MIXED SIGNAL
MSC
CIRCUIT DESIGN

Technische Universität Berlin

Grundlagen Digitalentwurf

Einstein-Prof. Dr.-Ing. Friedel Gerfers

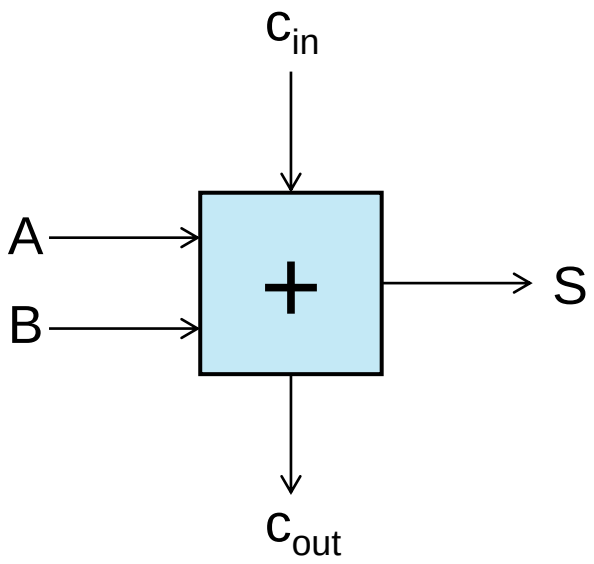
Ziele

Nach dieser VL sollten Sie in der Lage sein:

- Verständnis der grundlegenden logischen Gatter
- Die Wahrheitstabelle einer Boolesche Funktion aufzustellen
- Für eine gegebene Wahrheitstabelle, die disjunktive oder konjunktive Normalform aufzustellen
- Zu beweisen, dass ein Operatorensatz vollständig ist
- Eine arithmetische logische Einheit zu erweitern
- Die Funktionsweise eines CLA Addierers zu erklären
- Das Verhalten folgender Speicherelemente zu erklären: SR-Latch, D-Latch, Flip-Flop, Registersatz

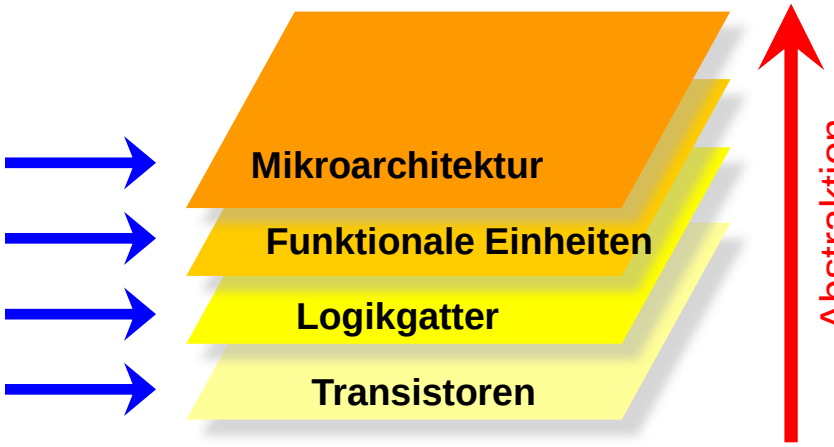
Digitalentwurf

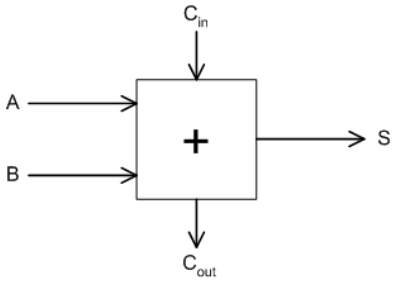
- Digitale Hardware wird mithilfe des „Digitalentwurfs-Flow“ konstruiert
  - Hardware-Beschreibungssprache (Hardware Description Language, HDL)
- Digital Schaltungen verarbeiten 2 diskrete Signalzustände: logisch 0 und logisch 1
- Beispiel: 1-bit Volladdierer (*Full Adder*, *FA*)

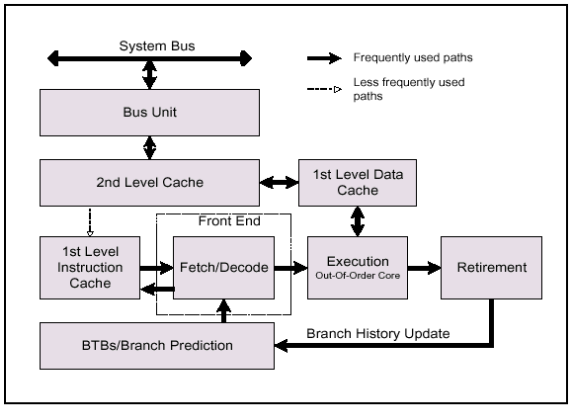


Hierarchischer Entwurf

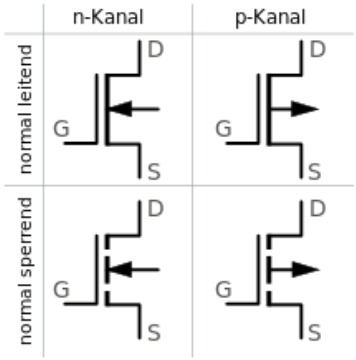
- Logikentwurf ist die Vernetzung logischer Gatter
- Zuhilfenahme von „höheren Sprachen“, um auf einer abstrakteren Ebene als Logikgatter entwerfen zu können

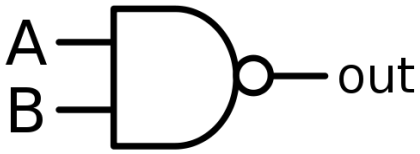




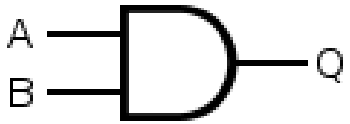
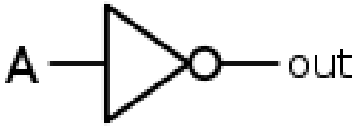


The P6 Processor Micro-Architecture with Advanced Transfer Cache Enhancement

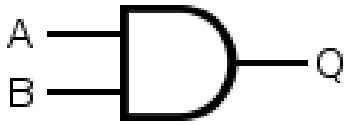
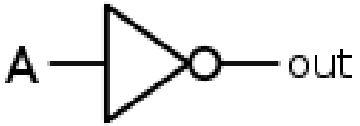




Grundlegende Gatter

	UND	ODER	NICHT
Graphisch			
Aussagelogik	$Q = A \wedge B$	$Q = A \vee B$	$out = \neg A$
C	Q = A&B	Q = A|B	out = !A (~A)
Schaltalgebra	$Q = A \bullet B$ (AB)	Q = A+B	$out = \overline{A}$

Wahrheitstabellen

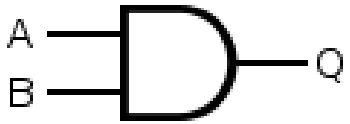
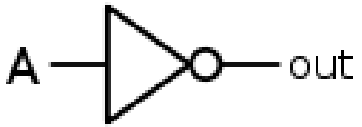
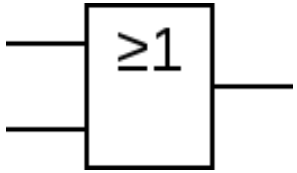
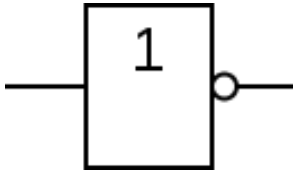
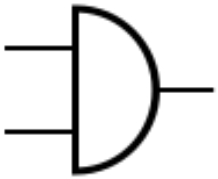
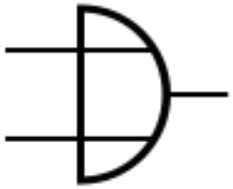
	UND	ODER	NICHT
Graphisch			

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

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

Alternative Darstellungen

	UND	ODER	NICHT
MIL/ANSI			
IEC			
DIN			

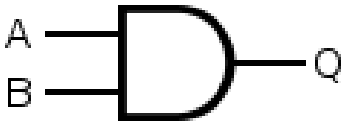
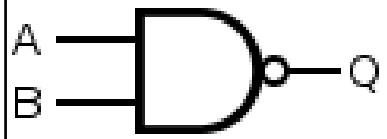
Weitere Gatter

	OR		NOR (NOT OR)
MIL/ANSI			

a	b	a + b
0	0	0
0	1	1
1	0	1
1	1	1

a	b	a NOR b	$\overline{(a + b)}$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

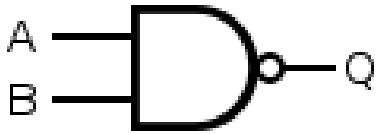
Weitere Gatter

		AND	NAND (NOT AND)
MIL/ANSI			

a	b	a * b
0	0	0
0	1	0
1	0	0
1	1	1

a	b	a NAND b	$\overline{(a * b)}$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Weitere Gatter

	XOR	NAND	NOR
MIL/ANSI			

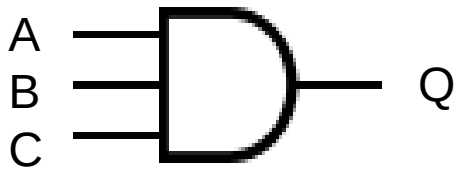
a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

Größere Gatter

- UND-, ODER-, ... Gatter können mehr als 2 Eingänge haben
- n-Input UND-Gatter hat als Ausgang 1, nur wenn alle Eingänge den Wert 1 haben
- n-Input ODER-Gatter hat als Ausgang 0, nur wenn alle Eingänge den Wert 0 haben
- 3-Input UND-Gatter:

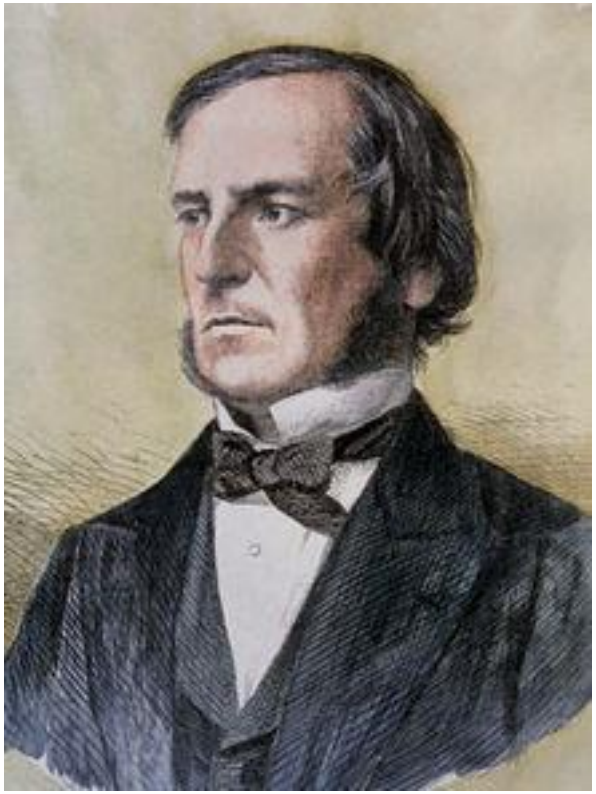


A	B	C	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Boolesche Algebra

Boolesche Algebra

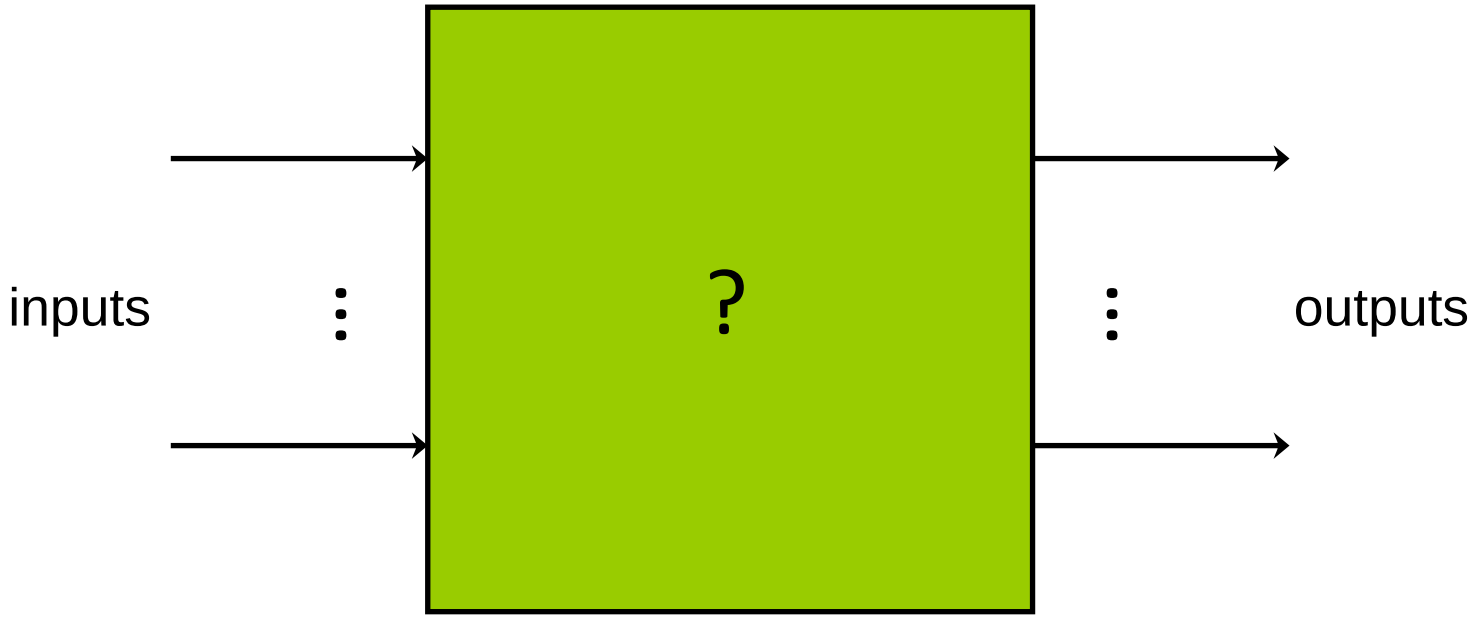
- George Boole (1815-1864): Algebra der Logik mit zwei Werten 0 und 1



Quelle: Wikipedia

Implementierung Boolescher Funktion

- Wie findet man ein Schaltnetz für eine beliebige boolesche Funktion?
- Schaltnetz (*combinational logic*):
  - Bausteine, die Daten verarbeiten
  - Ausgangssignale hängen nur von aktuellen Eingängen ab
  - Ausgangssignale hängen nicht von den internen Zuständen ab

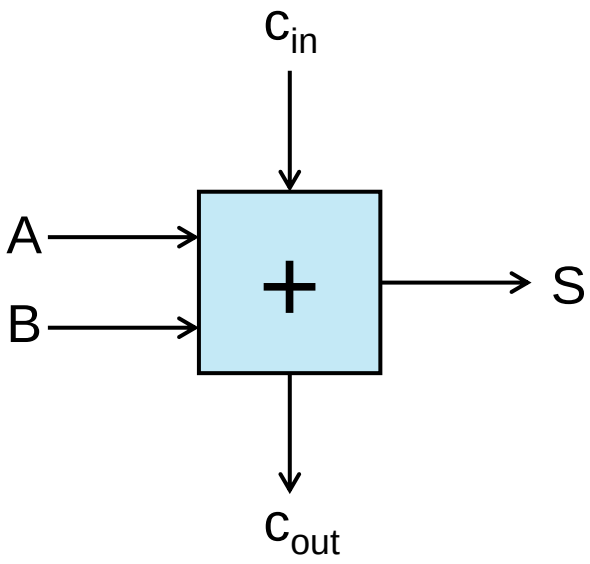


Spezifikation einer Funktion

- Wie spezifiziert man eine Boolesche Funktion. Das kann mittels natürlicher Sprache z.B.: Ausgang = 1 wenn ....
- $Output_A = F(Input_0, Input_1, \ldots, Input_{N-1})$
- $Output_B = F'(Input_0, Input_1, \ldots, Input_{N-1})$
- …
- Methoden:
  - Wahrheitstabelle
  - Disjunktive Normalform (*Sum of products*)
    - Disjunktion = ODER-Verknüpfung
    - $Z = (A \bullet B) + (\neg A \bullet B) = AB + \neg AB = AB + \overline{A}B$
  - Konjunktive Normalform (*Product of sums*)
    - Konjunktion = UND-Verknüpfung
    - $Z = (A+B) \bullet (\neg A+B) = (A+B) \bullet (\overline{A}+B)$

Beispiel: Wahrheitstabelle FA

- Disjunktiven Normalform am Beispiel: Volladdierer



Inputs			Outputs	
A	B	C_{in}	C_{out}	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
			MSB	LSB

Ableitung disjunktiven Normalform (DNF)

Inputs			Outputs	
A	B	C_{in}	C_{out}	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

- Cout = ¬ABCin+A¬BCin+AB¬Cin+ABCin
- S = ¬A¬BCin+¬AB¬Cin+A¬B¬Cin+ABCin

- Jede Zeile der Wahrheitstabelle entspricht einer Konjunktion
- Für jede Zeile, die als Resultat eine 1 liefert, wird eine Konjunktion gebildet, die alle Variablen der Funktion (der Zeile) verknüpft
  - Variablen, die in der Zeile mit 1 belegt sind, werden dabei nicht negiert und Variablen, die mit 0 belegt sind, werden negiert

Ableitung konjunktiven Normalform (KNF)

A	B	F
0	0	0
0	1	1
1	0	0
1	1	1

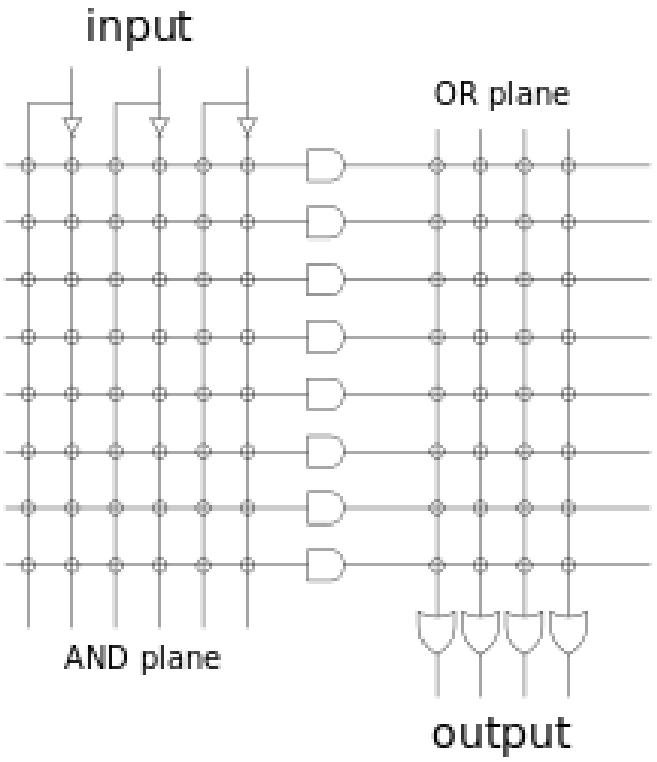
- Jede Zeile der Wahrheitstabelle entspricht einer Disjunktion:
  - negierter Eingang bei 1, unveränderter Eingang bei 0
- Die Disjunktionen bildet man bei denen der Ausgang 0 ist. Alle Terme werden mit UND verknüpft

