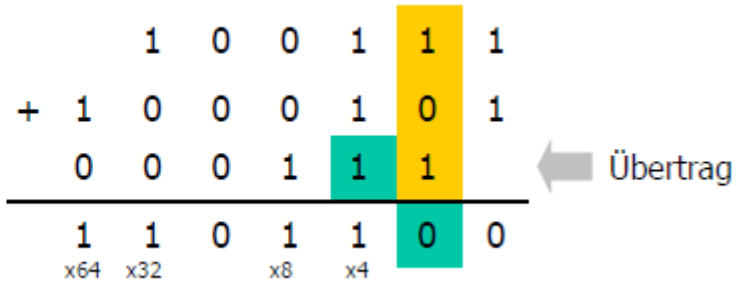
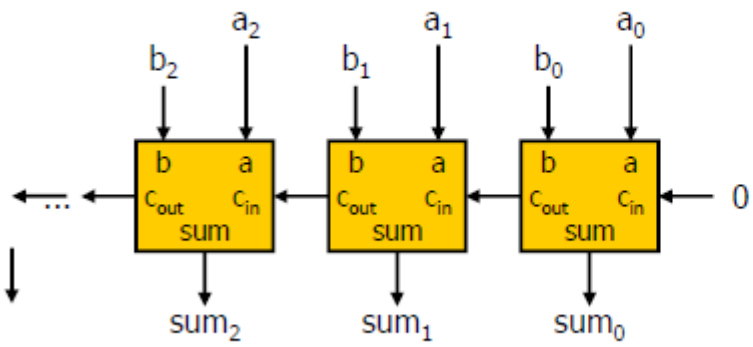
Vorlesung 7
Addition von Binärzahlen

Addition von Binärzahlen

- Die Addition erfolgt stellenweise wie bei Dezimalzahlen mit einem **Übertrag (carry)**:
- In jeder Stufe werden also aus den **3 Eingängen** a, b, cin die **Ausgänge** sum und cout erzeugt.
- Man nennt diesen wichtigen Schaltungsblock den **Volladdierer** (full adder, FA):



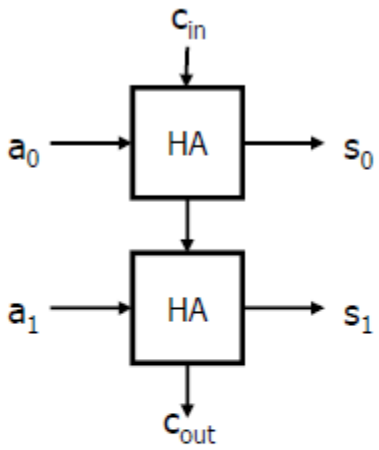
- Die Addition erfolgt stellenweise wie bei Dezimalzahlen mit einem **Übertrag (carry)**:
- In jeder Stufe werden also aus den **3 Eingängen** a, b, cin die **Ausgänge** sum und cout erzeugt.
- Man nennt diesen wichtigen Schaltungsblock den **Volladdierer** (full adder, FA):



c_{in}	b	a	c_{out}	sum
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

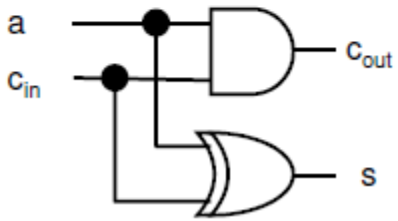
Halbaddierer

- Manchmal (z.B. in Zählern) muss NUR der Übertrag addiert werden.
- Der Addierer hat daher nur **einen** Dateneingang und einen Carry Eingang.
- Man nennt diesen Block einen Halbaddierer (Half-Adder, HA)



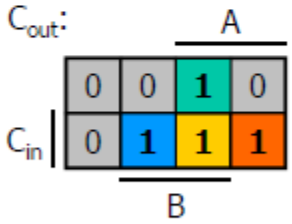
c_{in}	a	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$c_{out} = a \cdot c_{in}$$
$$s = a \oplus c_{in}$$

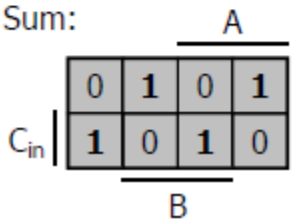


- …

C_{in}	A	B	C_{out}	S	$!C_{out}$
0	0	0	0	0	1
0	0	1	0	**1**	1
0	1	0	0	**1**	1
0	1	1	**1**	0	0
1	0	0	0	**1**	1
1	0	1	**1**	0	0
1	1	0	**1**	0	0
1	1	1	**1**	**1**	0

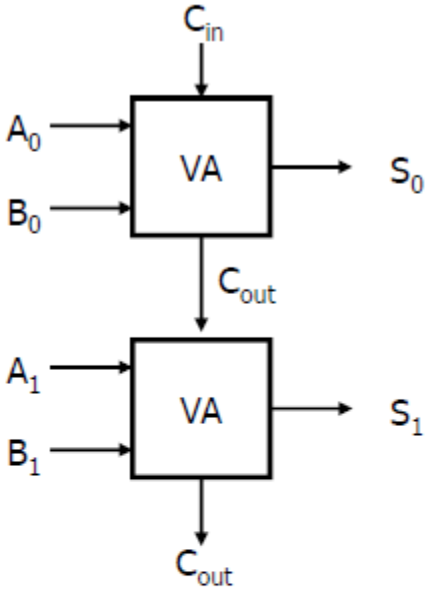


$C_{out} = AB + BC_{in} + AC_{in}$
$= AB + (A+B)\,C_{in}$



$S = A \oplus B \oplus C_{in}$
$= ABC_{in} + (A + B + C_{in}) \cdot !C_{out}$

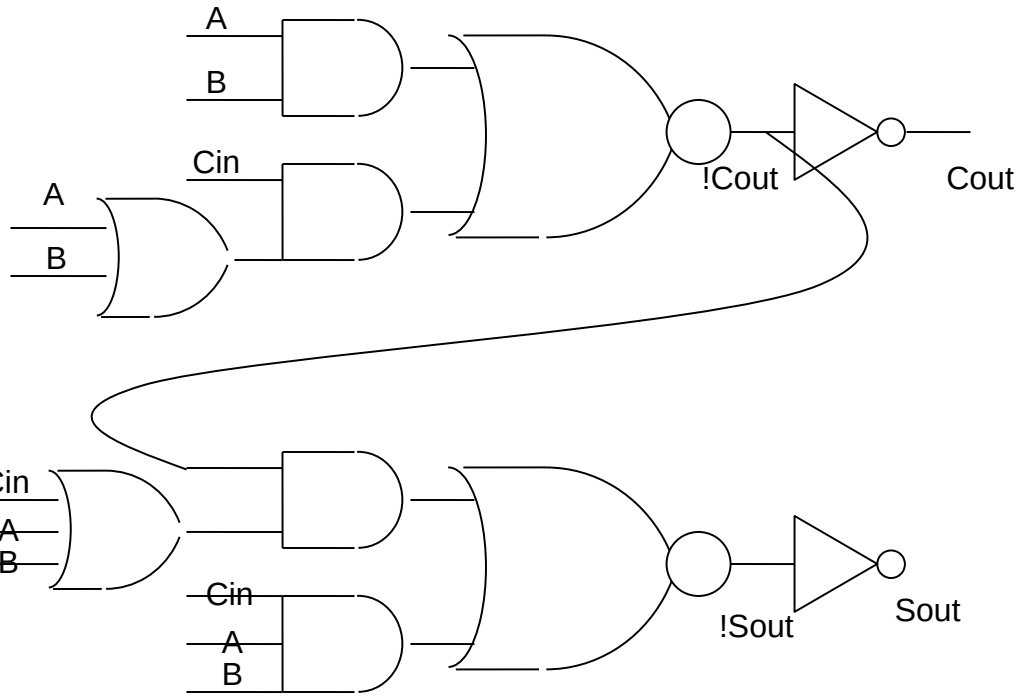
Trick: Carry-Ergebnis wird mitverwendet:
Mehrere Ebenen logische Tiefe:
'Multiple Output Minimization' (MOM).



