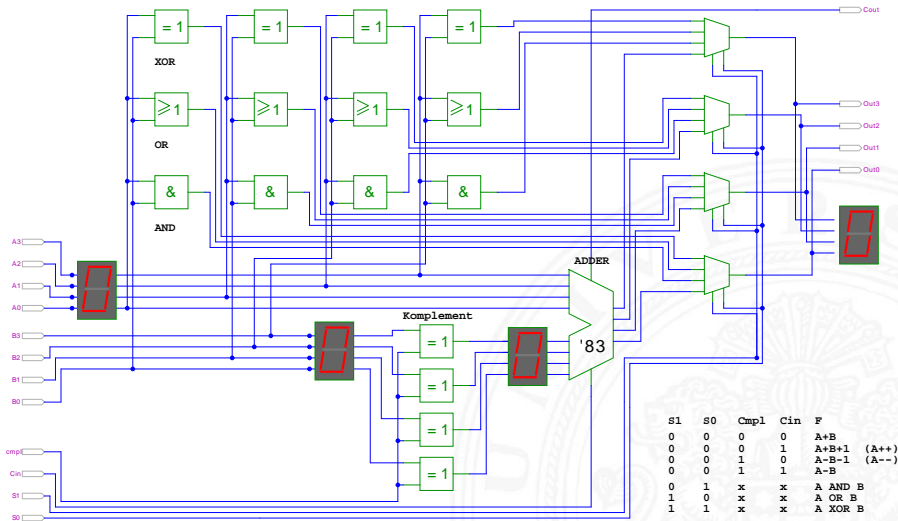
ALU: Addierer und Subtrahierer



ALU: Addierer und bitweise Operationen

9.8 Schaltnetze - ALU (Arithmetisch-Logische Einheit)

64-040 Rechnerstrukturen und Betriebssysteme



Schiffmann, Schmitz: Technische Informatik I [SS04]

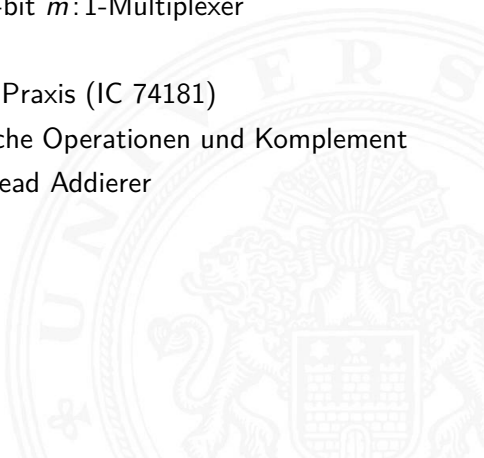


vorige Folie zeigt die „triviale“ Realisierung einer ALU

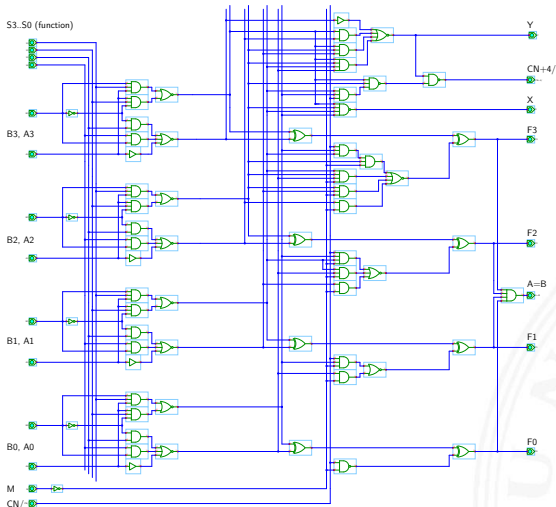
- ▶ mehrere parallele Rechenwerke für die m einzelnen Operationen
 n -bit Addierer, n -bit Komplement, n -bit OR usw.
- ▶ Auswahl des Resultats über n -bit $m:1$ -Multiplexer