- F = (A+B)•(¬A+B)
- Kontrolle:

A	B	A+B	¬A+B	(A+B)•(¬A+B)
0	0	0	1	0
0	1	1	1	1
1	0	1	0	0
1	1	1	1	1

Disjunktive Normalform

- Jede einwertige Boolesche Funktion kann als Disjunktion von Konjunktionen dargestellt werden
- Analog: Konjunktive Normalform ist eine Konjunktion von Disjunktionen
- Es gibt immer ein Schaltnetz, das nur aus NICHT, UND und ODER-Gattern besteht
- Maximal 3 Stufen: NICHT-Gatter, dann UND-Gatter, schließlich ein ODER-Gatter mit vielen Eingängen
- NICHT, UND und ODER bilden einen funktional vollständigen Operatorensatz



Programmierbare Logische Anordnung, *Programmable Logic Array (PLA)*
Quelle: Wikipedia

Weitere vollständige Operatorensätze

	NICHT	UND	ODER
Boolsche Basis	¬a	a·b	a+b
NICHT, UND	¬a	a·b	¬(¬a·¬b)
NICHT, ODER	*Übung*		
NAND	*Übung*		
NOR	*Übung*		

a	b	¬a·¬b	¬(¬a·¬b)	a+b
0	0	1	0	0
0	1	0	1	1
1	0	0	1	1
1	1	0	1	1

Boolesche Algebra (Zusammenfassung)

- Zwei Elemente: 0, 1
- Zwei binäre Verknüpfungen: UND (·), ODER (+)
- Eine unäre Verknüpfung: NICHT (¬)
- 8 Axiome:
  - Neutrale Elemente: $a \cdot 1 = a$, $a+0 = a$
  - Komplementäre Elemente: $a \cdot \neg a = 0$, $a+\neg a = 1$
  - Kommutativgesetze: $a \cdot b = b \cdot a$, $a+b = b+a$
  - Distributivgesetze: $(a + b) \cdot c = a \cdot c + b \cdot c$, $a+(b \cdot c) = (a+b) \cdot (a+c)$
- Bemerke: UND hat höhere Priorität als ODER
  - $a+b \cdot c = a+(b \cdot c)$

Weitere Gesetze

- Idempotenzgesetze
  - a·a = a+a = a
- Assoziativgesetze
  - a·(b·c) = (a·b)·c
  - a+(b+c) = (a+b)+c
- De Morgansche Gesetze
  - ¬(a+b) = ¬a·¬b
  - ¬(a·b) = ¬a+¬b
- Und viele mehr…
  - a+(a·b) = a
  - a·(a+b) = a
  - . . .

a	b	¬(a+b)	¬a·¬b
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

a	b	¬(a·b)	¬a+¬b
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

Logikminimierung / MUX

Logikminimierung

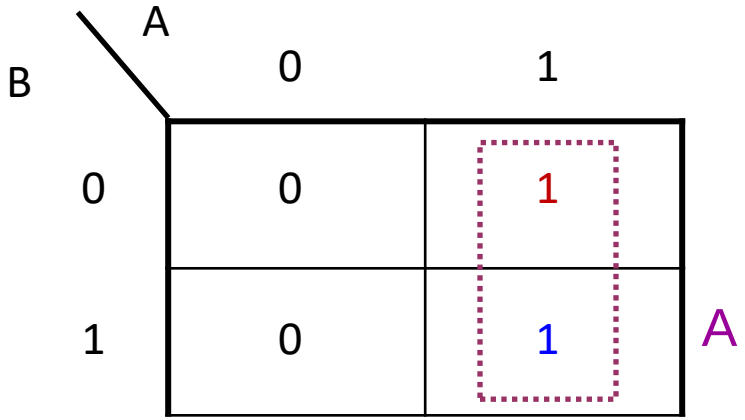
- Reduzierung der Gatter bei gleicher Funktionalität
- Früher wichtiger Kostenfaktor
- Heute werden Verbindungsleitungen und Energiebetrachtungen immer wichtiger

- Beispiel:
  - $C_{out} = \neg ABC_{in} + A\neg BC_{in} + AB\neg C_{in} + ABC_{in}$
- Minimiert ergibt sich:
  - $C_{out} = BC_{in} + AC_{in} + AB$

A	B	C_{in}	C_{out}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Karnaugh-Veitch-Diagramm [nicht klausurrelevant]

A	B	C_{out}
0	0	0
0	1	0
1	0	1
1	1	1



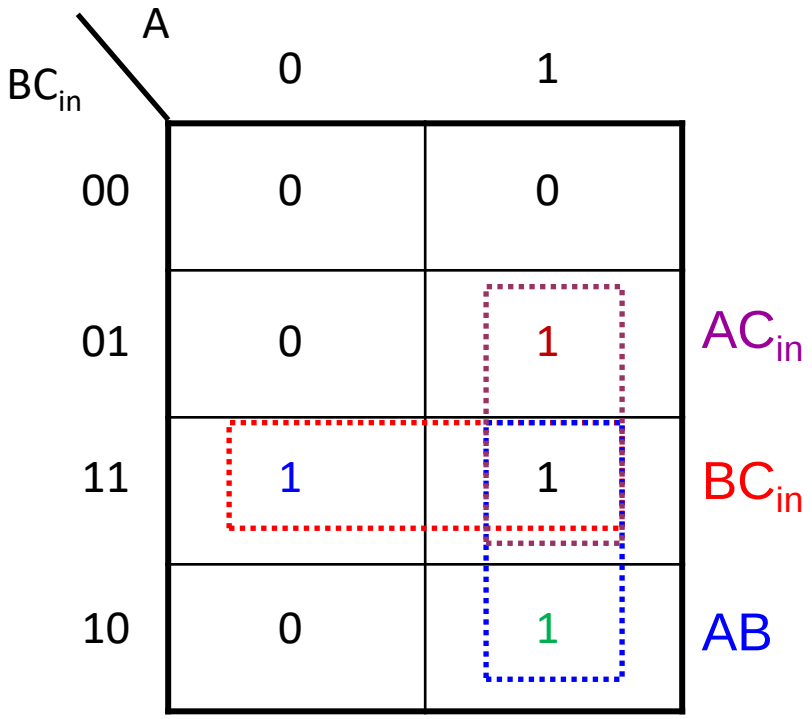
$C_{out} = A\,\neg B + AB$

$C_{out} = A\,(\neg B + B) = A$

$C_{out} = A$

Karnaugh-Veitch-Diagramm [nicht klausurrelevant]

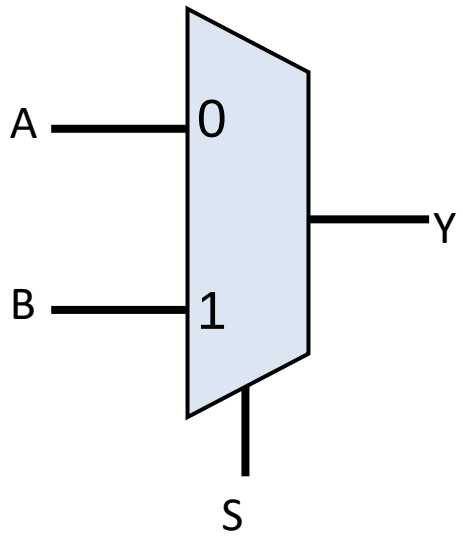
A	B	C_{in}	C_{out}
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



$$C_{out} = BC_{in} + AB + AC_{in}$$

Multiplexer (MUX)

- Wählt je nach Steuersignal S einen der Inputs A oder B aus.



Y = (S) ? B : A

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

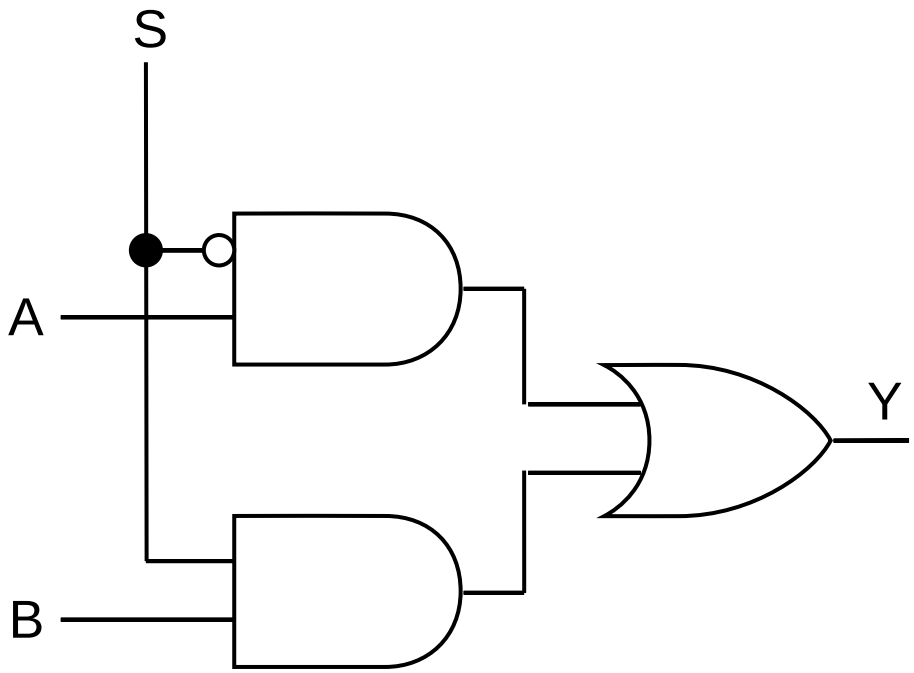
Y1 = ¬S A

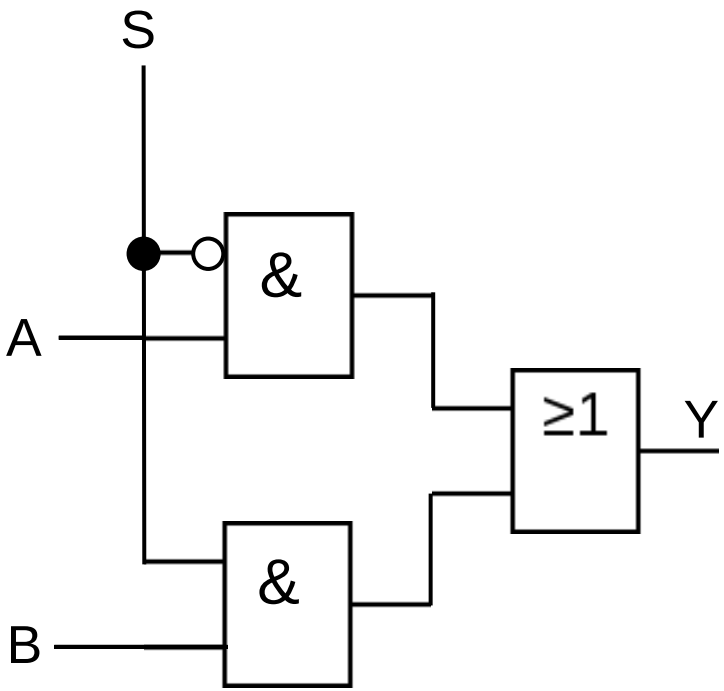
Y2 = S B

Y = ¬S A + S B

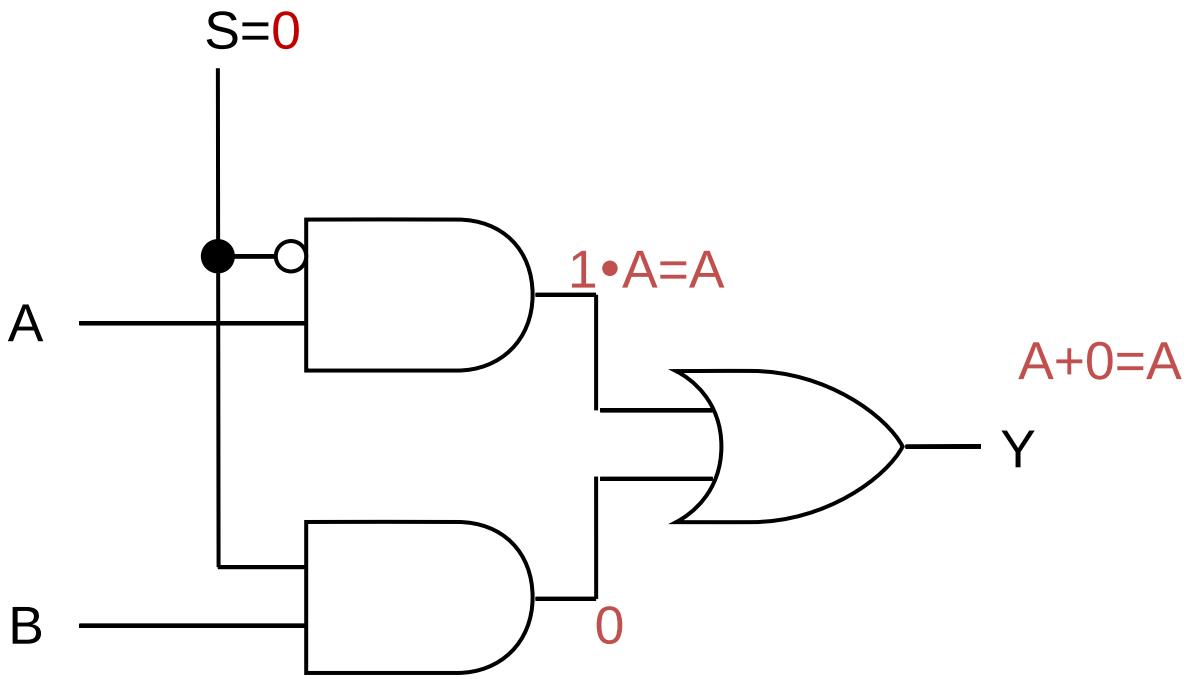
MUX-Schaltnetz

- Boolesche Gleichung: $Y = \neg S\, A + S\, B$



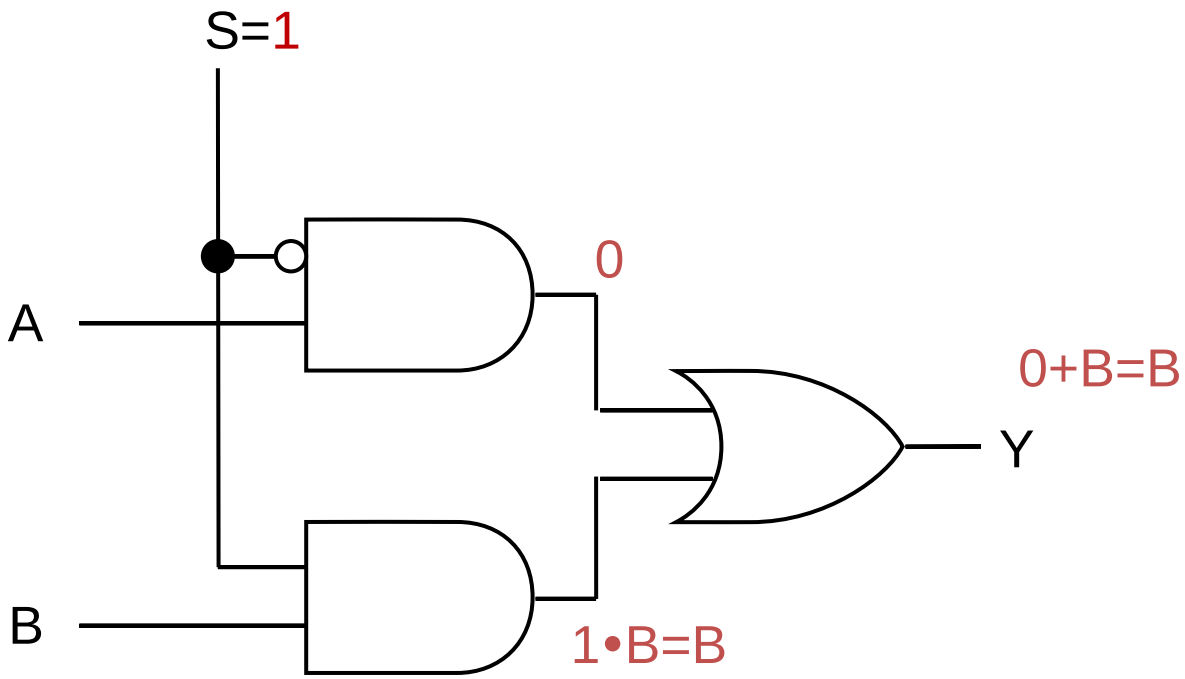


MUX-Schaltnetz



$$Y = \neg S\, A + S\, B$$

MUX-Schaltnetz



$$Y = \neg S\, A + S\, B$$

MUX Wahrheitstabelle mit Don‘t-Cares

- Don’t-Care (X oder -) bedeutet, dass dieser Wert keinen Einfluss auf die Logikschaltung bzw Funktion hat

Vollständige Wahrheitstabelle

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

$Y = \neg S\ A + S\ B$

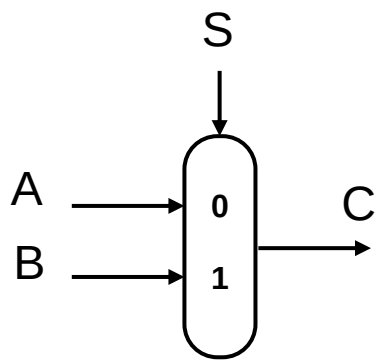
Wahrheitstabelle mit Don‘t-Cares

S	A	B	Y
0	0	X	0
0	1	X	1
1	X	0	0
1	X	1	1

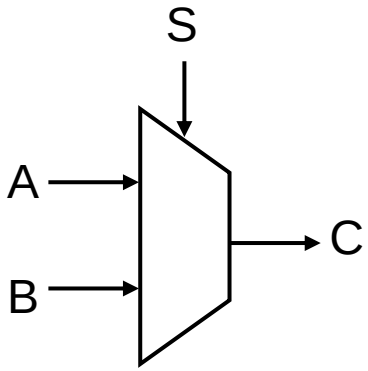
$Y = \neg S\ A + S\ B$

MUX-Symbol im Textbuch

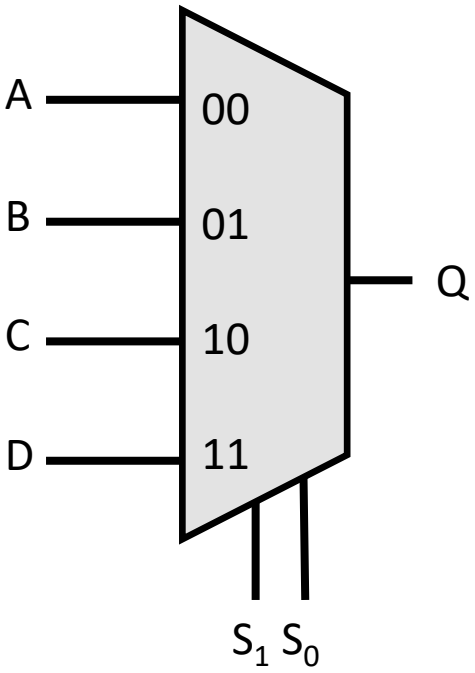
Symbol im Textbuch:

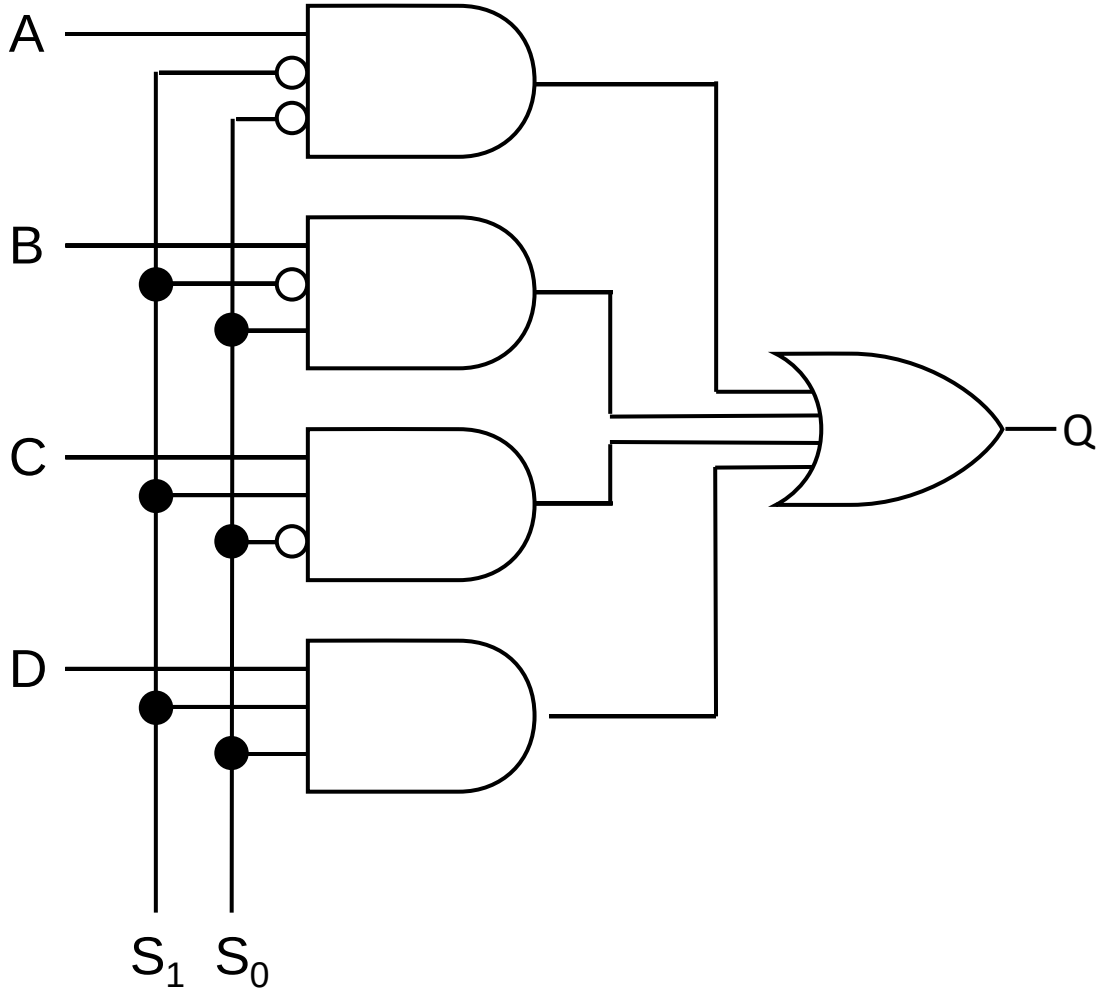


gebräuchlicher:

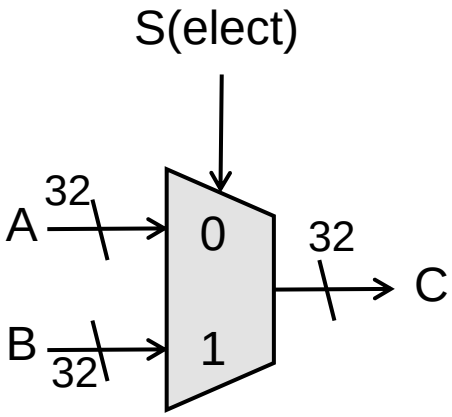


4-Input MUX

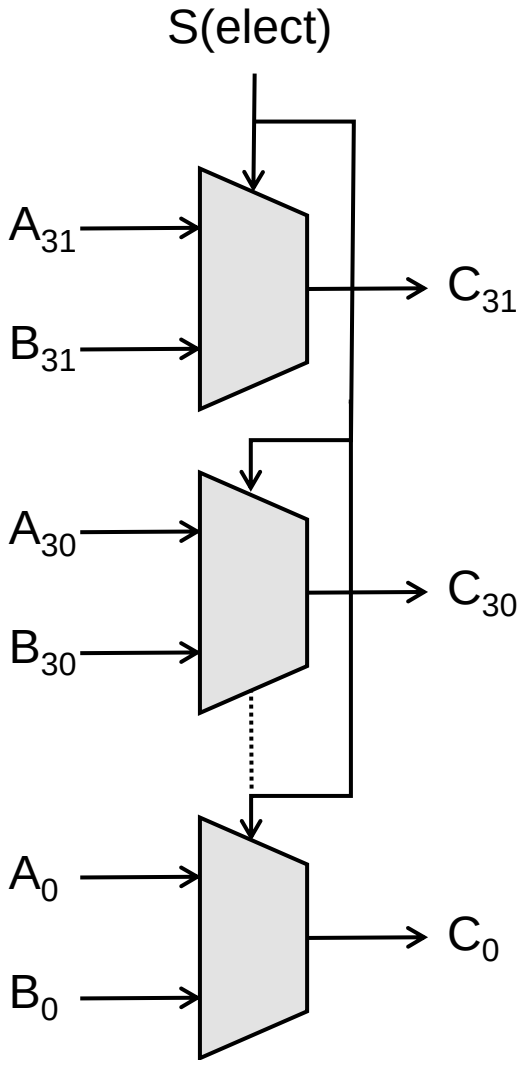




32-Bit 2-1 Multiplexer



32 Bit breiter 2-1 MUX



Tatsächlich ein Array von 32 Ein-Bit breitem MUX

Decoder / ALU

1-aus-n-Decoder

- Schaltung mit m Eingängen $a_{m-1}...a_0$ und $n = 2^m$ Ausgängen $y_{n-1}...y_0$
- Ausgang y_i wird 1 (High) wenn die m Eingänge der Dualzahl y_i entsprechen
  - Alle anderen Ausgänge sind 0 (Low)

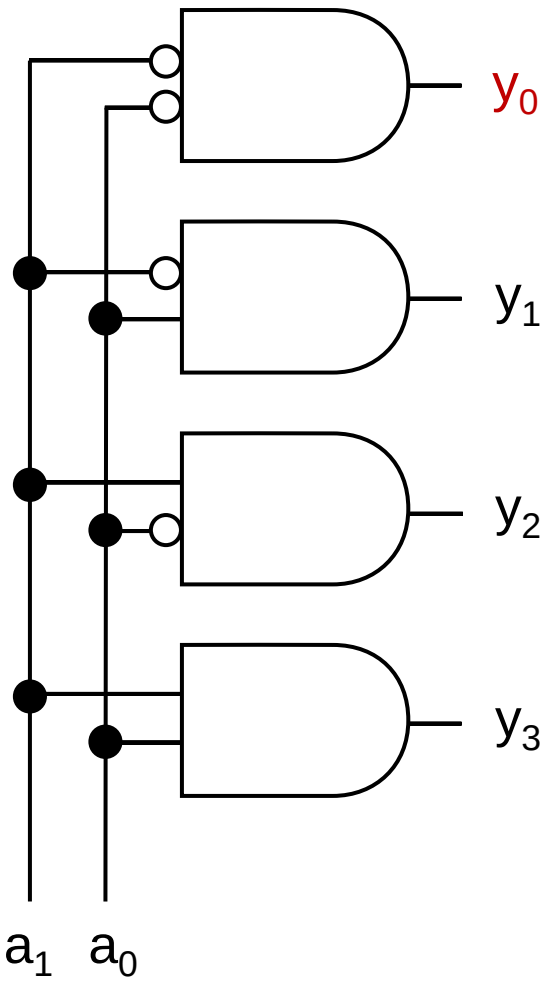
- Beispiel: Wahrheitstabelle eines 1-aus-4 Decoders

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

Schaltnetz 1-aus-4-Decoder

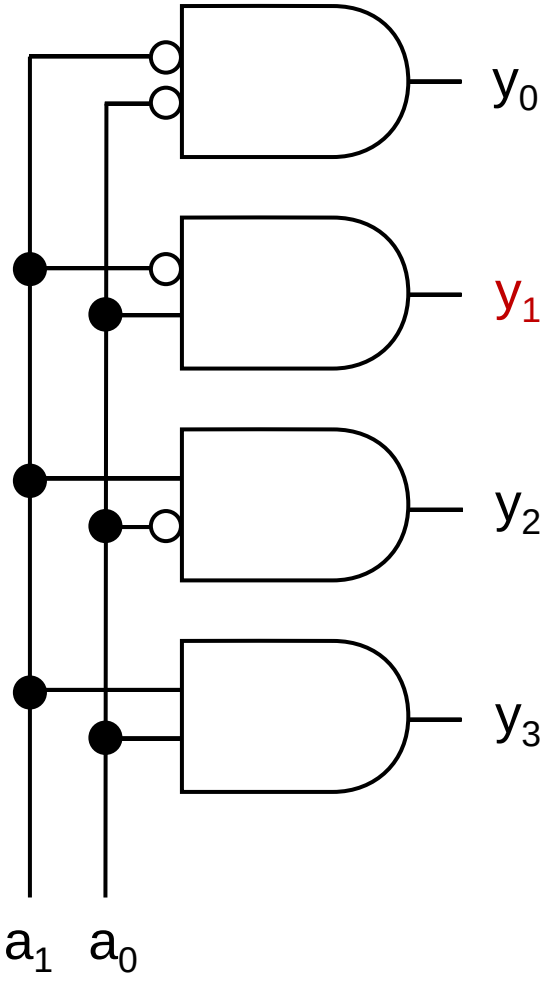
- Welche Gatter werden verwendet?

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



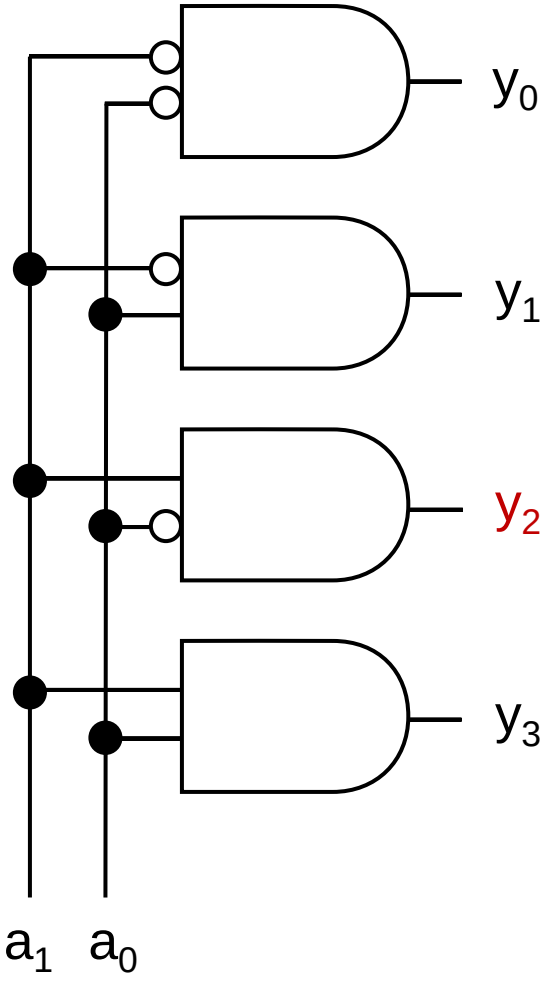
Schaltnetz 1-aus-4-Decoder

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



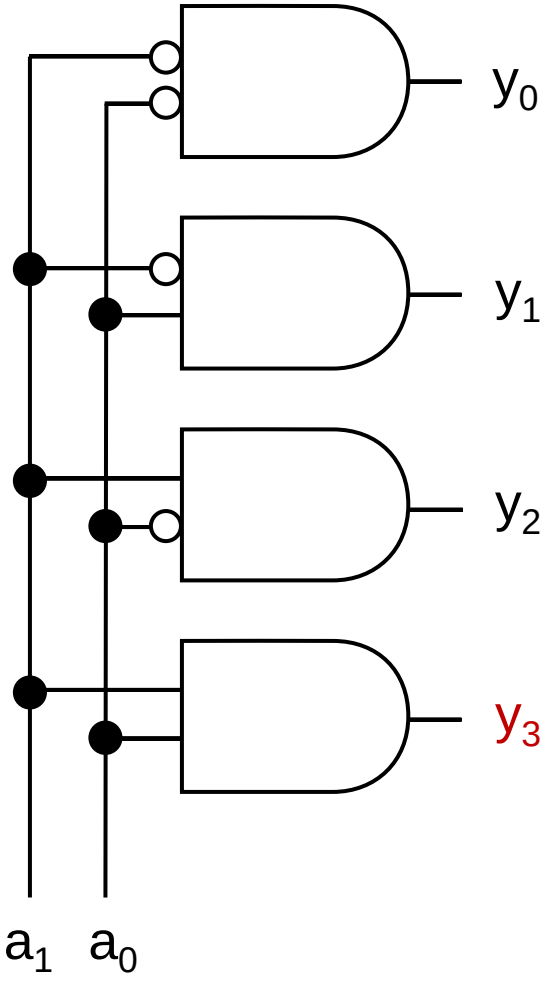
Schaltnetz 1-aus-4-Decoder

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



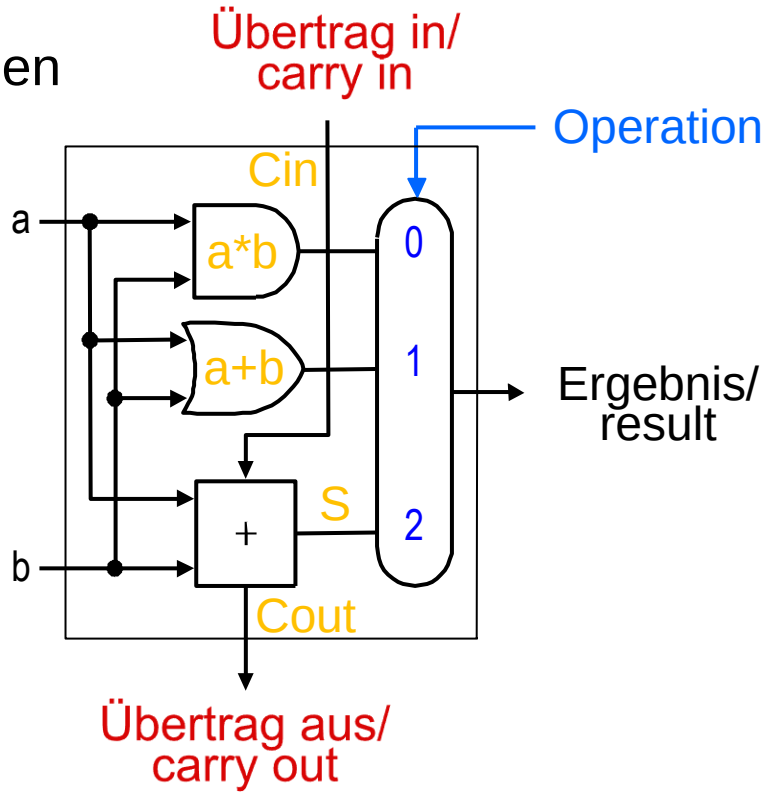
Schaltnetz 1-aus-4-Decoder

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



Arithmetisch logische Einheit (ALU)

- MUX Funktion kann benutzt werden, um eine 1-Bit ALU zu konstruieren, die die Operationen AND, OR und + (ADD) unterstützt
  - Notwendige Gatter: AND, OR, FA & MUX (Selection)
- Einfach alles parallel ausführen und den richtigen Output auswählen
- Eingänge: a, b, carry in, carry out, Operation

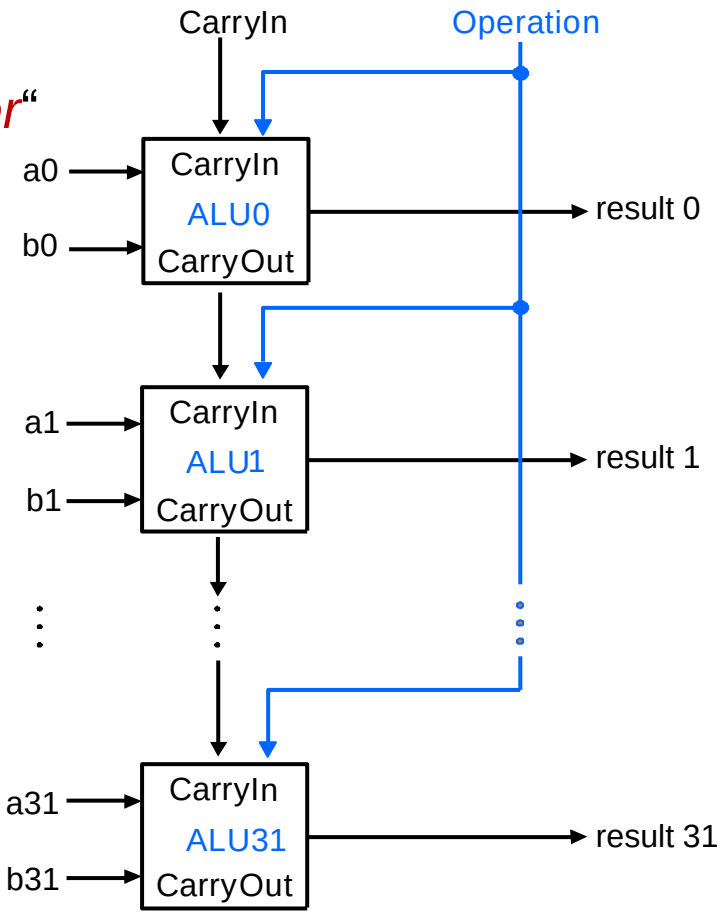


32-Bit ALU

- 1-Bit ALU kann zur Realisierung einer 32-Bit ALU genutzt werden.