Der Carry-Pfad muß optimiert werden, da das Carry durch alle N Bit 'rippeln' muß

- …

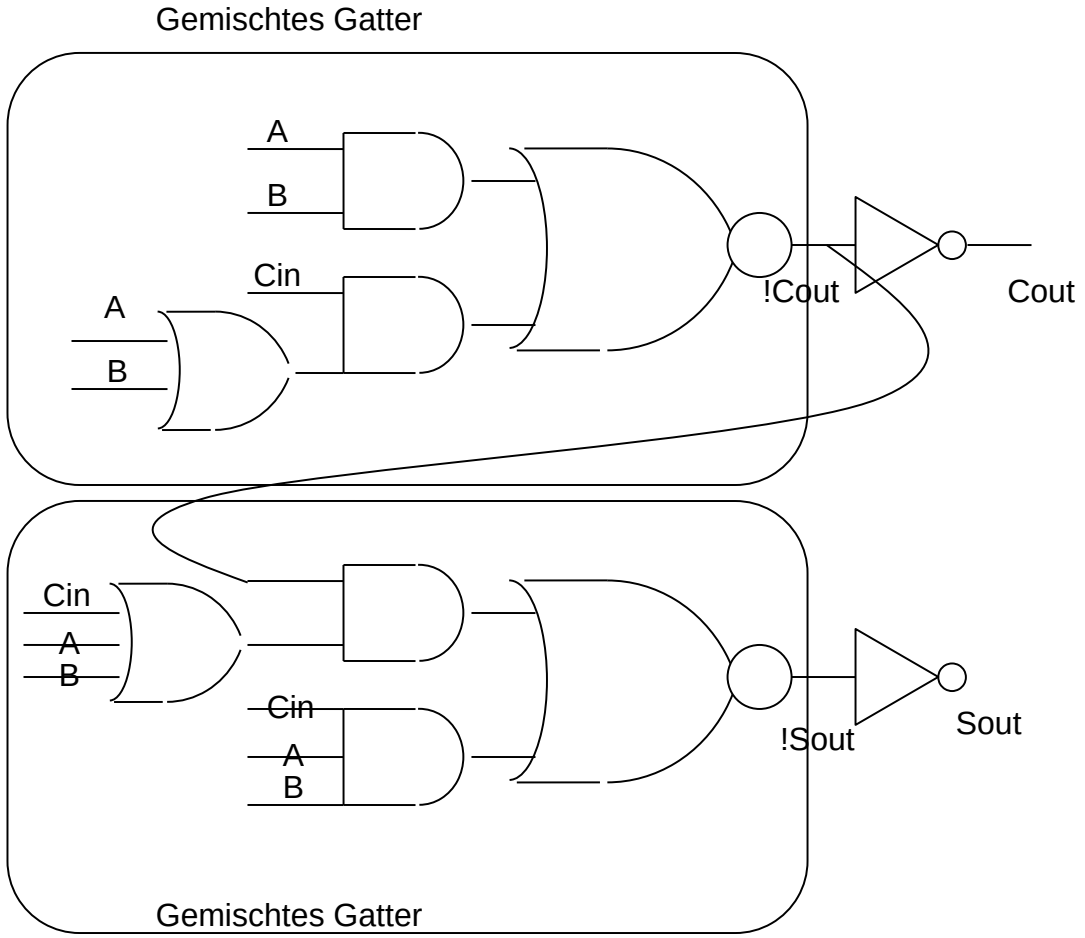
$C_{out} = AB + (A+B)\,C_{in}$:



$S = ABC_{in} + (A + B + C_{in}) \cdot !C_{out}$:

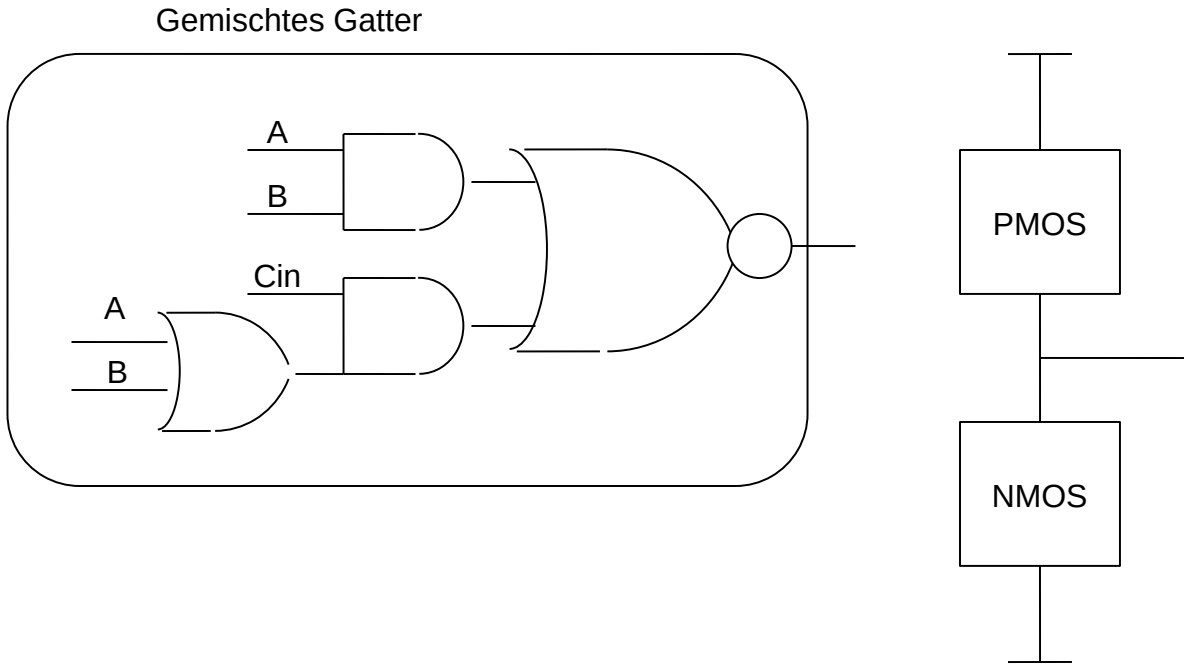
- …

$C_{out} = AB + (A+B)\,C_{in}$:

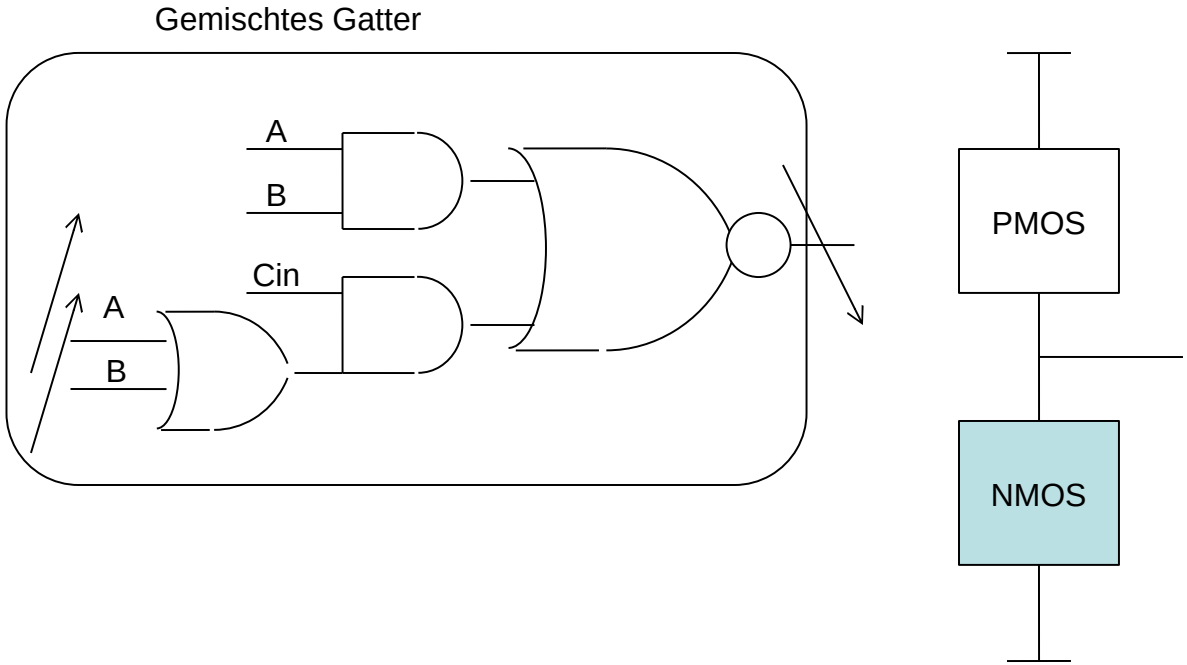


$S = ABC_{in} + (A + B + C_{in}) \cdot !C_{out}$:

- Versuchen wir ein gemischtes Gatter für !Cout herzuleiten

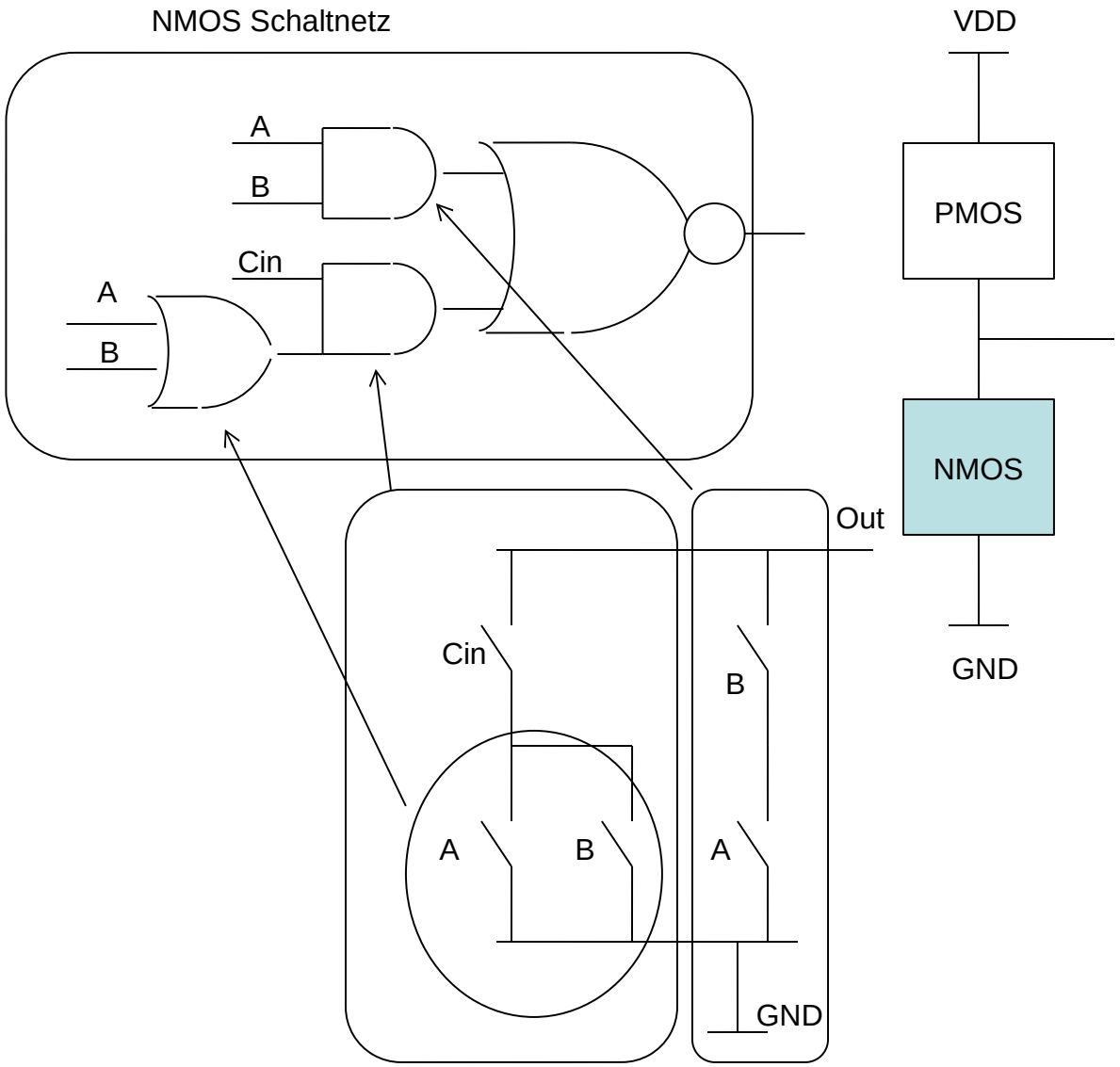


- Vorbereitung: alle Eingänge müssen positiv sein, eine Negation am Ausgang

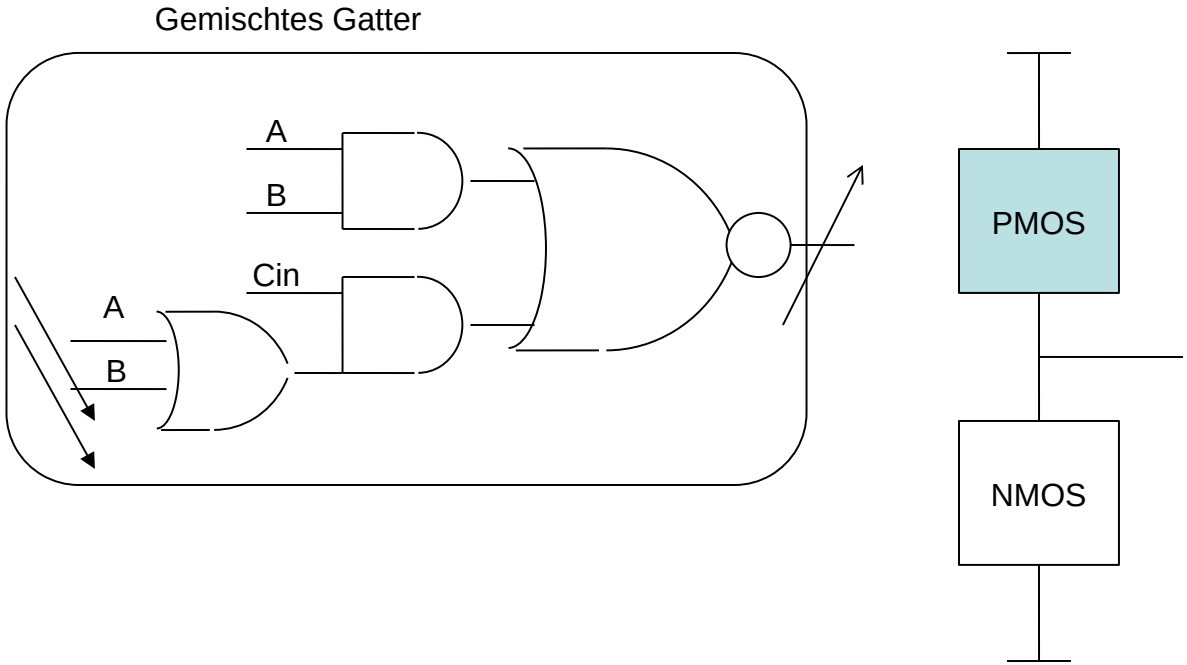


- UND wird durch Serien- und ODER durch Parallelschaltung ersetzt

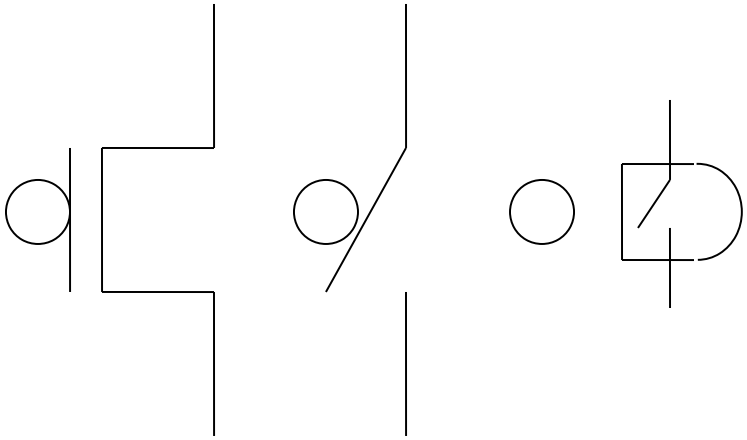
- NMOS Schaltnetz leitet für eine Kombination von positiven Eingängen

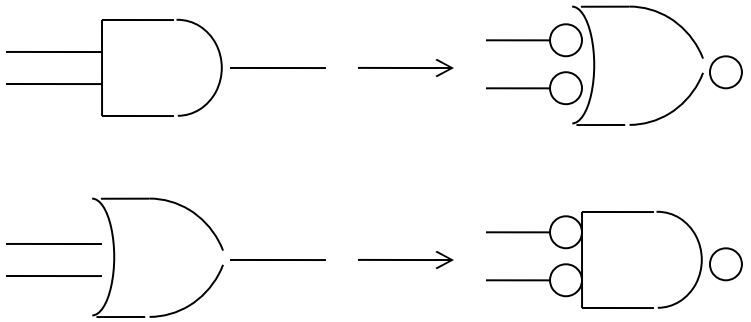


- PMOS Teil: erzeugt logische 1 am Ausgang für eine Kombination von negativen Eingängen

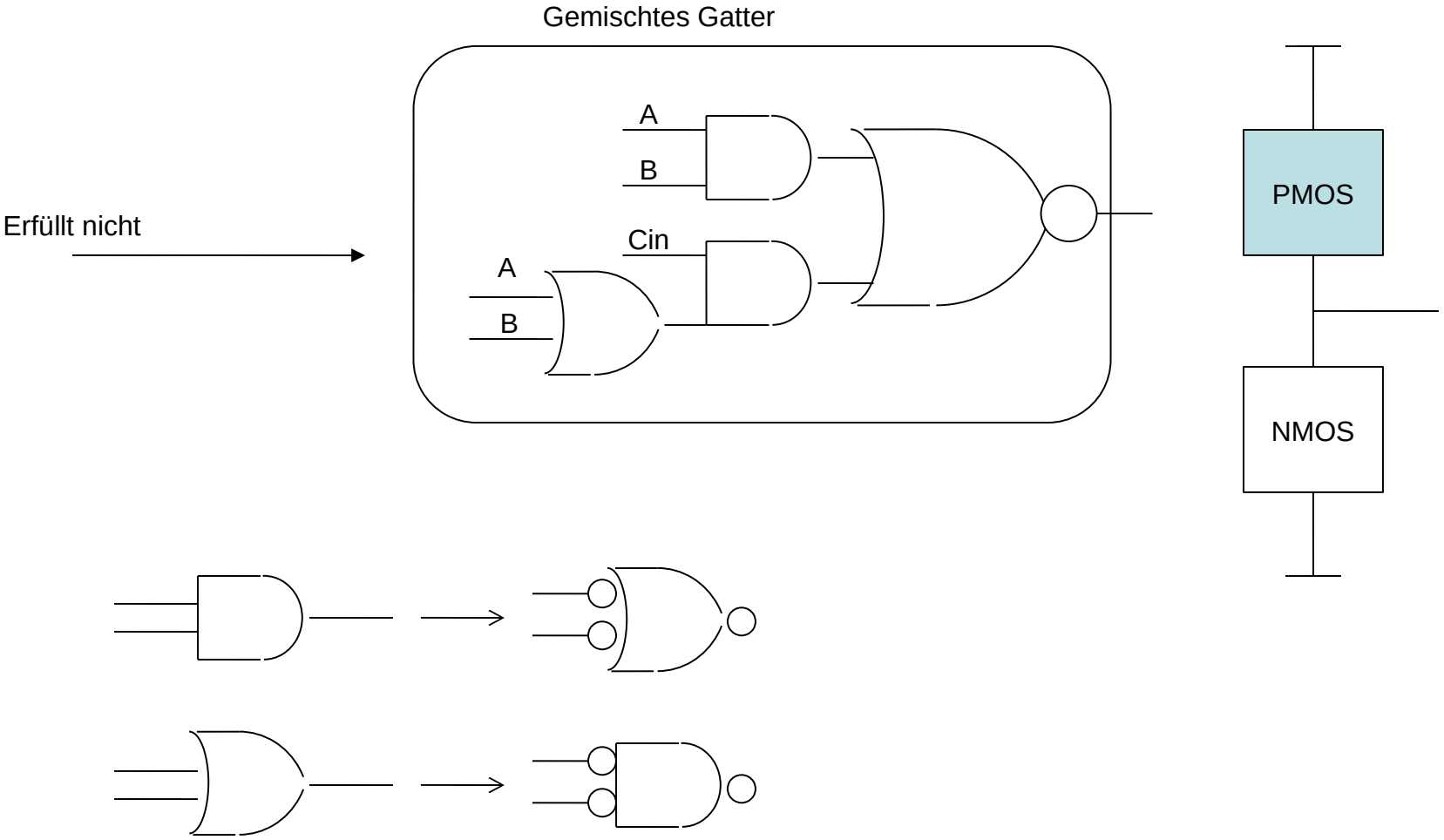


- PMOS: Kombination vom Schalter und Inverter am Eingang
- Es ist einfacher PMOS Teil eines Logikgatters herzuleiten wenn die Eingangssignale vor dem Gatter negiert werden
- Dafür verwenden wir De Morgansche Regeln
- Ausgang soll nicht-negiert sein