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



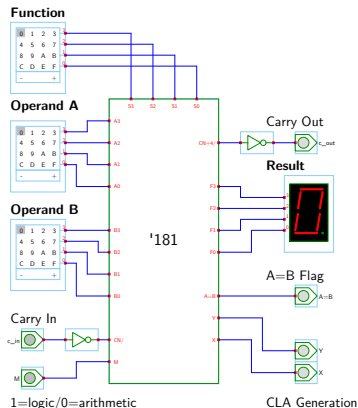
selection	logic functions	arithmetic functions
S3 S2 S1 S0	M = 1	M = 0 Cn = 1 (no carry)
0 0 0 0	F = !A	F = A
0 0 0 1	F = !(A or B)	F = A or B
0 0 1 0	F = !A and B	F = A or !B
0 0 1 1	F = !A and !B	F = -1
0 1 0 0	F = 0	F = A + (A and !B)
0 1 0 1	F = !B	F = (A or B) + (A and !B)
0 1 1 0	F = A xor B	F = A - B - 1
0 1 1 1	F = A and !B	F = (A and !B) - 1
1 0 0 0	F = !A or B	F = A + (A and B)
1 0 0 1	F = A xor B	F = A + B
1 0 1 0	F = B	F = (A or !B) + (A and B)
1 0 1 1	F = A and B	F = (A and B) - 1
1 1 0 0	F = 1	F = A + A
1 1 0 1	F = A or !B	F = (A or B) + A
1 1 1 0	F = A or B	F = (A or !B) + A
1 1 1 1	F = A	F = A - 1
		F + 1 Cn = 0 (carry in)

[HenHA] Hades Demo: 20-arithmetic/50-74181/SN74181

ALU: 74181 – Funktionstabelle

9.8 Schaltnetze - ALU (Arithmetisch-Logische Einheit)

64-040 Rechnerstrukturen und Betriebssysteme



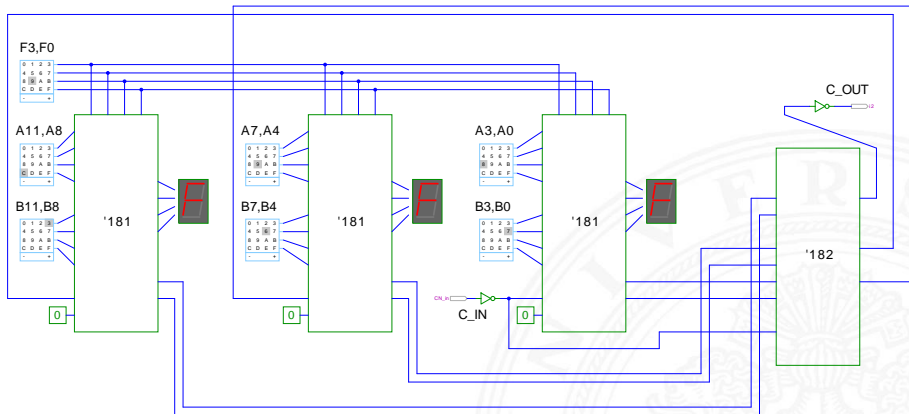
selection	logic functions	arithmetic functions
S3 S2 S1 S0	M = 1	M = 0 Cn = 1 (no carry)
0 0 0 0	F = !A	F = A
0 0 0 1	F = !(A or B)	F = A or B
0 0 1 0	F = !A and B	F = A or !B
0 0 1 1	F = !A and B	F = -1
0 1 0 0	F = 0	F = A + (A and !B)
0 1 0 1	F = !B	F = (A or B) + (A and !B)
0 1 1 0	F = A xor B	F = A - B - 1
0 1 1 1	F = A and !B	F = (A and !B) - 1
1 0 0 0	F = !A or B	F = A + (A and B)
1 0 0 1	F = A xnor B	F = A + B
1 0 1 0	F = B	F = (A or !B) + (A and B)
1 0 1 1	F = A and B	F = (A and B) - 1
1 1 0 0	F = 1	F = A + A
1 1 0 1	F = A or !B	F = (A or B) + A
1 1 1 0	F = A or B	F = (A or !B) + A
1 1 1 1	F = A	F = A - 1
		F + 1 Cn = 0 (carry in)