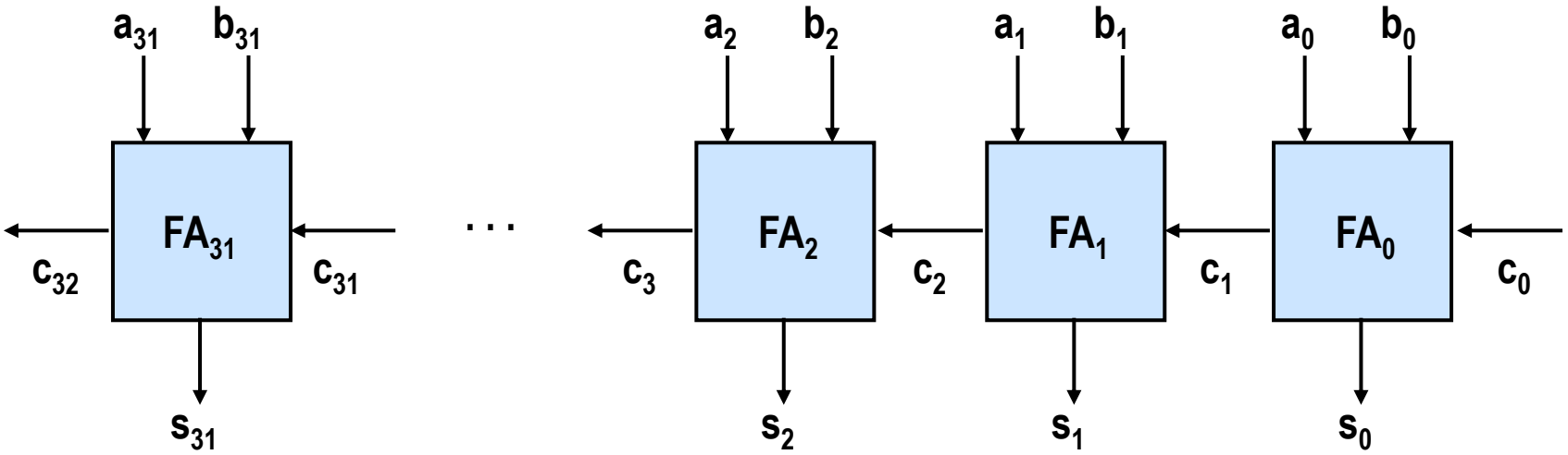
- Angenommen wird "*ripple-carry adder*"
  - Cout/CarryOut „ripples" through adder chain

- "*Carry-look-ahead adder*" ist schneller



32-Bit Addierer

Ripple-Carry-Addierer:

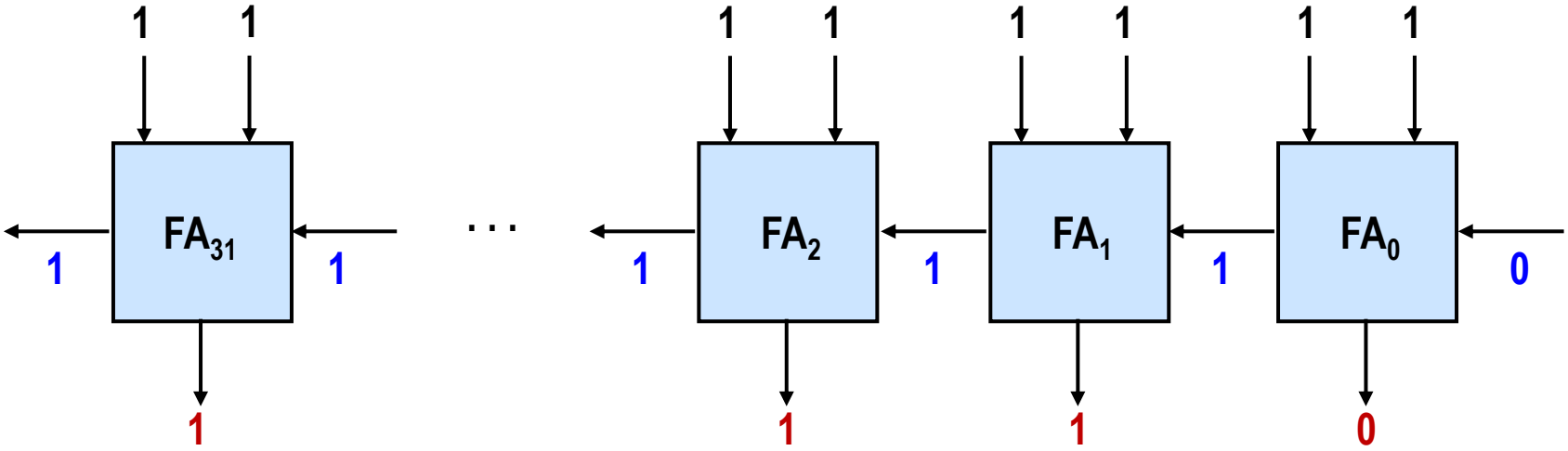


FA = Full Adder

- FA_0 wird üblicherweise als Halbaddierer ausgelegt wenn C_0 nicht vorhanden

32-Bit Addierer

Ripple-Carry-Addierer:

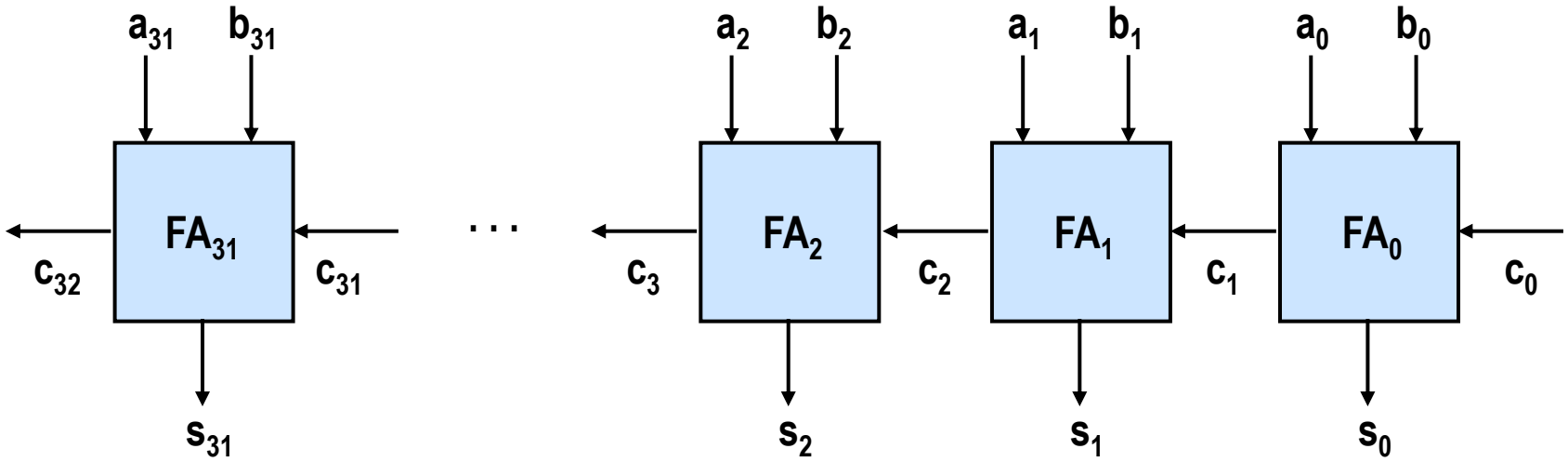


FA = Full Adder

- Problem: Lange Durchlaufzeit für den Übertrag bei großer Stellenzahl, z.B. bei 32-Bit- oder 64-Bit-Addierer
- Lösung: Parallelisierung der Übertragsbildung:

  →Carry-Look-Ahead-Addierer (CLA-Addierer)

Carry-Look-Ahead Addierer (Motivation)



- Gleichungen (Summe von Produkten):

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

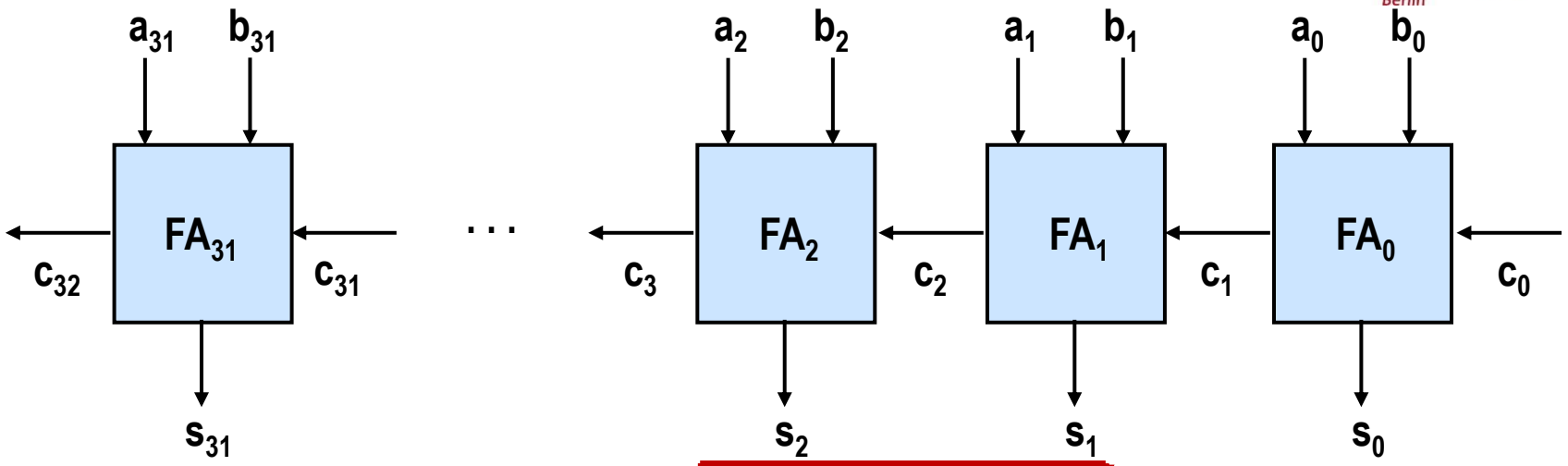
$c_2 = b_1c_1 + a_1c_1 + a_1b_1 = b_1b_0c_0 + b_1a_0c_0 + b_1a_0b_0 + a_1b_0c_0 + a_1a_0c_0 + a_1a_0b_0 + a_1b_1$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2 = ...$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3 = ...$

- Nicht umsetzbar! Warum?

Carry-Look-Ahead (CLA) Addierer



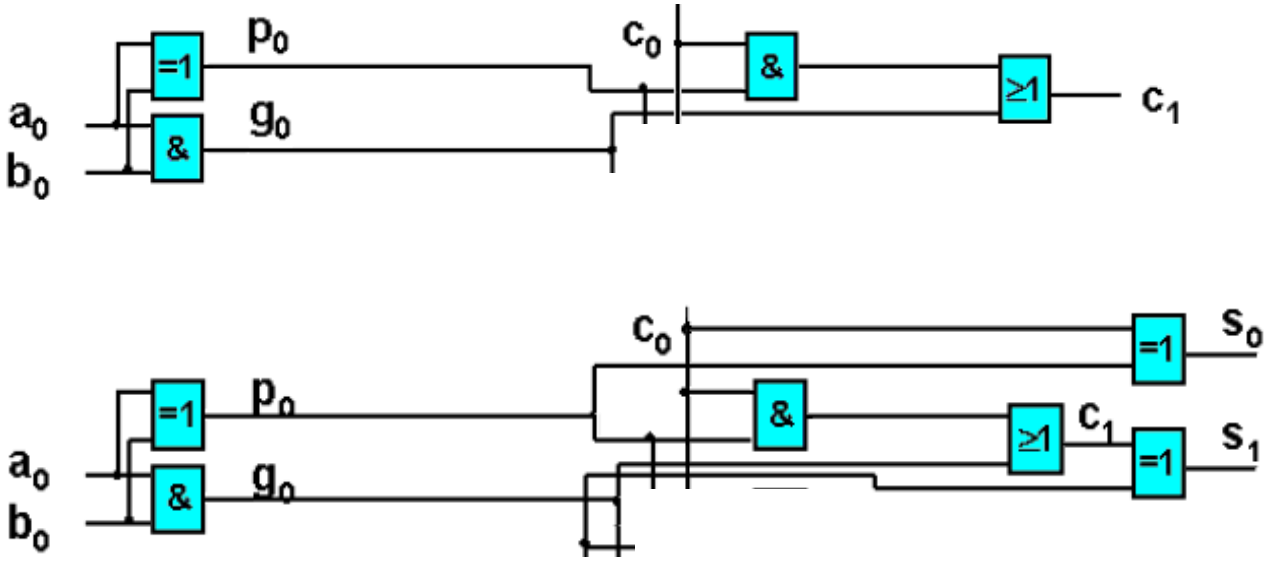
- FA_i wird immer ein Übertrag **g**enerieren, wenn $g_i = a_i\, b_i$ (= 1)
- FA_i **p**ropagiert den Übertrag c_{i-1}, wenn $p_i = a_i$ XOR b_i (= 1)

$c_1 = g_0 + p_0 c_0 = a_0\, b_0 + (a_0 \text{ XOR } b_0)\, c_0 \quad s_1 = c_0 \text{ XOR } p_0$

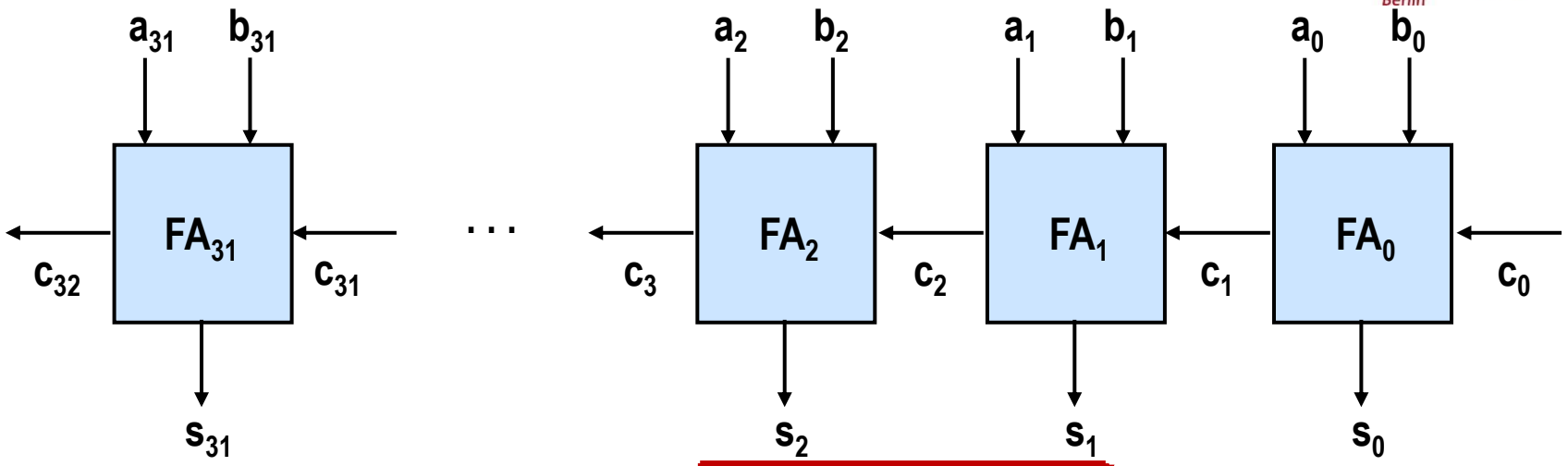
Übertraglogik eines 4-Bit CLA-Addierers

- $c_1 = g_0 + p_0 c_0 = a_0\, b_0 + (a_0 \text{ XOR } b_0)\, c_0 \quad s_1 = c_0 \text{ XOR } p_0$

=1 XOR



Carry-Look-Ahead (CLA) Addierer



- FA_i wird immer ein Übertrag **g**enerieren, wenn $g_i = a_i\, b_i$ (= 1)
- FA_i **p**ropagiert den Übertrag, wenn $p_i = a_i$ XOR b_i (= 1)

  $c_1 = g_0 + p_0c_0 = a_0\, b_0 + (a_0 \text{ XOR } b_0)\, c_0$ $\quad s_1 = c_0 \text{ XOR } p_0$

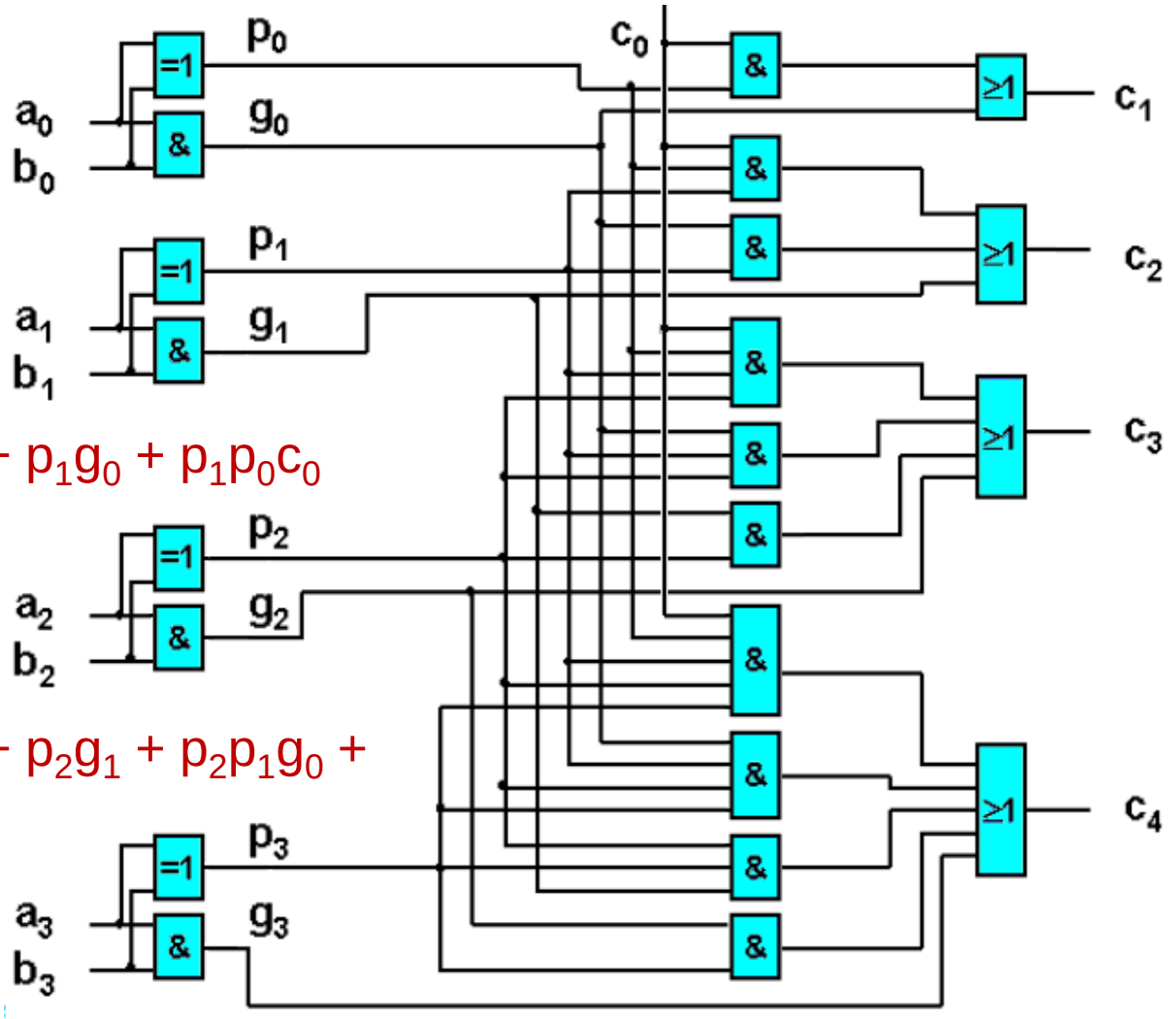
  $c_2 = g_1 + p_1c_1 = g_1 + p_1g_0 + p_1p_0c_0$

  $c_3 = g_2 + p_2c_2 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$

  $c_4 = g_3 + p_3c_3 = ...$
- Umsetzbar! Warum?