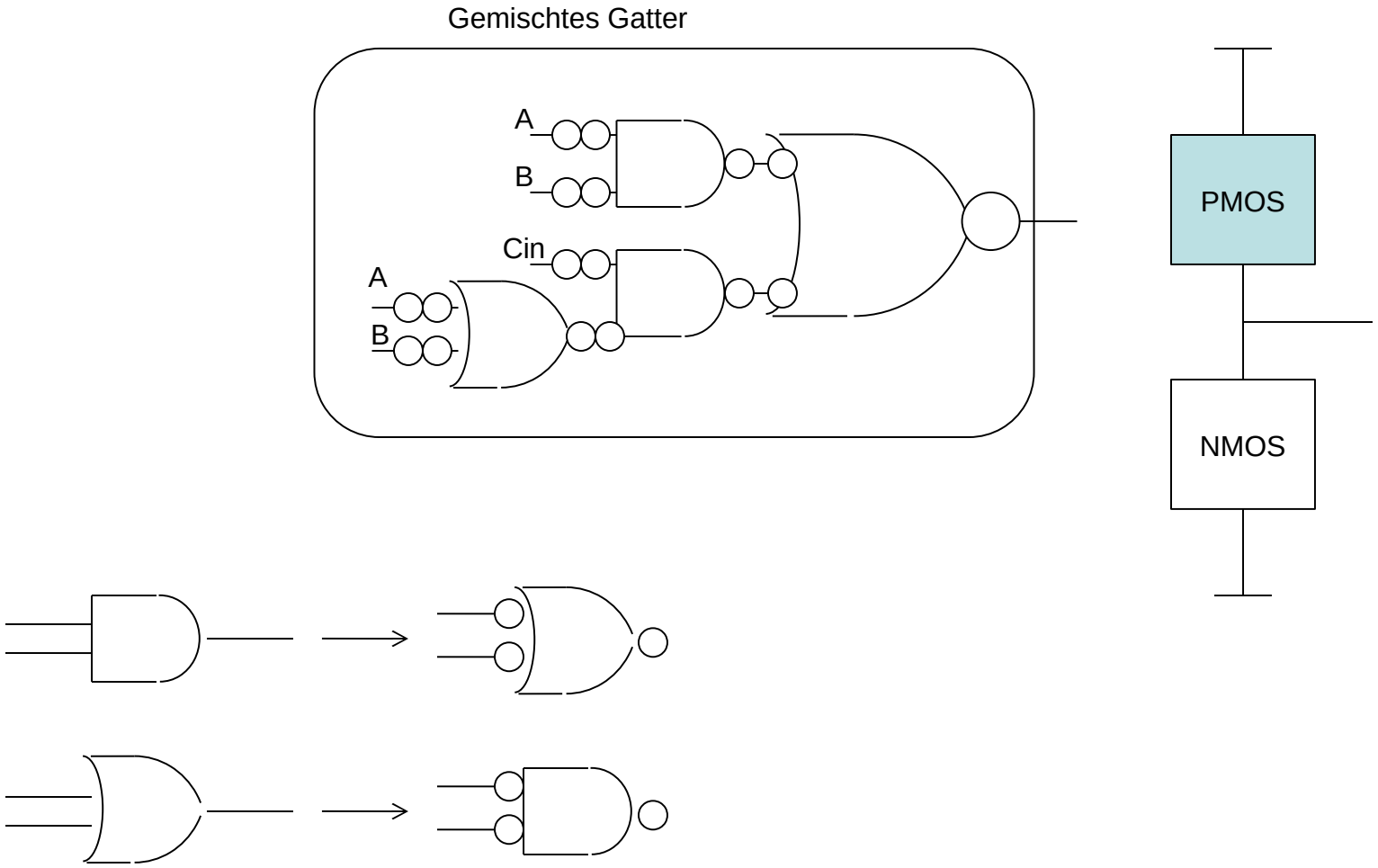




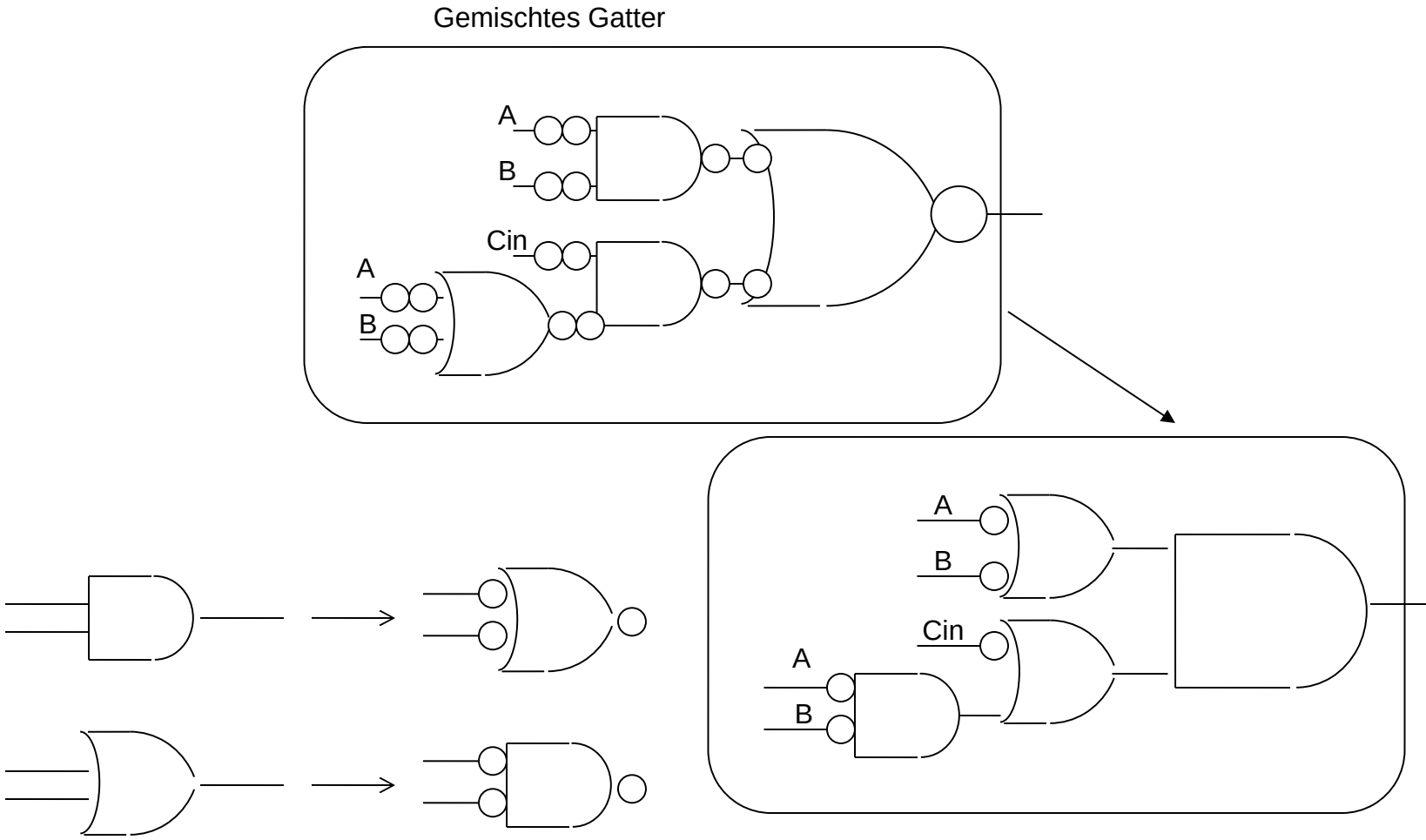
- Vorbereitung: alle Eingänge müssen positiv sein, eine Negation am Ausgang