[HenHA] Hades Demo: 20-arithmetic/50-74181/demo-74181-ALU



ALU: 74181 und 74182 CLA

12-bit ALU mit Carry-Lookahead Generator 74182



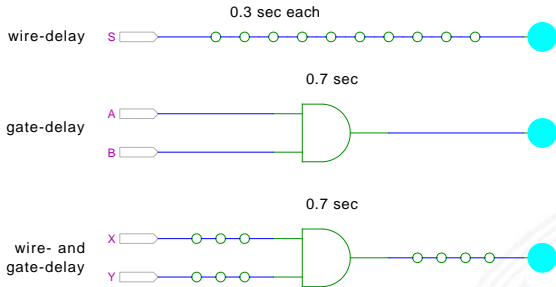
[HenHA] Hades Demo: 20-arithmetic/50-74181/demo-74182-ALU-CLA

Wie wird das Zeitverhalten eines Schaltnetzes modelliert?

Gängige Abstraktionsebenen mit zunehmendem Detaillierungsgrad

1. algebraische Ausdrücke: keine zeitliche Abhängigkeit
2. „fundamentales Modell“: Einheitsverzögerung des algebraischen Ausdrucks um eine Zeit τ
3. individuelle Gatterverzögerungen
 - ▶ mehrere Modelle, unterschiedlich detailliert
 - ▶ Abstraktion elektrischer Eigenschaften
4. Gatterverzögerungen + Leitungslaufzeiten (geschätzt, berechnet)
5. Differentialgleichungen für Spannungen und Ströme (verschiedene „Ersatzmodelle“)

Gatterverzögerung vs. Leitungslaufzeiten



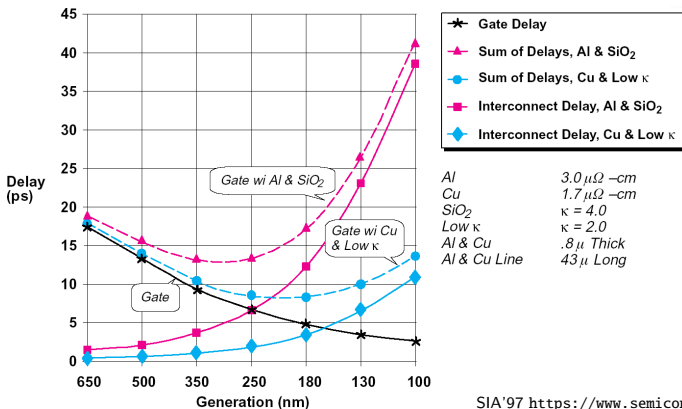
[HenHA] Hades Demo: 12-gatedelay/10-delaydemo/gate-vs-wire-delay

- ▶ früher: Gatterverzögerungen \gg Leitungslaufzeiten
- ▶ Schaltungen modelliert durch Gatterlaufzeiten
- ▶ aktuelle „Submicron“-Halbleitertechnologie: Leitungslaufzeiten \gg Gatterverzögerungen

Gatterverzögerung vs. Leitungslaufzeiten (cont.)