Übertraglogik eines 4-Bit CLA-Addierers

=1 XOR

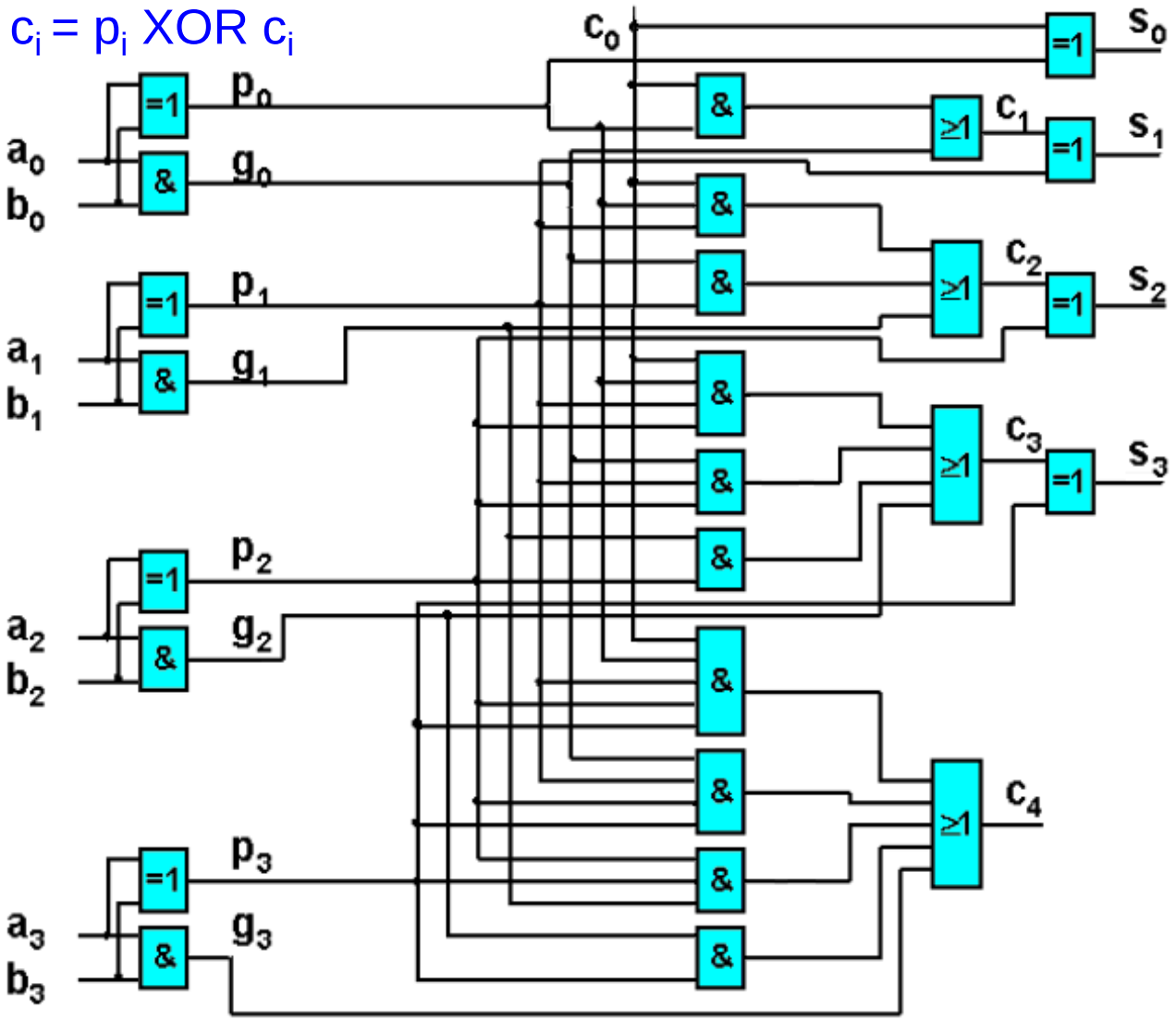


$c_2 = g_1 + p_1c_1 = g_1 + p_1g_0 + p_1p_0c_0$

$c_3 = g_2 + p_2c_2 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$

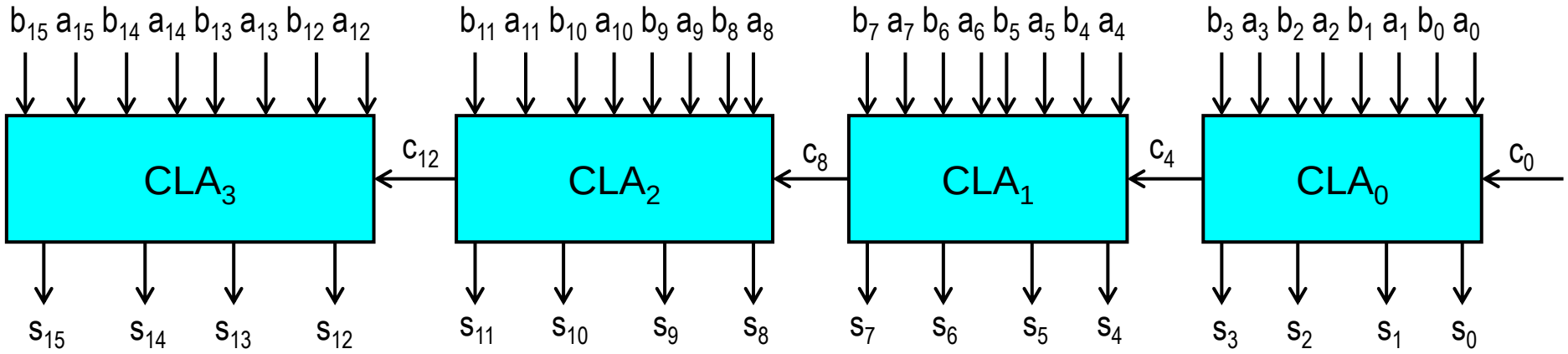
4-Bit CLA-Addierer

$s_i = a_i \text{ XOR } b_i \text{ XOR } c_i = p_i \text{ XOR } c_i$



Größere CLA-Addierer (16-Bit)

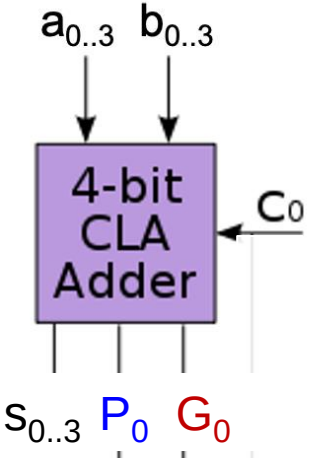
- 16-Bit oder mehr CLA-Addierer werden zu groß (!!!?)
- Man könnte einen 16-Bit CLA-Addierer durch ein Hintereinanderschalten von vier 4-Bit CLA-Addierern realisieren (Ripple-Carry-Prinzip).

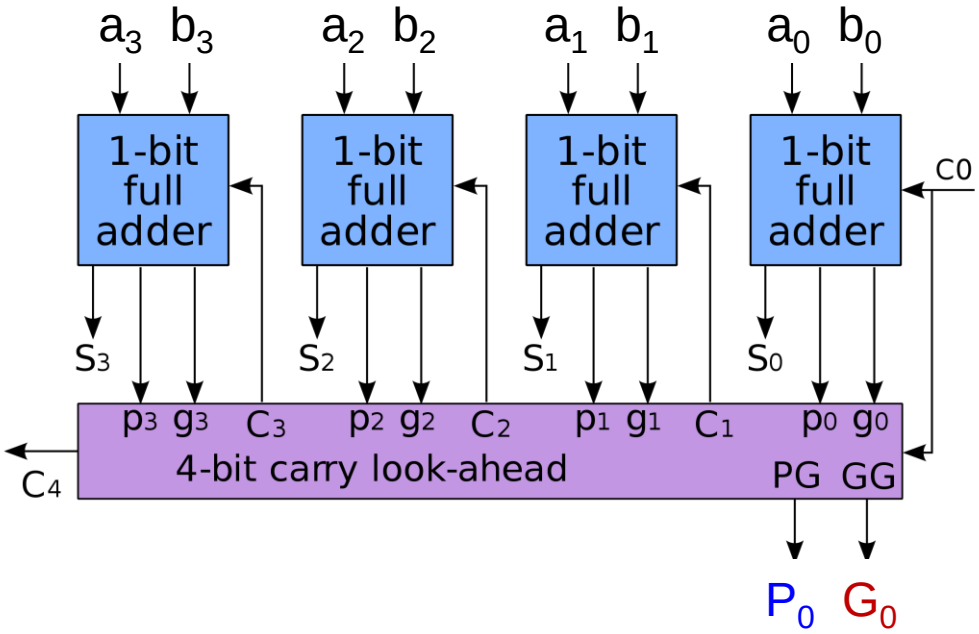


- Besser: Erneutes Anwenden des CLA-Prinzips!

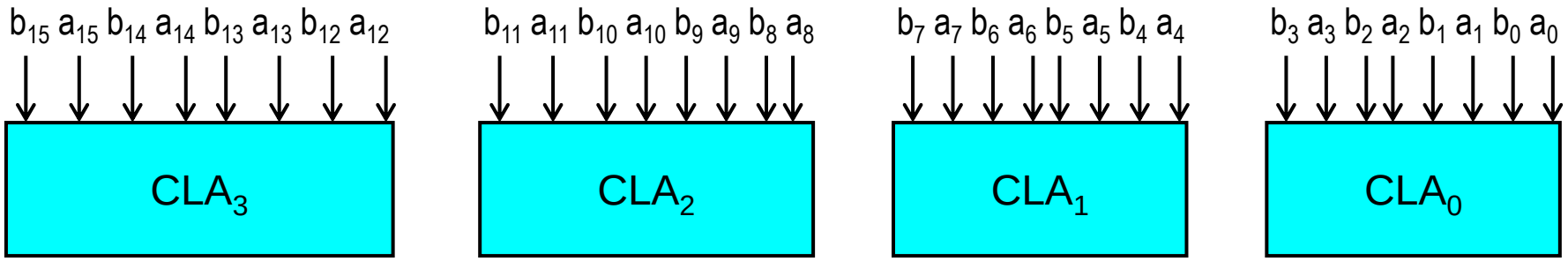
4-Bit CLA-Addierer (PG & GG)

- 4-Bit CLA-Addierer kann auch in übergeordneten Schaltungen verwendet werden
- Jede 4-Bit CLA-Logikschaltung erzeugt ein Propagierungs- und Generierungssignal
  - Gruppenpropagier - (PG) und Gruppengeneriersignale - (GG)
  - GG = $G_0 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$
  - PG = $P_0 = p_0p_1p_2p_3$





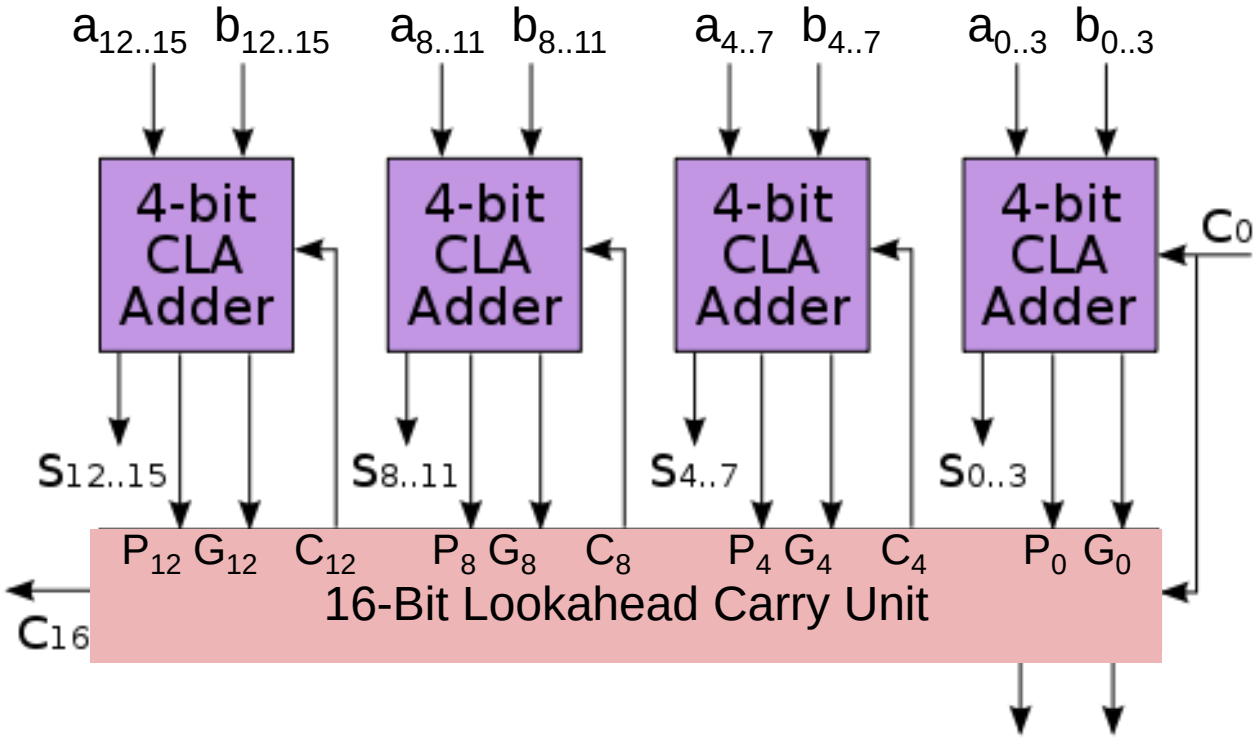
Erneutes Anwenden des CLA-Prinzips



- CLA_i propagiert den Carry, wenn
  - $P_0 = p_3p_2p_1p_0$
  - $P_1 = p_7p_6p_5p_4$
  - $P_2 = p_{11}p_{10}p_9p_8$
  - $P_3 = p_{15}p_{14}p_{13}p_{12}$
- CLA_i generiert ein Carry, wenn
  - $G_0 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$
  - $G_1 = g_7 + p_7g_6 + p_7p_6g_5 + p_7p_6p_5g_4$
  - $G_2 = g_{11} + p_{11}g_{10} + p_{11}p_{10}g_9 + p_{11}p_{10}p_9g_8$
  - $G_3 = g_{15} + p_{15}g_{14} + p_{15}p_{14}g_{13} + p_{14}p_{14}p_{13}g_{12}$

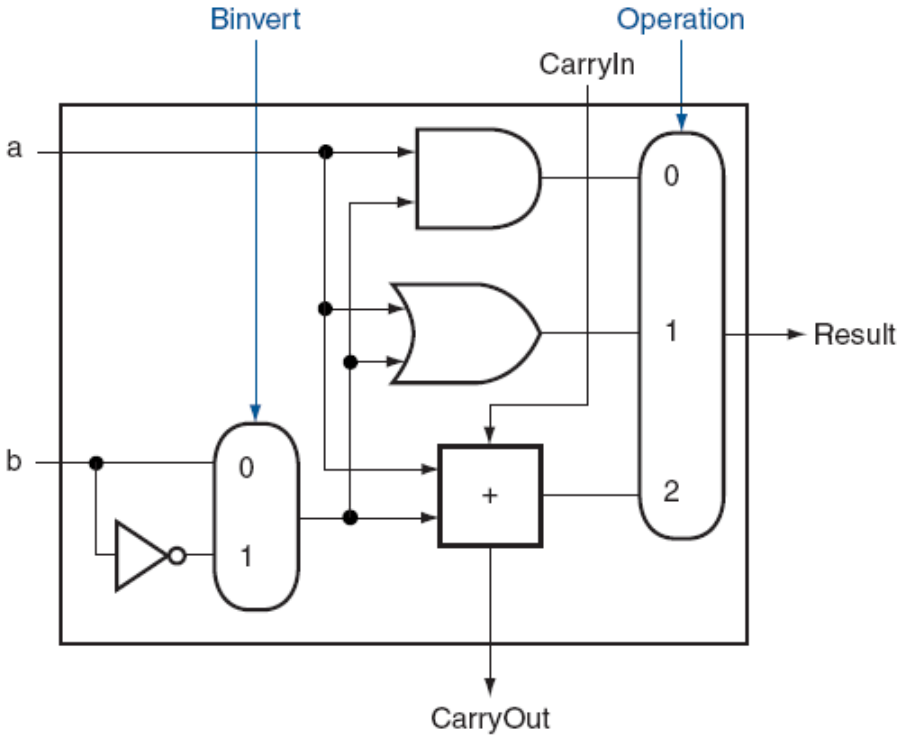
Erneutes Anwenden des CLA-Prinzips

- CLA_i hat ein *CarryIn*, wenn
  - $C_4 = G_0 + P_0c_0$
  - $C_8 = G_1 + P_1G_0 + P_1P_0c_0$
  - $C_{12} = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0c_0$