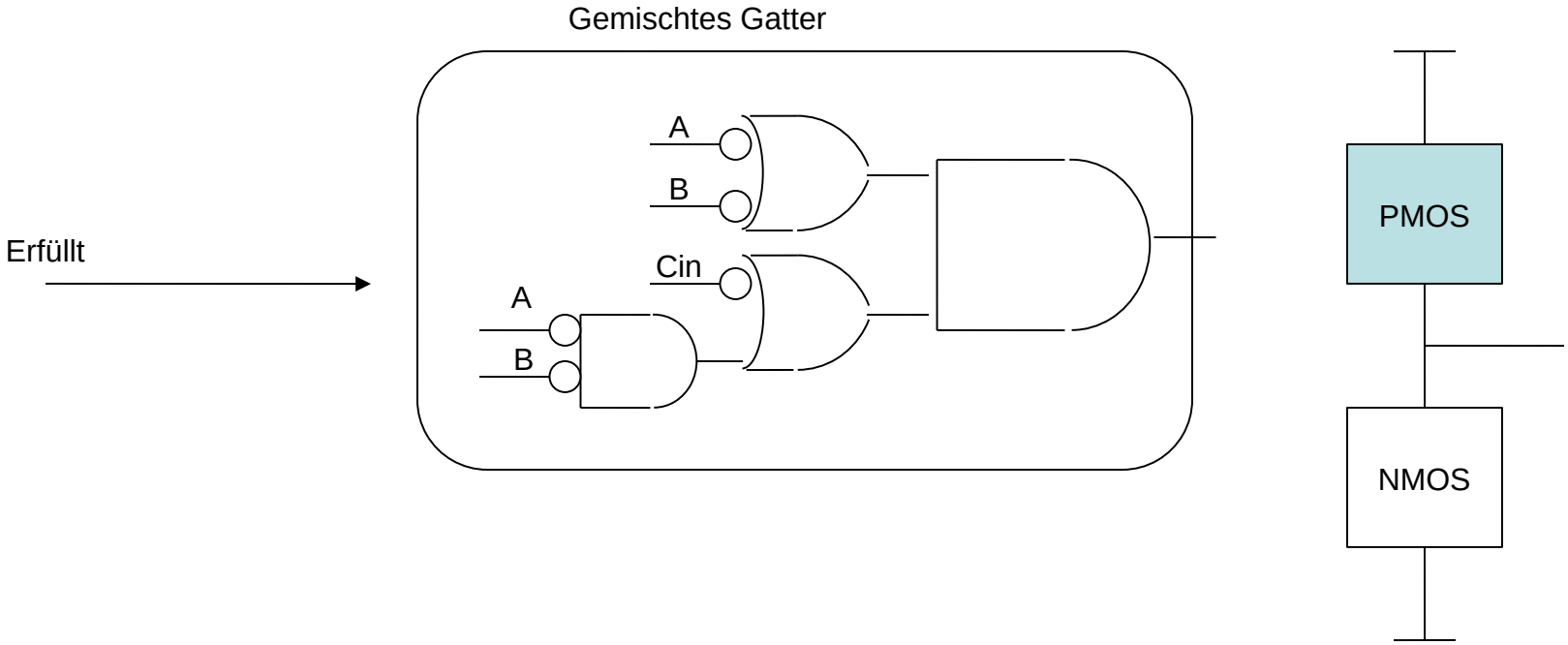
- Erster Schritt: doppelte Negation



- Zweiter Schritt: De Morgansche Relegn

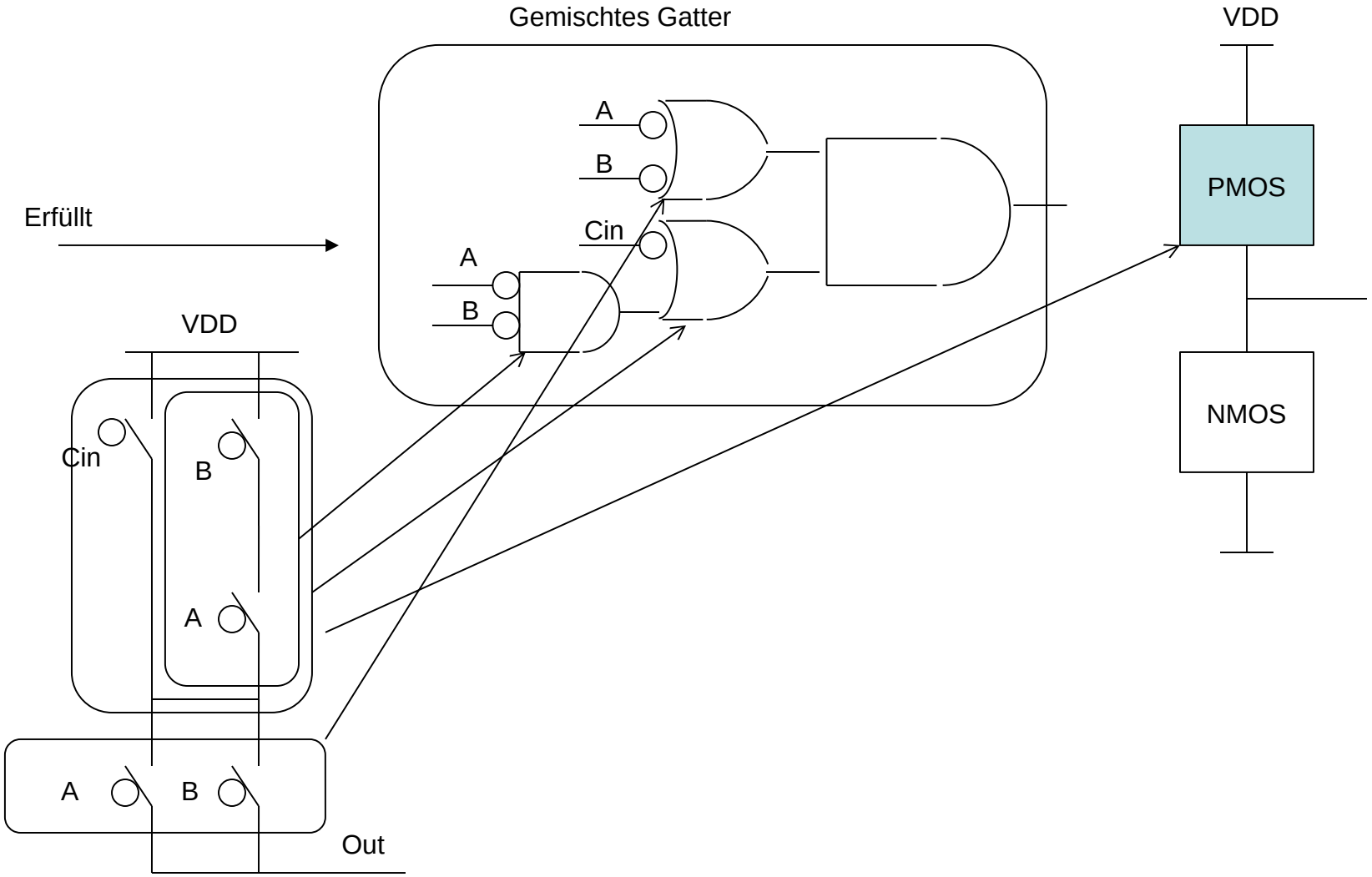


- Vorbereitung: alle Eingänge müssen positiv sein, eine Negation am Ausgang