▶ Leitungslaufzeiten

- ▶ lokale Leitungen: schneller (weil Strukturen kleiner)
- ▶ globale Leitungen: langsamer
- nicht mehr alle Punkte des Chips in einem Taktzyklus erreichbar

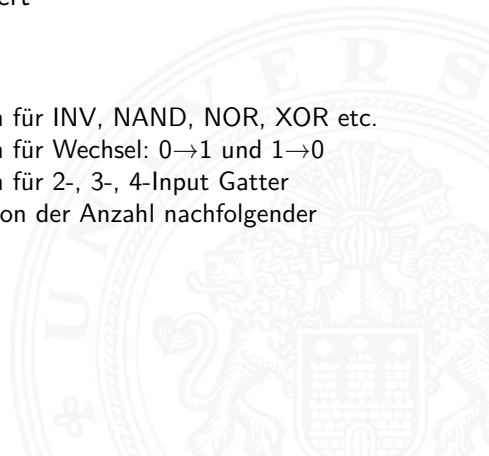


SIA '97 <https://www.semiconductors.org>



- ▶ alle folgenden Schaltungsbeispiele werden mit Gatterverzögerungen modelliert (einfacher Handhabbar)
- ▶ Gatterlaufzeiten als Vielfache einer Grundverzögerung (τ)
- ▶ aber Leitungslaufzeiten ignoriert

- ▶ mögliche Verfeinerungen
 - ▶ gatterabhängige Schaltzeiten für INV, NAND, NOR, XOR etc.
 - ▶ unterschiedliche Schaltzeiten für Wechsel: $0 \rightarrow 1$ und $1 \rightarrow 0$
 - ▶ unterschiedliche Schaltzeiten für 2-, 3-, 4-Input Gatter
 - ▶ Schaltzeiten sind abhängig von der Anzahl nachfolgender Eingänge (engl. *fanout*)



Exkurs: Lichtgeschwindigkeit und Taktraten

- ▶ Lichtgeschwindigkeit im Vakuum: $c \approx 300\,000 \text{ km/sec}$
 $\approx 30 \text{ cm/ns}$
 - ▶ in Metallen und Halbleitern langsamer: $c \approx 20 \text{ cm/ns}$
- ⇒ bei 1 Gigahertz Takt: Ausbreitung um ca. 20 Zentimeter

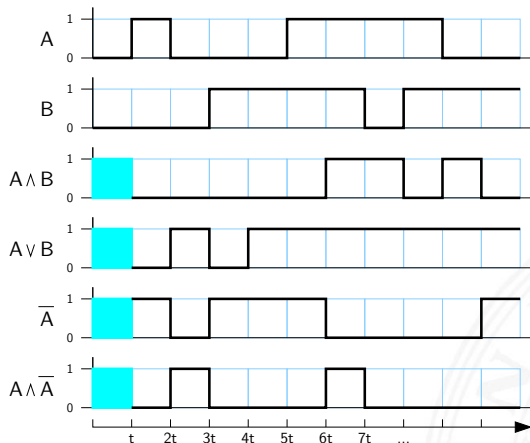
Abschätzungen:

- ▶ Prozessor: ca. 3 cm Diagonale < 10 GHz Taktrate
 - ▶ Platine: ca. 20 cm Kantenlänge < 1 GHz Takt
- ⇒ prinzipiell kann (schon heute) ein Signal innerhalb eines Takts nicht von einer Ecke des ICs zur Anderen gelangen



- ▶ **Impulsdiagramm** (engl. *waveform*): Darstellung der logischen Werte einer Schaltfunktion als Funktion der Zeit
- ▶ als Abstraktion des tatsächlichen Verlaufs
- ▶ Zeit läuft von links nach rechts
- ▶ Schaltfunktion(en): von oben nach unten aufgelistet
- ▶ Vergleichbar den Messwerten am Oszilloskop (analoge Werte) bzw. den Messwerten am Logic-State-Analyzer (digitale Werte)
- ▶ ggf. Darstellung mehrerer logischer Werte (z.B. 0,1,Z,U,X)