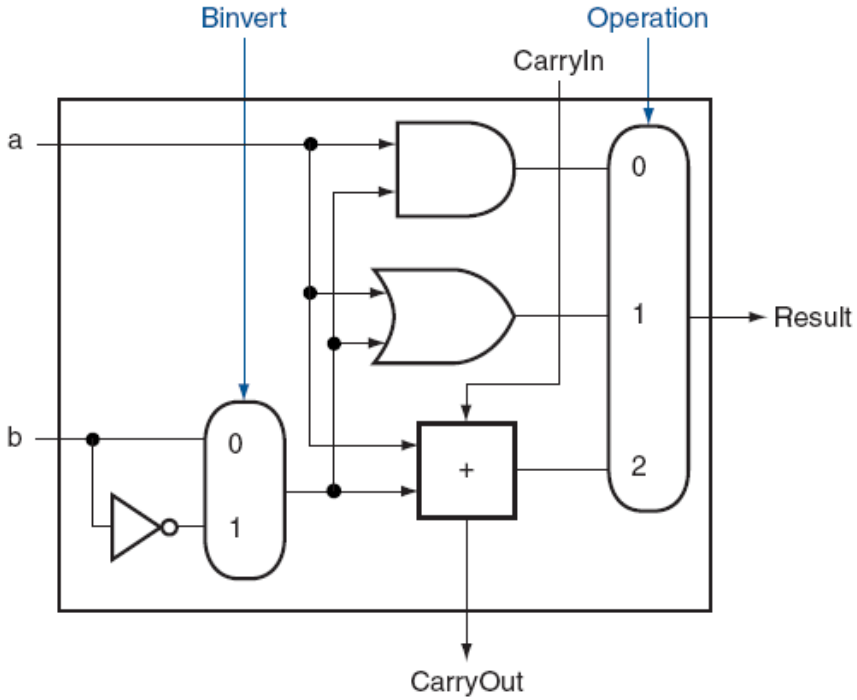
Support für die Subtraktion a-b

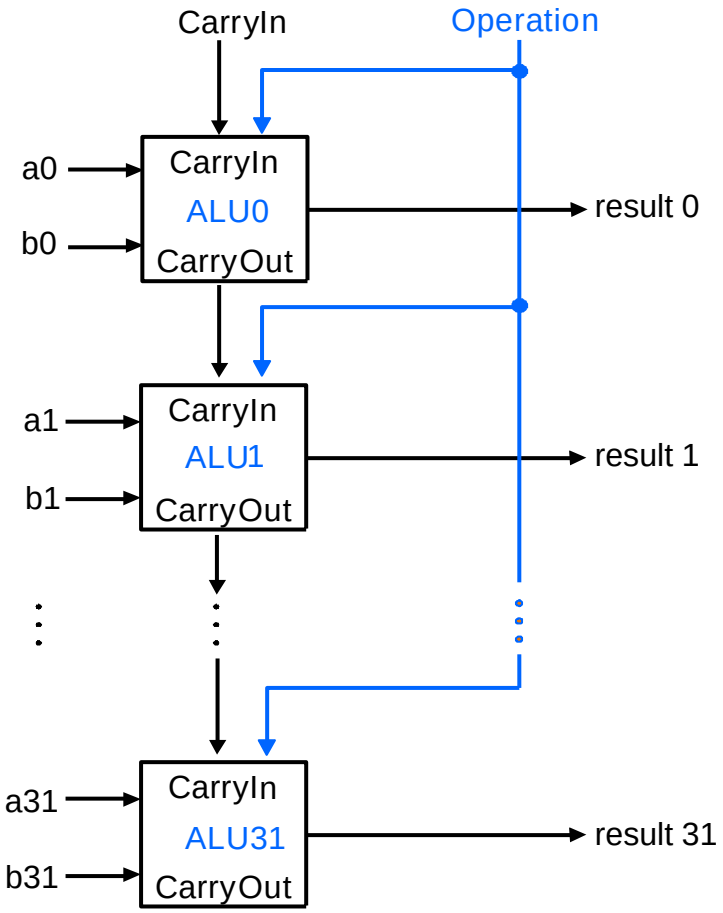
- Subtraktion verwendet 2er-Komplement:
  → einfach b negieren und addieren
- Wie negiert man b (2er comp.)?
  → alle Bits invertieren und 1 addieren
- Einfache Lösung für Subtraktion:
  - setze *CarryIn* von ALU0 auf 1
    → +1 addieren
  - *Binvert* auf 1
    → b invertieren
  - *Operation* auf 10 (2)
    → FA Ausgang



Support für die Subtraktion a-b

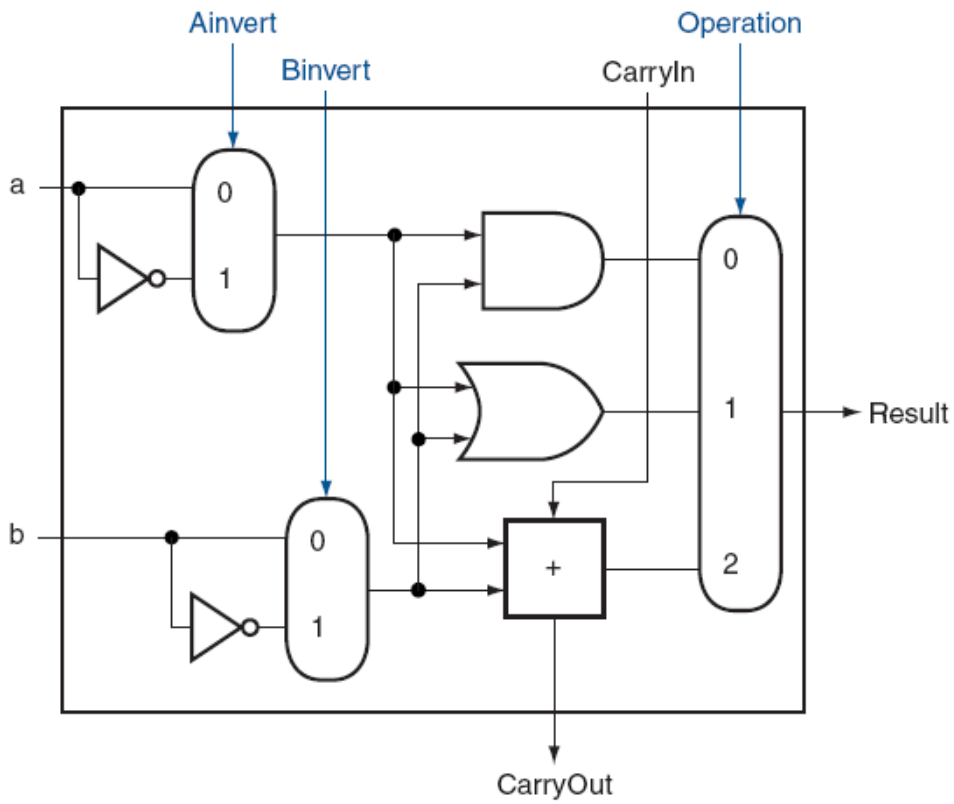
- Um zu subtrahieren, setze *CarryIn* von ALU0 auf 1, *Binvert* auf 1 und *Operation* auf 10





Support für NAND und NOR Logik

- Wahlweise kann auch a invertiert werden:



a	b	a NAND b	a NOR b
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

NAND: $Z = \neg (a * b) = \neg a + \neg b$

NAND: $Z = \overline{a * b} = \overline{a} + \overline{b}$

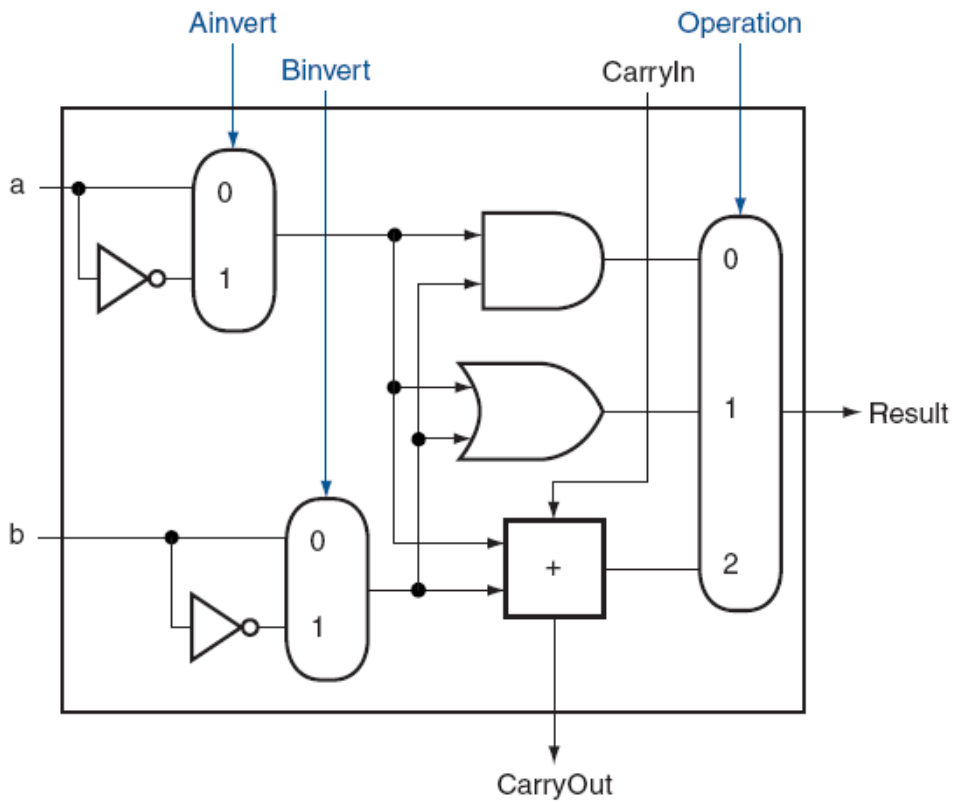
NOR: $Z = \neg (a + b) = \neg a * \neg b$

NOR: $Z = \overline{a + b} = \overline{a} * \overline{b}$

- Wie erhält man „a NAND b”, und wie ein „a NOR b“?

Support für NAND und NOR Logik

- Wahlweise kann auch a invertiert werden:



a	b	a NAND b	a NOR b
0	0	1	1
0	1	1	0
1	0	1	0
1	1	0	0

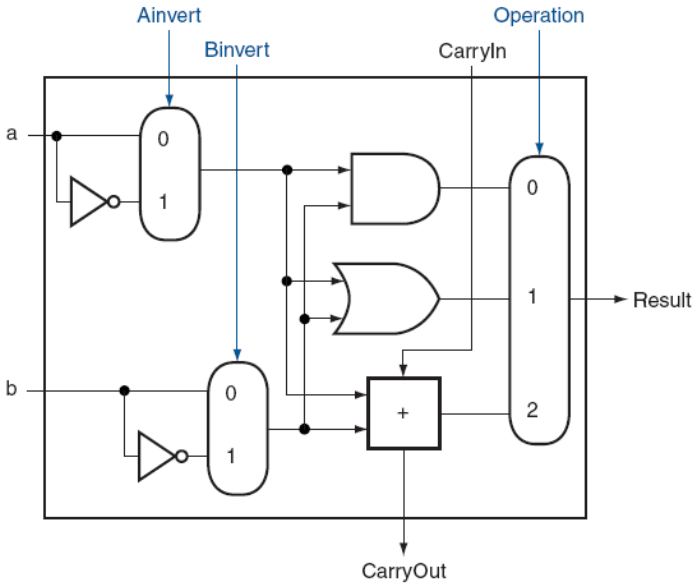
¬a	¬b	¬a OR ¬b	¬a AND ¬b
1	1	1	1
1	0	1	0
0	1	1	0
0	0	0	0

- Wie erhält man „a NAND b”, und wie ein „a NOR b“?

ALU Steuersignale

- Steuersignale um ALU-Operation zu bestimmen:

Gewünschte ALU-Operation	*Ainvert*	*Binvert*	*Operation*
and	0	0	00
or	0	0	01
add	0	0	10
sub, a-b	0	1	10
sub, -a+b	?	?	?
nand	1	1	01
nor	?	?	?

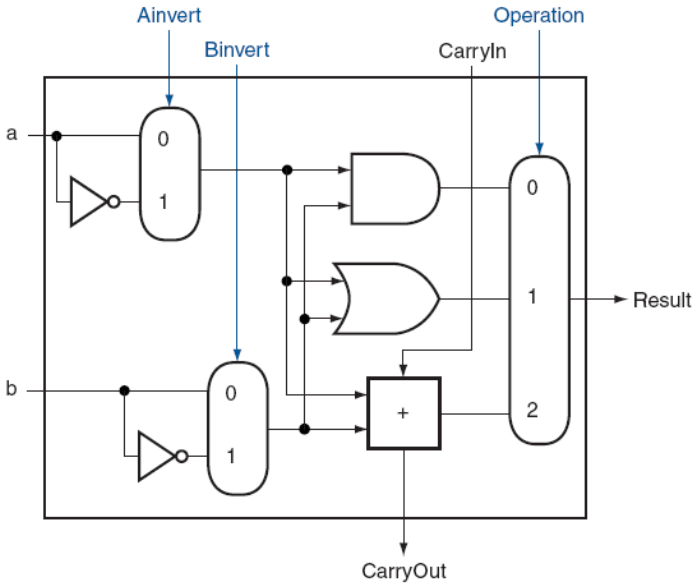


- Steuersignale haben festgelegte Bedeutung!

ALU Steuersignale

- Steuersignale um ALU-Operation zu bestimmen:

Gewünschte ALU-Operation	*Ainvert*	*Binvert*	*Operation*
and	0	0	00
or	0	0	01
add	0	0	10
sub, a-b	0	1	10
sub, -a+b	1	0	10
nand	1	1	01
nor	1	1	00

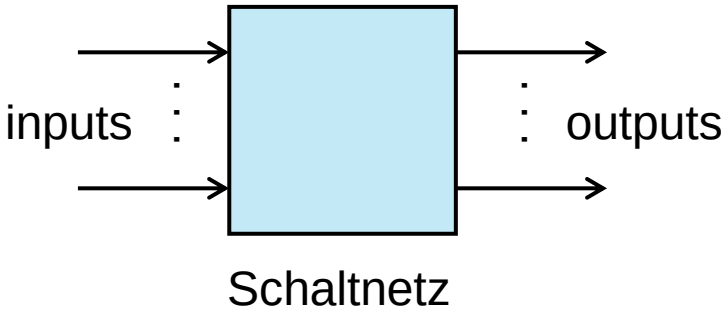


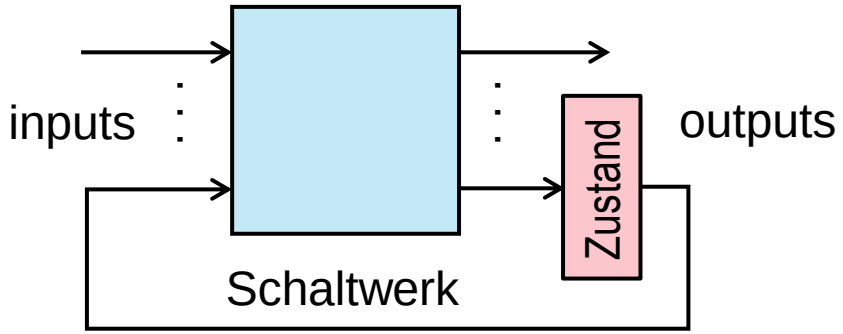
- Steuersignale haben festgelegte Bedeutung!

Schaltwerke

Schaltnetz vs. Schaltwerk

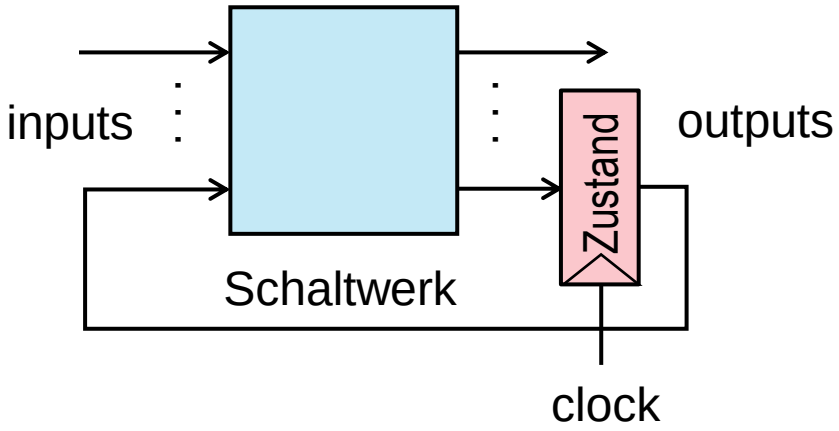
- Zwei Arten von Logikbausteinen:
  - Schaltnetze (*combinational logic*):
    - Bausteine, die Daten verarbeiten
    - Ausgangssignale hängen nur von aktuellen Inputs ab: ALU, MUX…
  - Schaltwerke (*sequential logic, state elements*):
    - Baustein kann Zustand speichern, verfügt über internen Speicher
    - Ausgänge hängen ab von Eingängen **und** vom Inhalt des internen Speichers: Register, Speicher,...