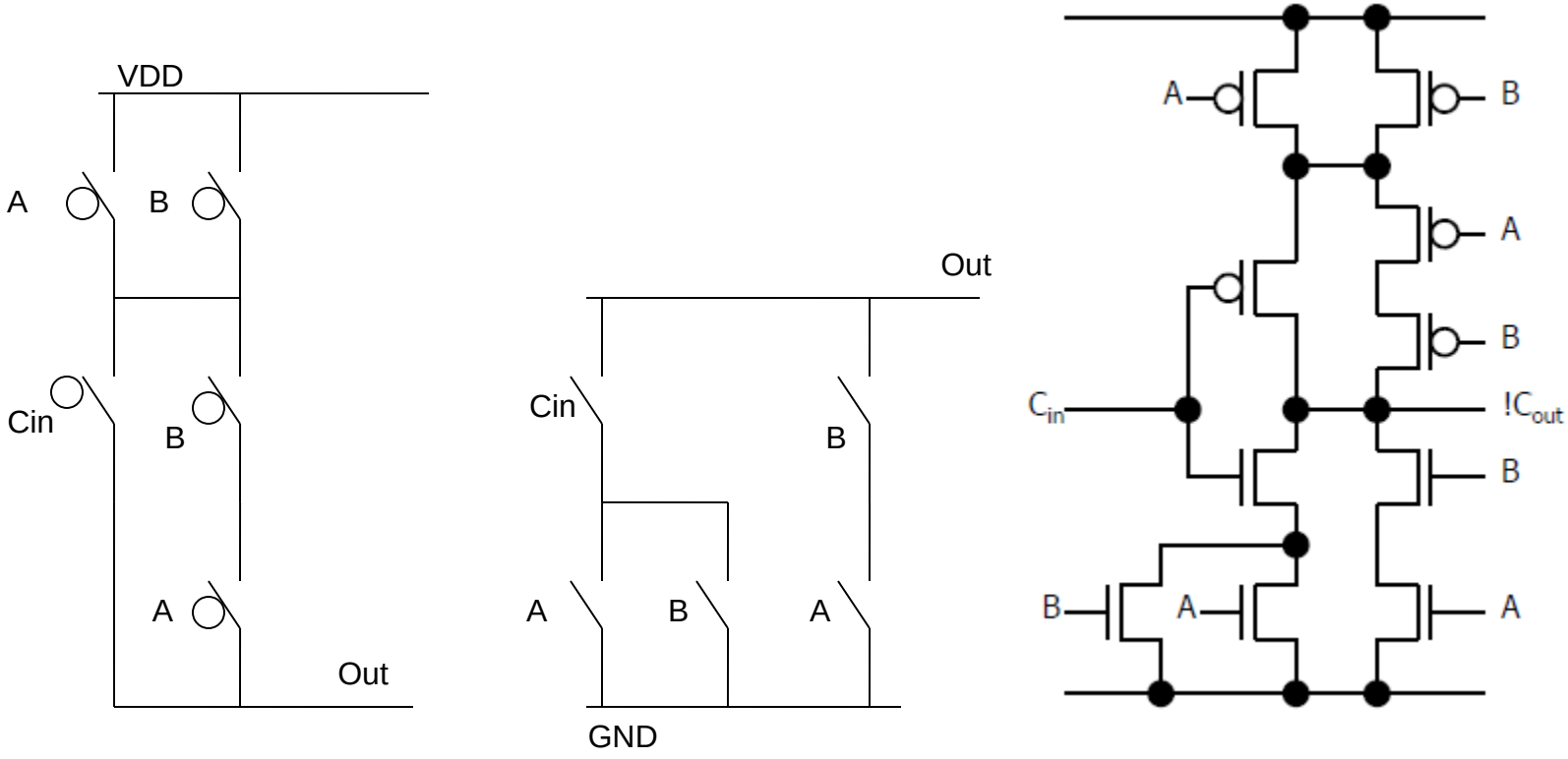


- UND (mit negierten Eingängen) wird durch Serien- und ODER durch Parallelschaltung ersetzt

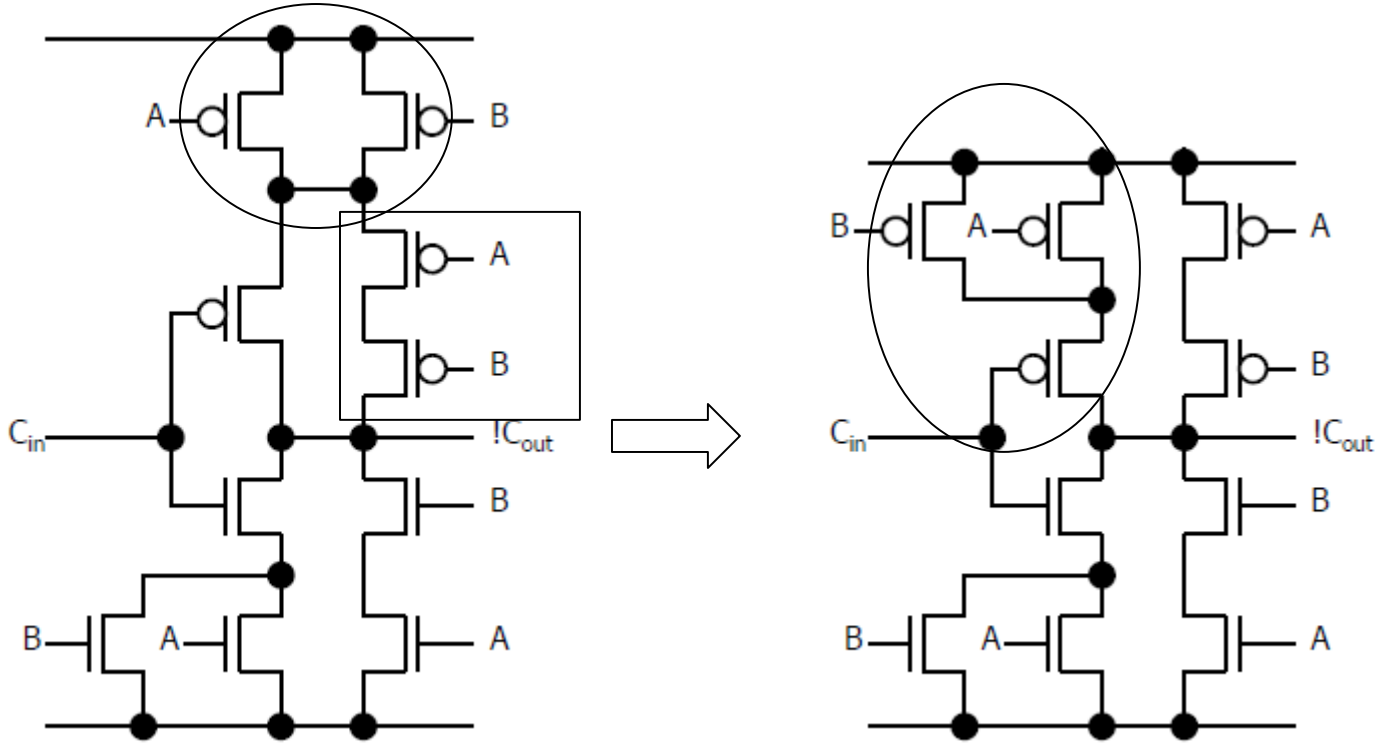
- Vorbereitung: alle Eingänge müssen positiv sein, eine Negation am Ausgang