Impulsdigramm: Beispiel

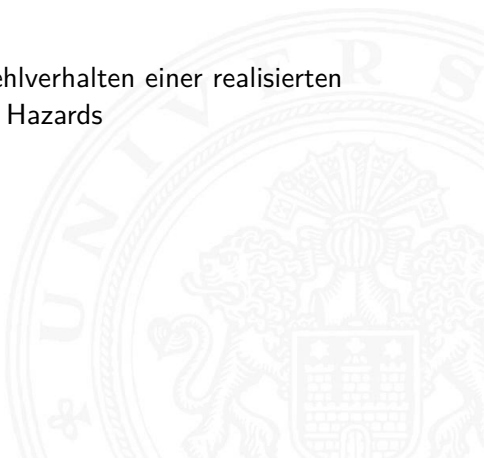


- ▶ im Beispiel jeweils eine „Zeiteinheit“ Verzögerung für jede einzelne logische Operation
- ▶ Ergebnis einer Operation nur, wenn die Eingaben definiert sind
- ▶ im ersten Zeitschritt noch undefinierte Werte



- ▶ **Hazard:** die Eigenschaft einer Schaltfunktion, bei bestimmten Kombinationen der individuellen Verzögerungen ihrer Verknüpfungsglieder ein Fehlverhalten zu zeigen
- ▶ engl. auch *Glitch*

- ▶ **Hazardfehler:** das aktuelle Fehlverhalten einer realisierten Schaltfunktion aufgrund eines Hazards



nach der Erscheinungsform am Ausgang

- ▶ **statisch**: der Ausgangswert soll unverändert sein, es tritt aber ein Wechsel auf
- ▶ **dynamisch**: der Ausgangswert soll (einmal) wechseln, es tritt aber ein mehrfacher Wechsel auf

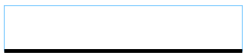
nach den Eingangsbedingungen, unter denen der Hazard auftritt

- ▶ **Strukturhazard**: bedingt durch die Struktur der Schaltung, auch bei Umschalten eines einzigen Eingangswertes
- ▶ **Funktionshazard**: bedingt durch die Funktion der Schaltung

Hazards: statisch vs. dynamisch

erwarteter Signalverlauf

Verlauf mit Hazard



statischer 0-Hazard



statischer 1-Hazard



dynamischer 0-Hazard



dynamischer 1-Hazard

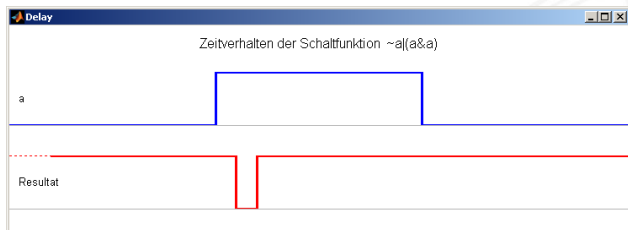
! Begriffsbildung in der Literatur nicht einheitlich:

0 / 1 als „richtiger Wert“

–“– „fehlerhafter Wert“

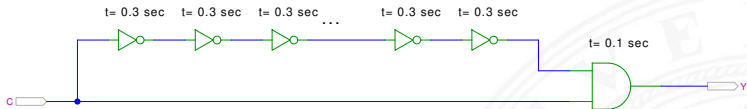
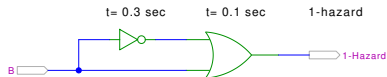
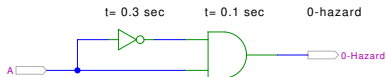
- ▶ 0-Hazard wenn der Wert 0 ausgegeben werden soll, zwischenzeitlich aber 1 erscheint (und umgekehrt)
- ▶ statisch oder dynamisch (dann auch 1-0, bzw. 0-1 Hazard)
- ▶ es können natürlich auch mehrfache Hazards auftreten

- ▶ **Strukturhazard** wird durch die gewählte Struktur der Schaltung verursacht
- ▶ auch, wenn sich nur eine Variable ändert
- ▶ Beispiel: $f(a) = \bar{a} \vee (a \wedge a)$
 \bar{a} schaltet schneller ab, als $(a \wedge a)$ einschaltet