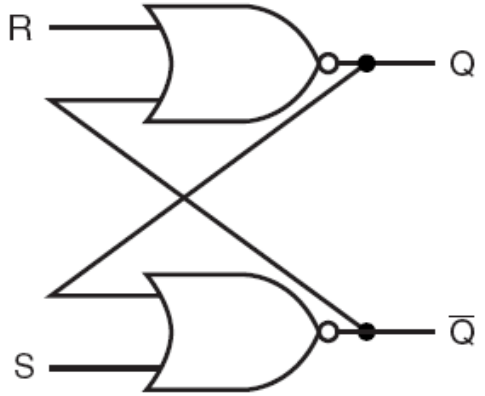
Synchrone vs. Asynchrone Schaltwerke

- Schaltwerke können asynchron oder synchron (durch Taktsignale synchronisiert) sein.
- Taktsignale werden in synchroner Logik verwendet
  - Taktsignal gibt an, wann der Zustand sich ändert



SR-Latch: ungetakteter Zustandsspeicher

- Kreuzgekoppelte “NOR”-Gatter
- Output abhängig vom aktuellen Input und von vergangenen Inputs



- Wahrheitstabelle NOR:

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

Input = 0 ⟶ Output = Gegenteil des anderen Inputs

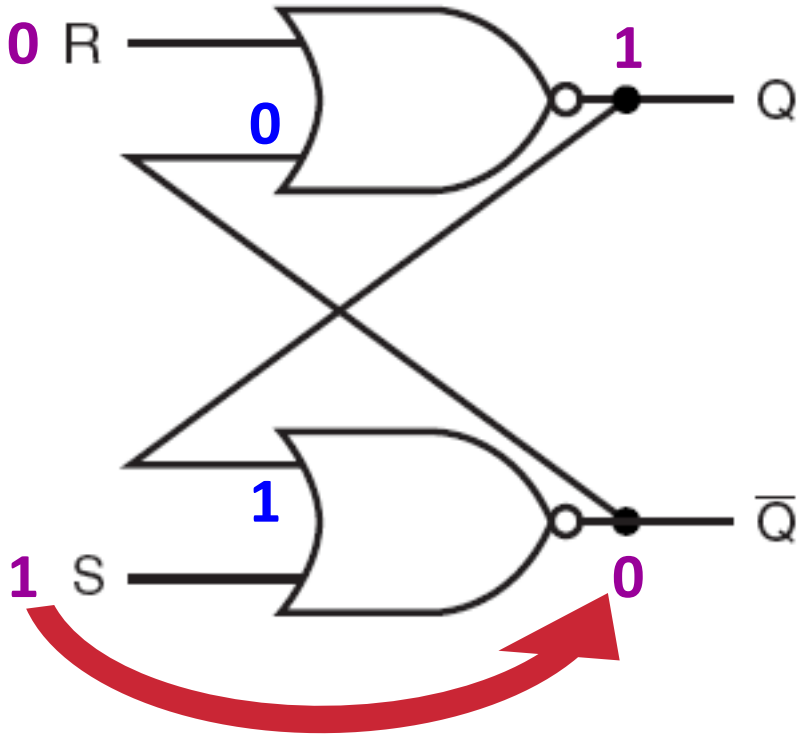
Input = 1 ⟶ Output = 0

SR-Latch - Set

- Q wird gesetzt (1) wenn **S=1, R=0**

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

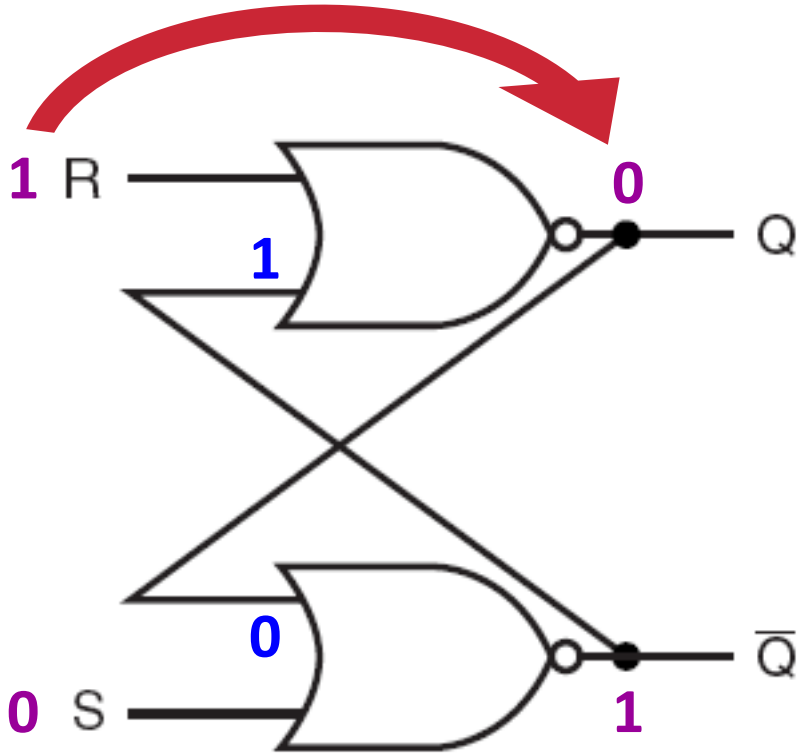
00 → zwingt 1 am Ausgang



Input 1 → zwingt 0 am Ausgang, 11 → Ausgang bleibt 0

SR-Latch - Reset

- Q wird zurückgesetzt (0) wenn **R=1, S=0**

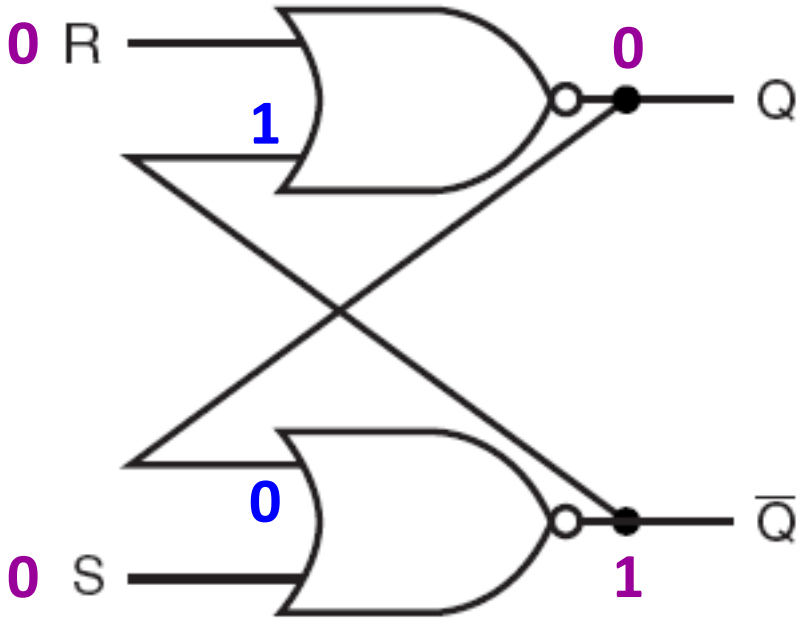


a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

SR-Latch - Halten

- Wenn **R=S=0** und **Q war 0**, wird Q gehalten

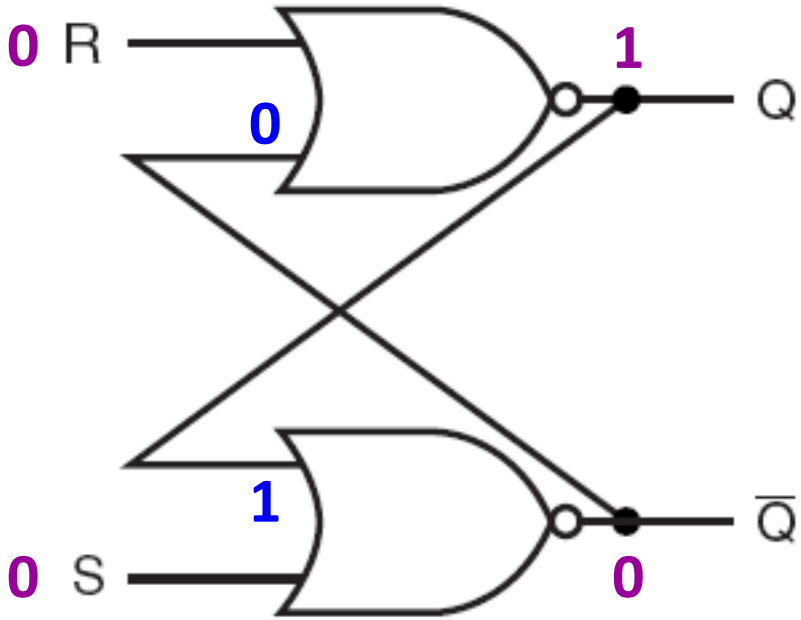
a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0



SR-Latch - Halten

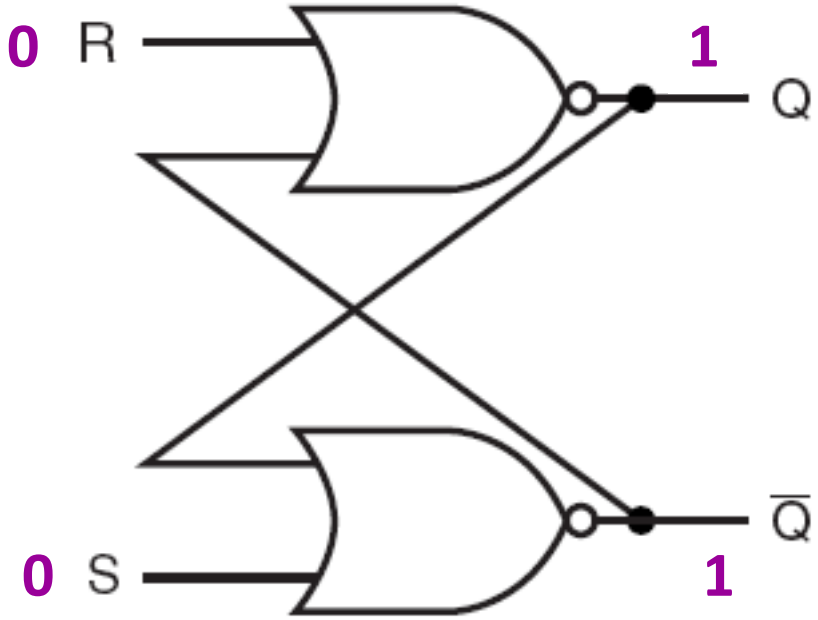
- Wenn **R=S=0** und **Q war 1**, wird Q gehalten

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0



SR-Latch, verbotene Inputs

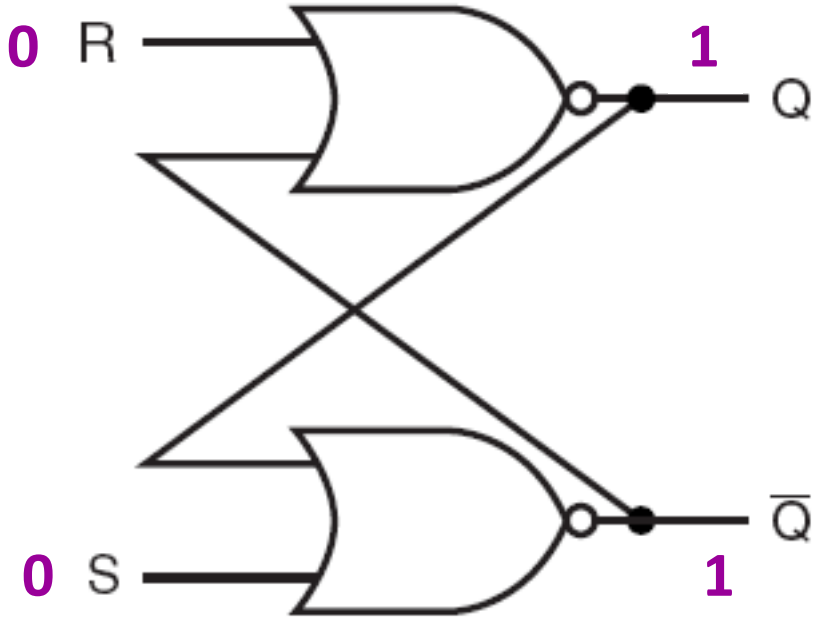
- Was passiert wenn **S=R=1**? → verbotener Input
- Q und $\overline{Q}$ werden beide 0. Was heißt das? → stabiler Zustand
  → verbotener Zustand
- Was passiert wenn dann **S=R=0 wechselt**? → Zustand nicht stabil



a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

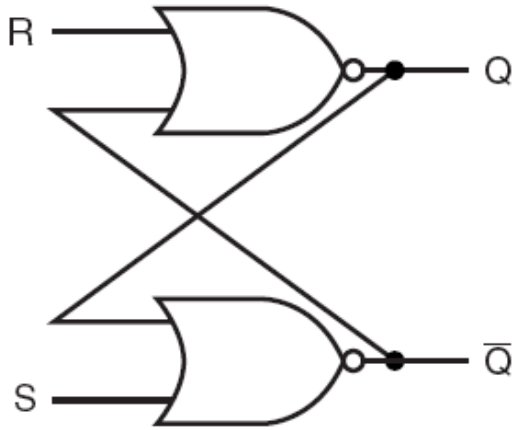
SR-Latch, verbotene Inputs

- Problem, falls wir zum Haltezustand zurück wollen
  - **Output oszilliert zwischen 0 und 1**
  - **oder Outputs fest auf 1 oder 0,** abhängig von Signallaufzeit der Gatter



a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

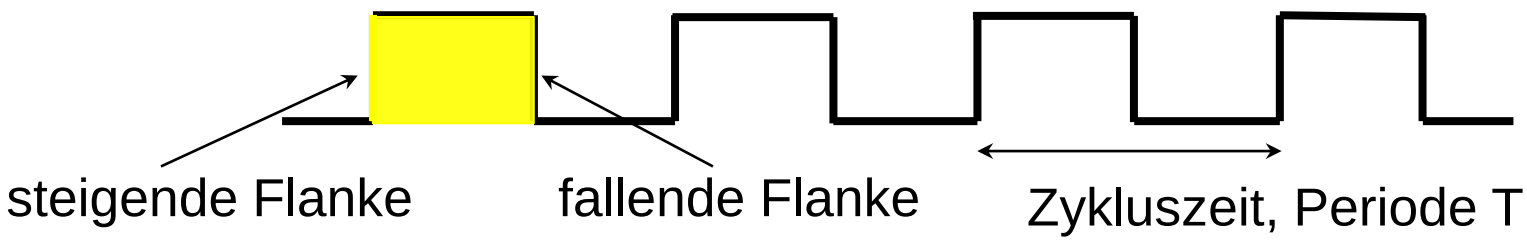
Wahrheitstabelle SR-Latch



S	R	Q	
0	0	Q	Halten
0	1	0	Wechsel
1	0	1	
1	1	?	Verboten

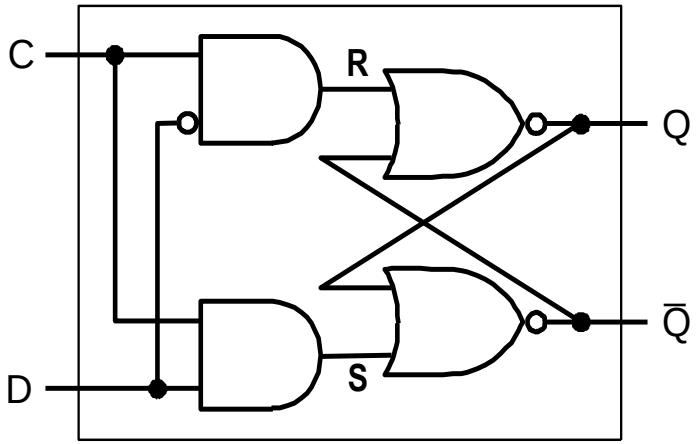
Latches und Flip-Flops

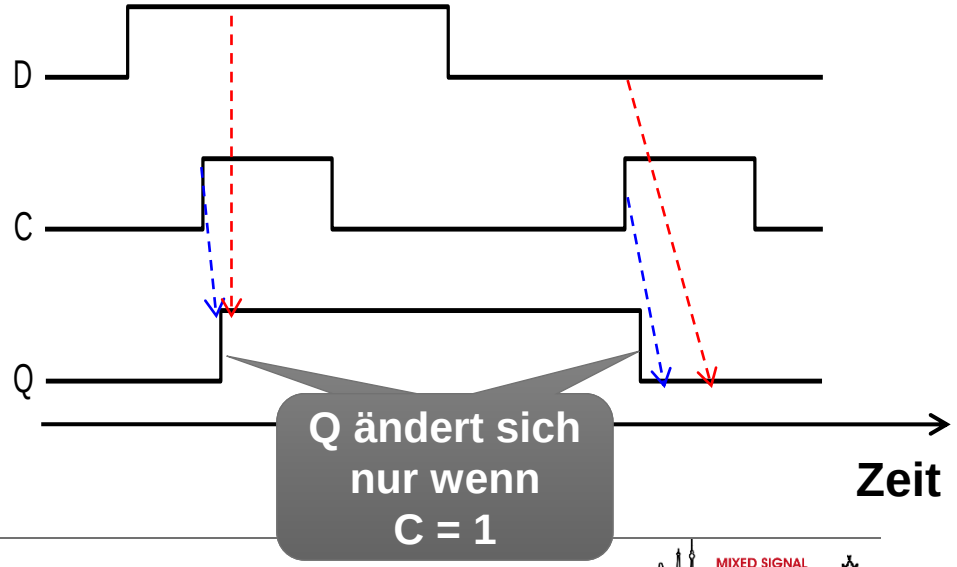
- Zustandswechsel (Wert) abhängig vom Takt (*Clock*)
- Latch: Zustandswechsel, immer wenn der Input wechselt und Clock = „hoch“/„an“/1 ist
- Flip-Flop: Zustandwechsel nur an Taktflanken
  - Taktflankengesteuertes Verfahren (*edge-triggered methodology*)
- *Clocking methodology* (*Taktverfahren*) definiert, wann die Signale geschrieben werden
  - legt fest, was gelesen wird wenn ein Signal zur „gleichen Zeit“ gelesen und geschrieben wird



D-Latch

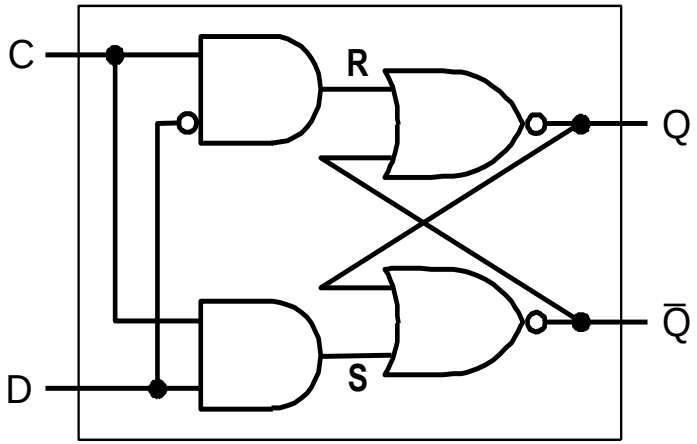
- Zwei Inputs + 2 NAND Gatter + INV:
  - Datenwert (D) ist das Datum das gespeichert werden soll
  - Taktsignal (C): C=**1**: Eingang D **überschreibt** Ausgang Q
    C=1, D=1 → S=1, R=0 → Q=1
    C=**0**: Ausgang Q wird **gehalten,** → S=0, R=0
  - Wert des internen Zustands (Q) und dessen Komplement
- Wahrheitstabelle reicht nicht aus, um Verhalten zu definieren

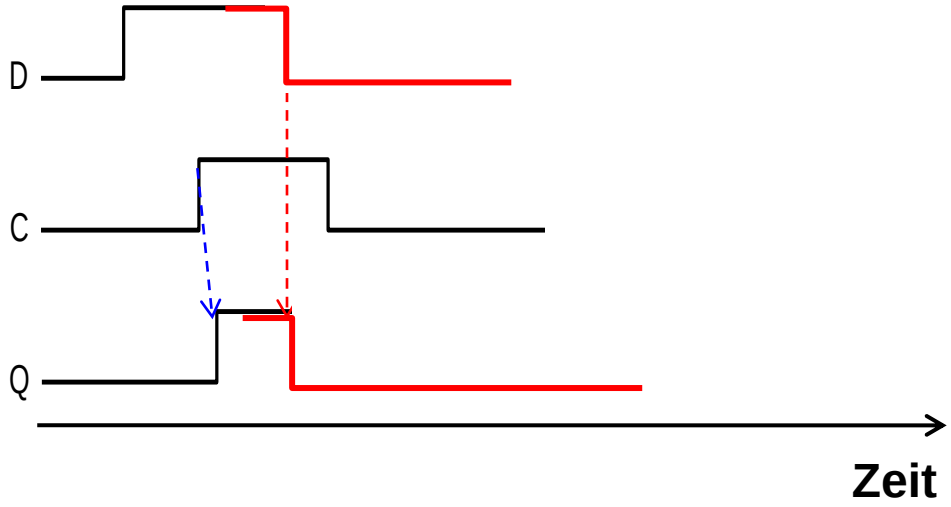




D-Latch

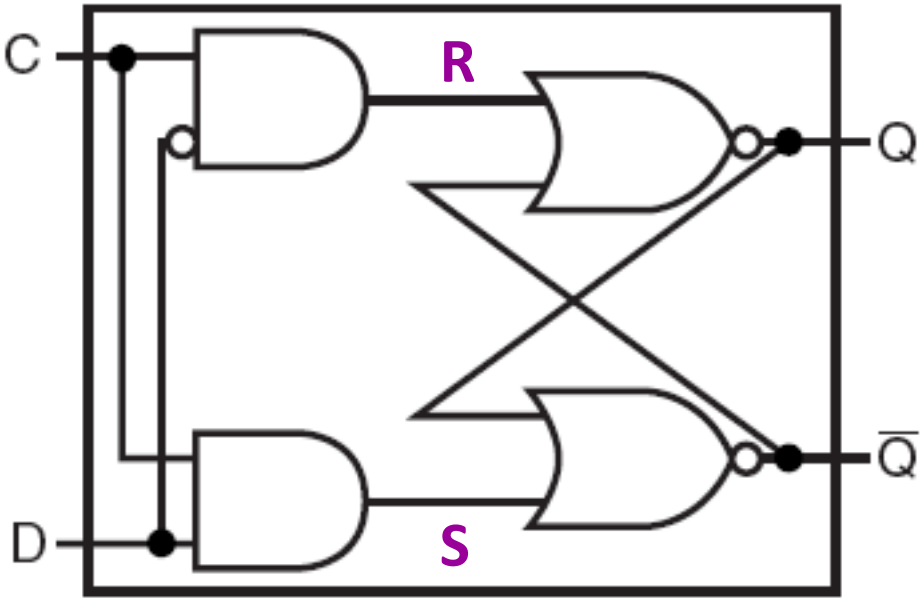
- Zwei Inputs + 2 NAND Gatter + INV:
  - Datenwert (D) ist das Datum das gespeichert werden soll
  - Taktsignal (C): C=**1**: Eingang D **überschreibt** Ausgang Q
    C=1, D=1 → S=1, R=0 → Q=1
    C=**0**: Ausgang Q wird **gehalten,** → S=0, R=0
  - Wert des internen Zustands (Q) und dessen Komplement
- Achtung: Wenn C=1 ist das Latch transparent!





Verhalten D-Latch

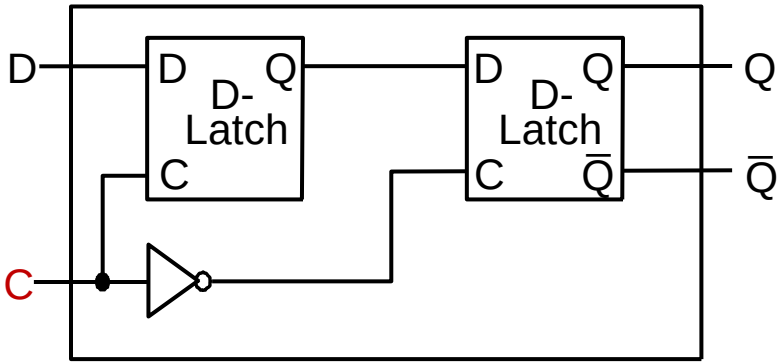
- Clock = C = 1 → Übernahme von D auf den Ausgang Q
- Clock = C = 0 → Speicher vom vorherigen Zustand, D ist don‘t care

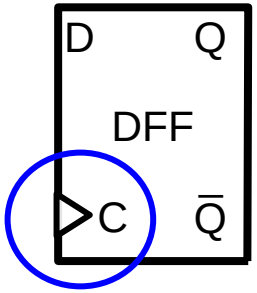


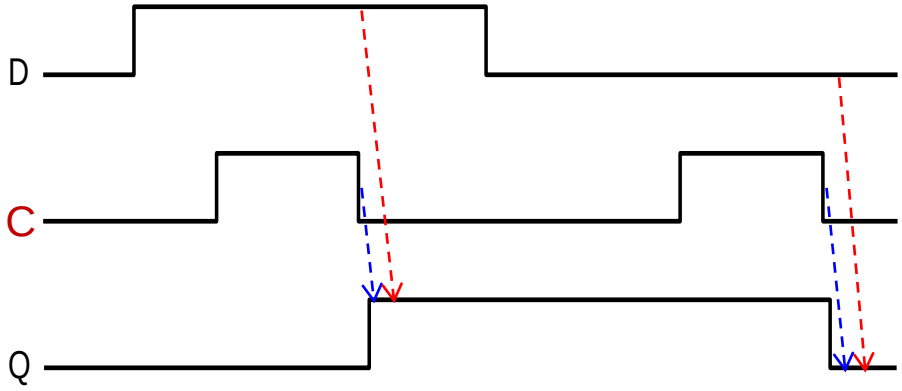
C	D	R	S	Q(t)
0	0	0	0	Q(t-1)
0	1	0	0	Q(t-1)
1	0	1	0	0
1	1	0	1	1

D-Flip-Flop

- Output wechselt nur bei fallender Taktflanke von C
  - DFF besteht aus 2 latches deren Clock (C) invertiert zueinander ist

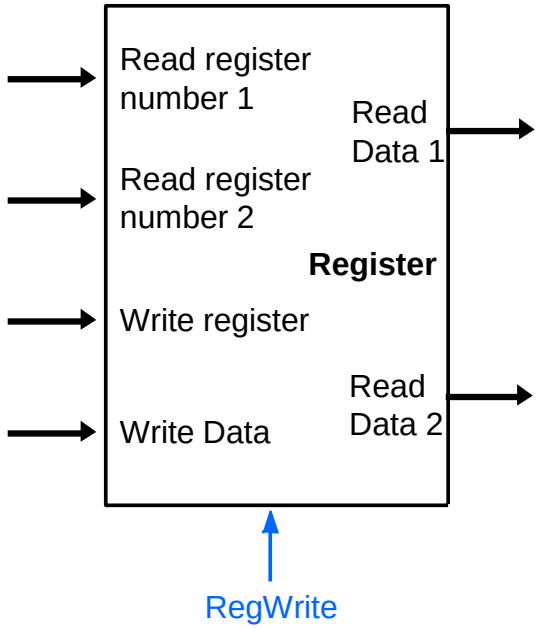






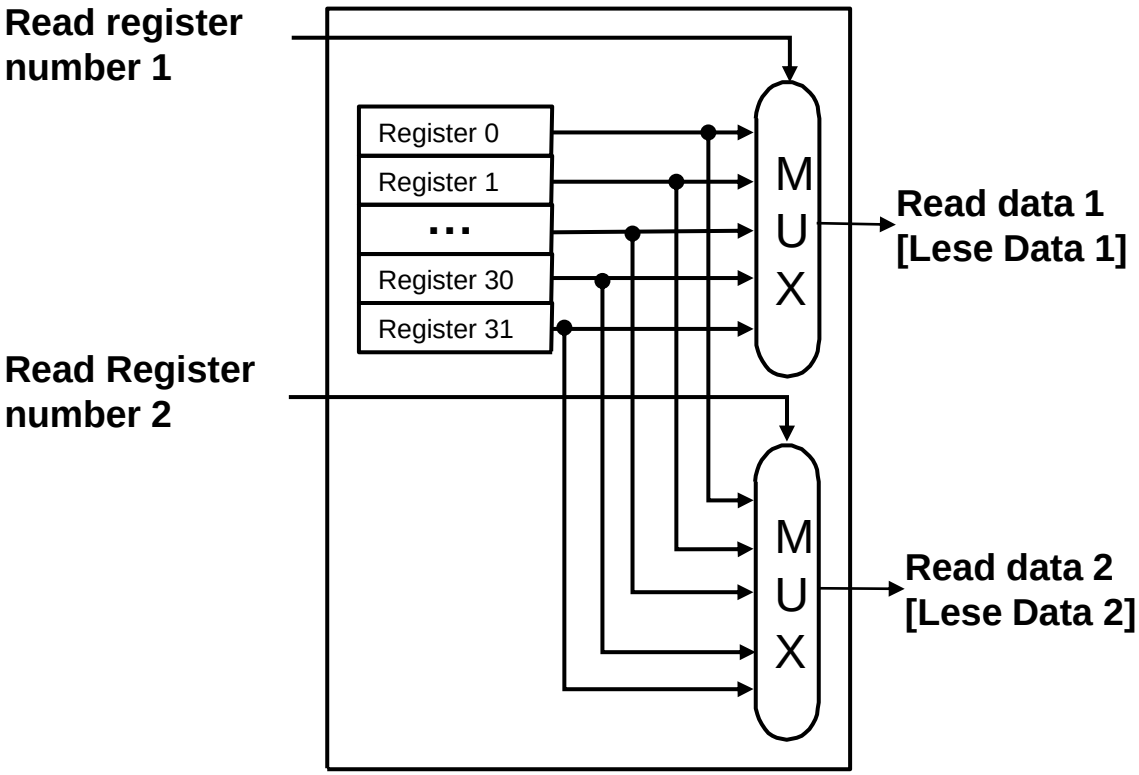
Register

- Unser Prozessor verfügt über einen Registersatz, ~datei oder Registerspeicher (*register file*) von 32 32-Bit breiten Register.
- *Wikipedia*: Als Register bezeichnet man Speicherbereiche, die innerhalb eines Prozessorkerns direkt mit der eigentlichen Recheneinheit verbunden sind und die unmittelbaren Operanden und Ergebnisse aller Berechnungen aufnehmen.
  - Register sind in der Regel genau so breit/groß wie die Wortbreite des Prozessorkerns (8, 16, 32 oder 64 Bit).



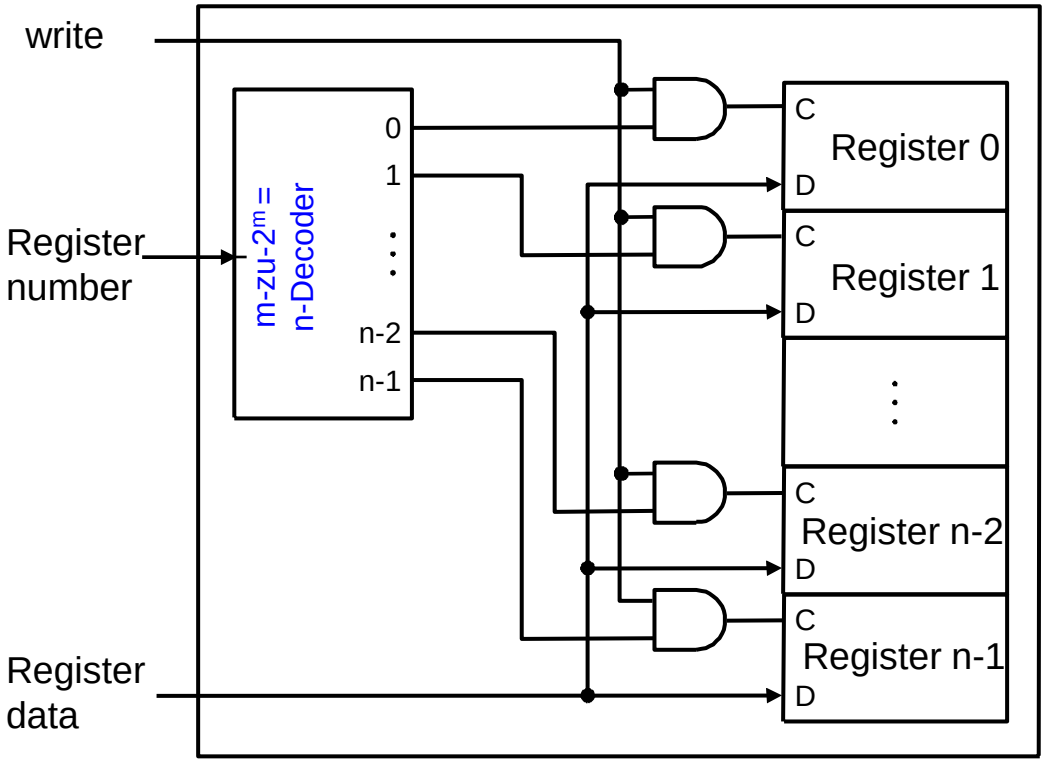
Implementierung des Registersatzes

- Jedes Register besteht aus 32 D-Flip-Flops (32 x 32 DFFs)
- Implementierung der Leseports:



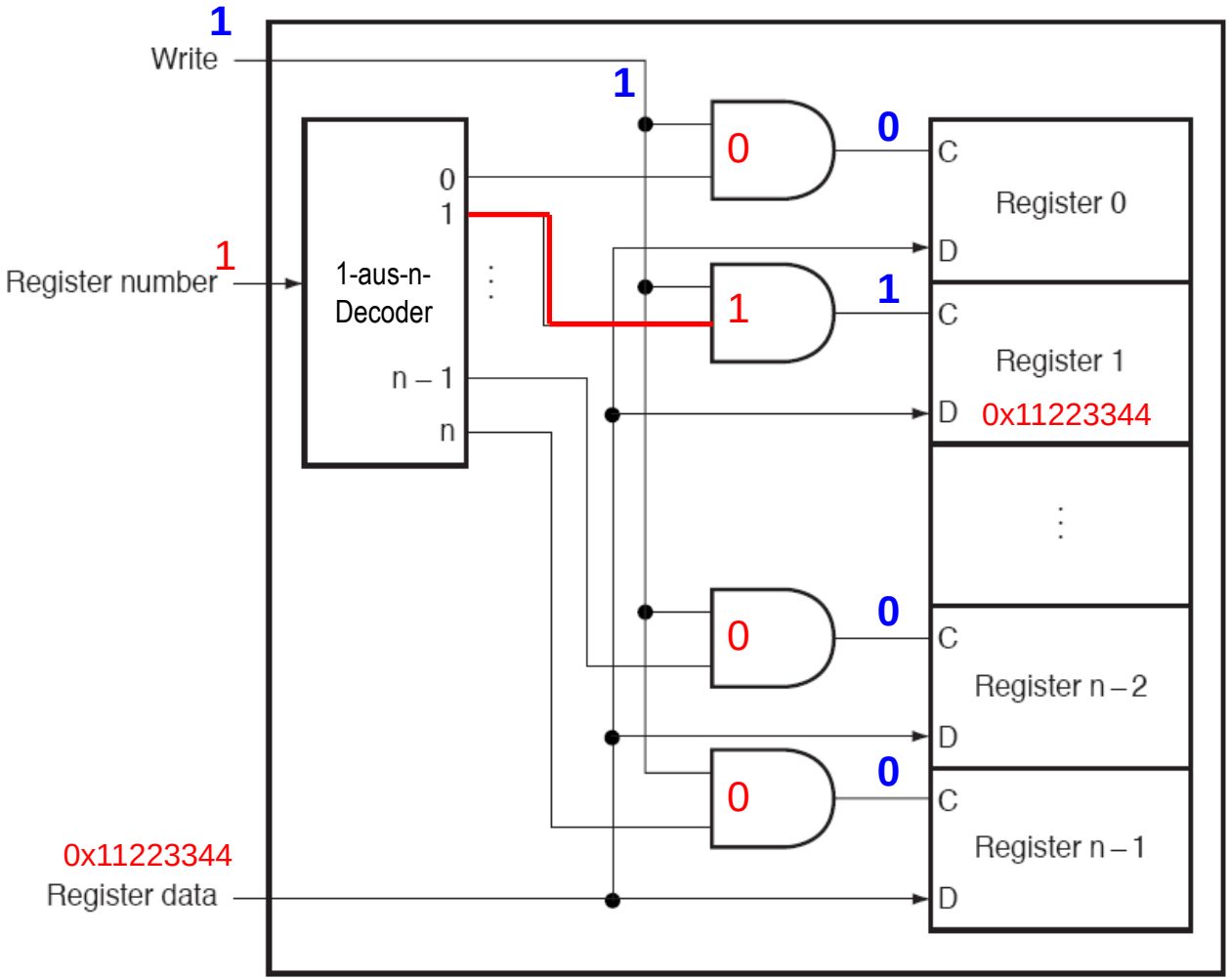
Implementierung des Registersatzes

- Implementierung des Schreibports (write):



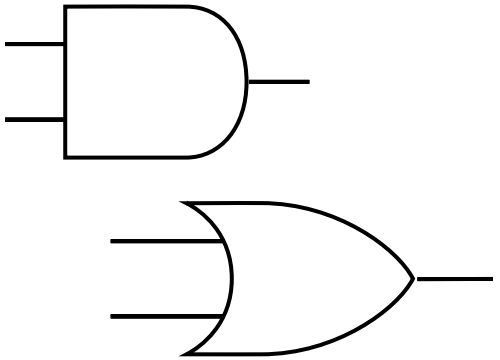
- Takt (clock) wird nicht angezeigt, wird aber trotzdem verwendet, um zu bestimmen, **wann** geschrieben wird

Schreibaktion

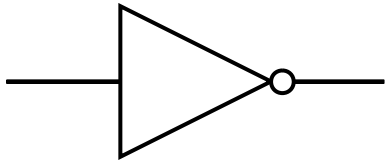


Zusammenfassung

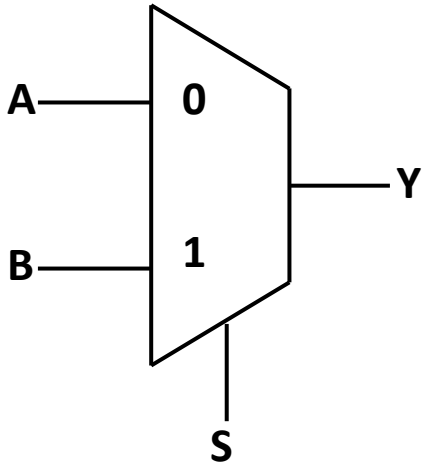
- Die logischen Operatoren UND, ODER und NICHT bilden einen funktional vollständigen Operatorensatz.



- Eine Boolesche Funktion kann mithilfe einer Wahrheitstabelle spezifiziert werden.



- Ein Multiplexer wählt je nach Steuersignal S einen Input A oder B aus.



Zusammenfassung

- Wir können eine ALU konstruieren, die die Operationen AND, OR, ADD, SUB, NAND, NOR unterstützt.
- Die Outputs eines Schaltnetzes hängen nur von aktuellen Inputs ab.
- Schaltwerke verfügen über internen Speicher.