- ▶ Hazard kann durch Modifikation der Schaltung beseitigt werden
im Beispiel mit: $f(a) = 1$

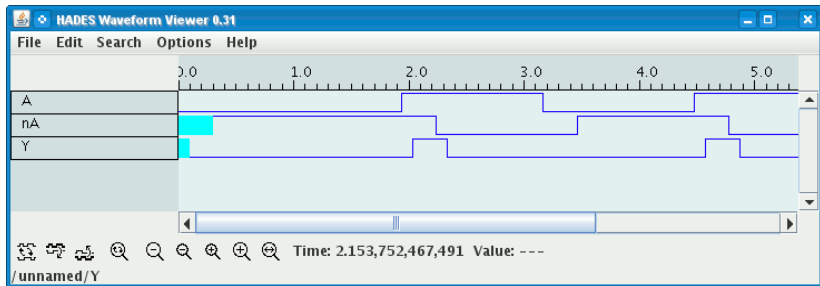
Strukturhazards: Beispiele



[HenHA] Hades Demo: 12-gatedelay/30-hazards/padding

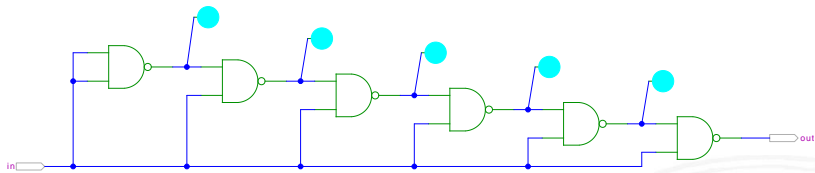
- ▶ logische Funktion ist $(a \wedge \bar{a}) = 0$ bzw. $(a \vee \bar{a}) = 1$
 - ▶ aber ein Eingang jeweils durch Inverter verzögert
- ⇒ kurzer Impuls beim Umschalten von $0 \rightarrow 1$ bzw. $1 \rightarrow 0$

Strukturhazards: Beispiele (cont.)



- ▶ Schaltung $(a \wedge \bar{a}) = 0$ erzeugt (statischen-0) Hazard
- ▶ Länge des Impulses abhängig von Verzögerung im Inverter
- ▶ Kette von Invertern erlaubt Einstellung der Pulslänge

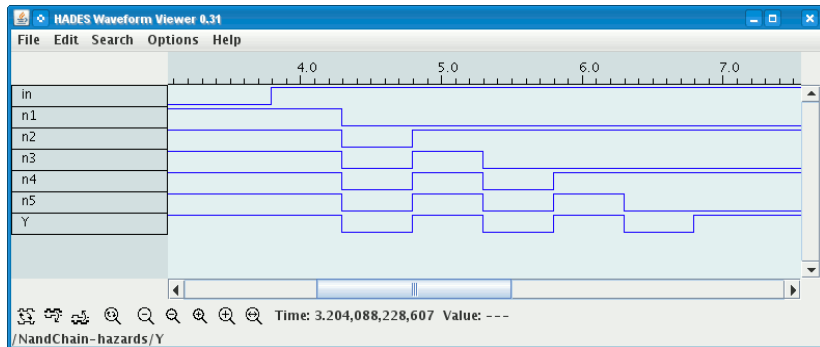
Strukturhazards extrem: NAND-Kette



[HenHA] Hades Demo: 12-gatedelay/30-hazards/nandchain

- ▶ alle NAND-Gatter an Eingang in angeschlossen
- ▶ $in = 0$ erzwingt $y_i = 1$
- ▶ Übergang in von 0 auf 1 startet Folge von Hazards

Strukturhazards extrem: NAND-Kette (cont.)



- ▶ Schaltung erzeugt Folge von (statischen-1) Hazards
- ▶ Anzahl der Impulse abhängig von Anzahl der Gatter

Strukturhazards im KV-Diagramm

$x_2 \backslash x_1 x_0$	00	01	11	10
0	0	0	1	0
1	1	1	1	0

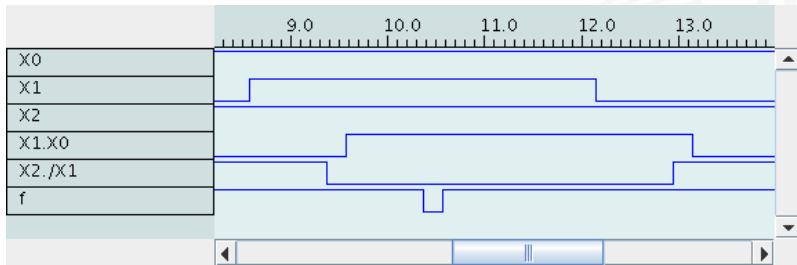
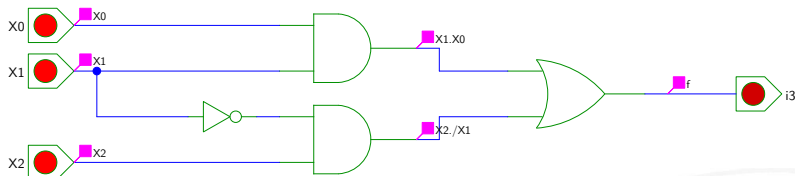
$x_2 \backslash x_1 x_0$	00	01	11	10
0	0	0	1	0
1	1	1	1	0

- ▶ Funktion $f = (x_2 \bar{x}_1) \vee (x_1 x_0)$
- ▶ realisiert in disjunktiver Form mit 2 Schleifen

Strukturhazard beim Übergang von $(x_2 \bar{x}_1 x_0)$ nach $(x_2 x_1 x_0)$

- ▶ Gatter $(x_2 \bar{x}_1)$ schaltet ab, Gatter $(x_1 x_0)$ schaltet ein
- ▶ Ausgang evtl. kurz 0, abhängig von Verzögerungen

Strukturhazards im KV-Diagramm (cont.)



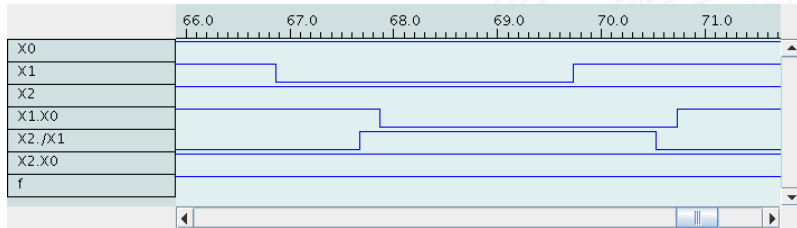
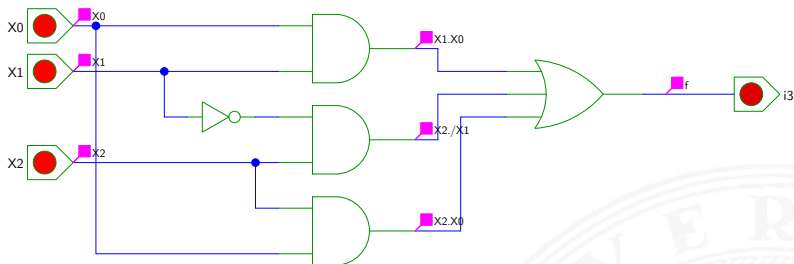
Strukturhazards beseitigen

$x_2 \backslash x_1 x_0$	00	01	11	10
0	0	0	1	0
1	1	1	1	0

$x_2 \backslash x_1 x_0$	00	01	11	10
0	0	0	1	0
1	1	1	1	0

- ▶ Funktion $f = (x_2 \bar{x}_1) \vee (x_1 x_0)$
- ▶ realisiert in disjunktiver Form mit **3 Schleifen**
 $f = (x_2 \bar{x}_1) \vee (x_1 x_0) \vee (x_2 x_0)$
- + Strukturhazard durch zusätzliche Schleife beseitigt
- aber höhere Hardwarekosten als bei minimierter Realisierung

Strukturhazards beseitigen (cont.)



Hazards: Funktionshazard

- ▶ **Funktionshazard** kann bei gleichzeitigem Wechsel mehrerer Eingangswerte als **Eigenschaft der Schaltfunktion** entstehen
- ▶ Problem: Gleichzeitigkeit an Eingängen
- ⇒ Funktionshazard kann nicht durch strukturelle Maßnahmen verhindert werden

- ▶ Beispiel: Übergang von $(x_2 \bar{x}_1 x_0)$ nach $(\bar{x}_2 x_1 x_0)$

$x_2 \backslash x_1 x_0$	00	01	11	10
0	0	0	1	0
1	1	1	1	0

$x_2 \backslash x_1 x_0$	00	01	11	10
0	0	0	1	0
1	1	1	1	0

- [Knu08] D.E. Knuth: *The Art of Computer Programming, Volume 4, Fascicle 0, Introduction to Combinatorial Algorithms and Boolean Functions.*
Addison-Wesley Professional, 2008. ISBN 978-0-321-53496-5
- [Knu09] D.E. Knuth: *The Art of Computer Programming, Volume 4, Fascicle 1, Bitwise Tricks & Techniques; Binary Decision Diagrams.*
Addison-Wesley Professional, 2009. ISBN 978-0-321-58050-4
- [SS04] W. Schiffmann, R. Schmitz: *Technische Informatik 1 – Grundlagen der digitalen Elektronik.*
5. Auflage, Springer-Verlag, 2004. ISBN 978-3-540-40418-7
- [Weg87] I. Wegener: *The Complexity of Boolean Functions.*
John Wiley & Sons, 1987. ISBN 3-519-02107-2.
ls2-www.cs.uni-dortmund.de/monographs/bluebook

- [BM08] B. Becker, P. Molitor: *Technische Informatik – eine einführende Darstellung*. 2. Auflage, Oldenbourg, 2008. ISBN 978-3-486-58650-3
- [Fur00] S. Furber: *ARM System-on-Chip Architecture*. 2nd edition, Pearson Education Limited, 2000. ISBN 978-0-201-67519-1
- [Omo94] A.R. Omondi: *Computer Arithmetic Systems – Algorithms, Architecture and Implementations*. Prentice-Hall International, 1994. ISBN 0-13-334301-4
- [Kor01] I. Koren: *Computer Arithmetic Algorithms*. 2nd edition, CRC Press, 2001. ISBN 978-1-568-81160-4. www.ecs.umass.edu/ece/koren/arith
- [Spa76] O. Spaniol: *Arithmetik in Rechenanlagen*. B. G. Teubner, 1976. ISBN 3-519-02332-6

- [Hei05] K. von der Heide: *Vorlesung: Technische Informatik 1 — interaktives Skript*. Uni. Hamburg, FB Informatik, 2005. tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1
- [HenHA] N. Hendrich: *HADES — HAMBURG DEsign System*. Universität Hamburg, FB Informatik, Lehrmaterial. tams.informatik.uni-hamburg.de/applets/hades/webdemos
- [HenKV] N. Hendrich: *KV-Diagram Simulation*. Universität Hamburg, FB Informatik, Lehrmaterial. tams.informatik.uni-hamburg.de/applets/kvd
- [Kor16] Laszlo Korte: *TAMS Tools for eLearning*. Universität Hamburg, FB Informatik, 2016, BSc Thesis. tams.informatik.uni-hamburg.de/research/software/tams-tools
- [Laz] J. Lazzaro: *Chipmunk design tools (AnaLog, DigLog)*. UC Berkeley, Berkeley, CA. john-lazzaro.github.io/chipmunk