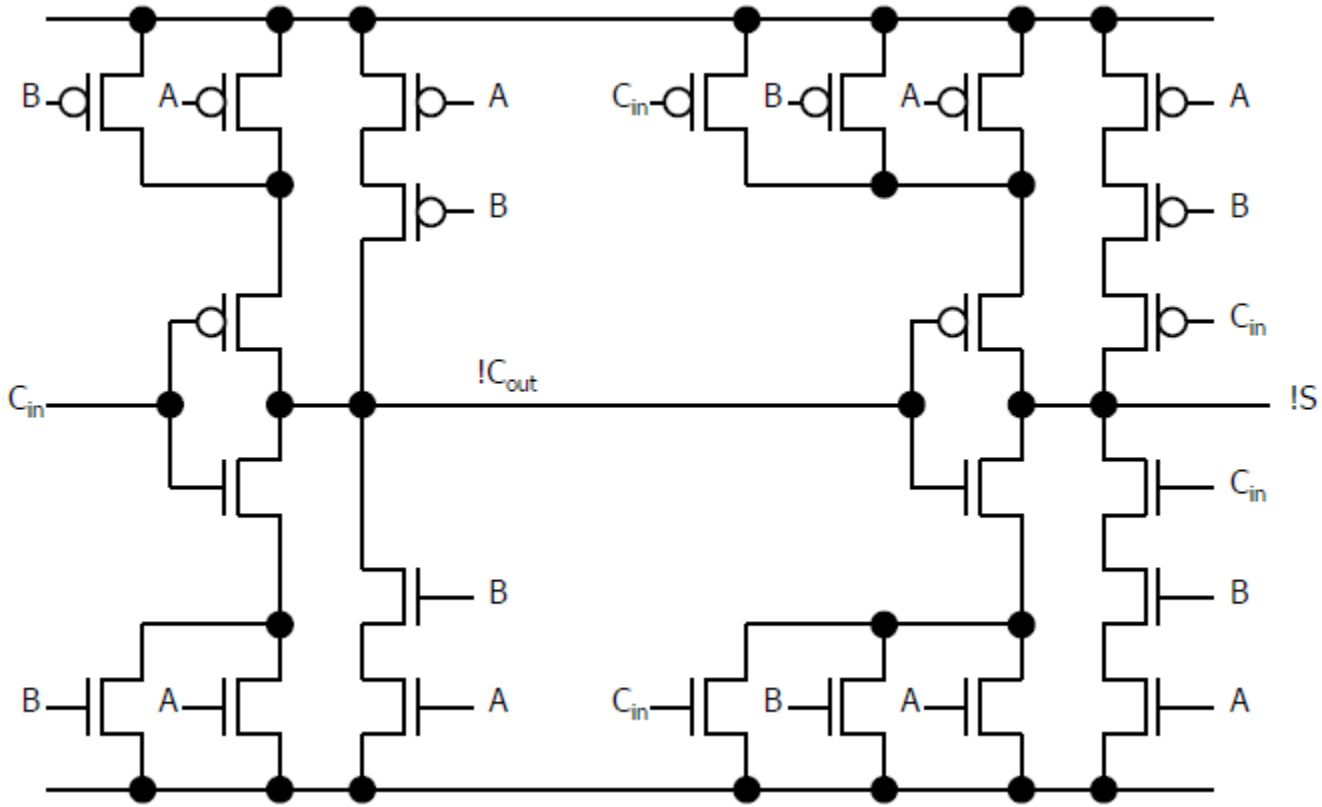
- Das Carry wird durch Cout = AB + (A+B)Cin gegeben.
- Diese Funktion kann mit dem gemischten Gatter für !Cout und eine Inverter implementiert werden
- Problem: 3 PMOS übereinander ('Stack height' = 3)



- Der PMOS Zweig kann umgeformt werden
- Idee: Wenn !A UND !B leitet (Schaltung im Rechteck) dann leitet !A ODER !B auch (Schaltung im Kreis)
- => Man kann !A UND !B an VDD anschliessen



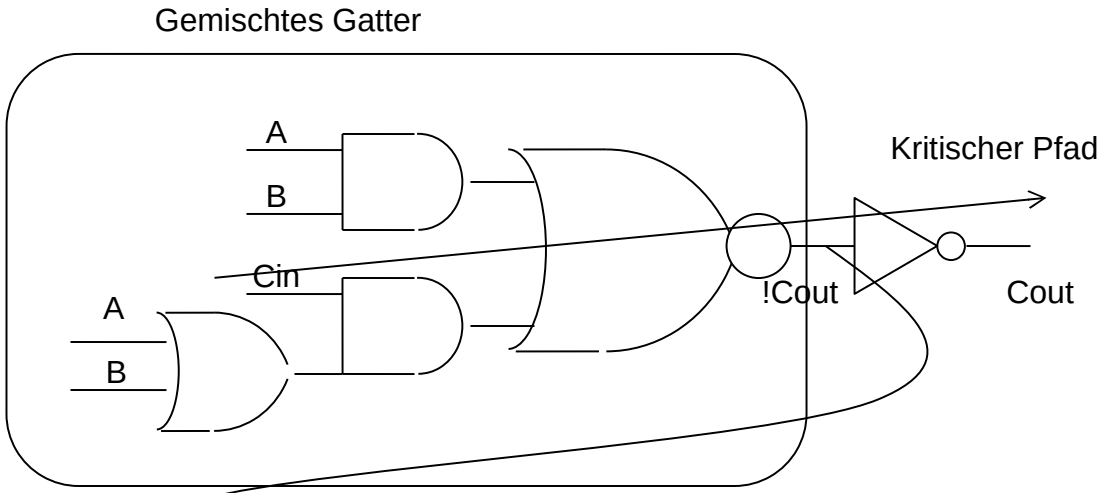
- -> Optimierter Volladierer



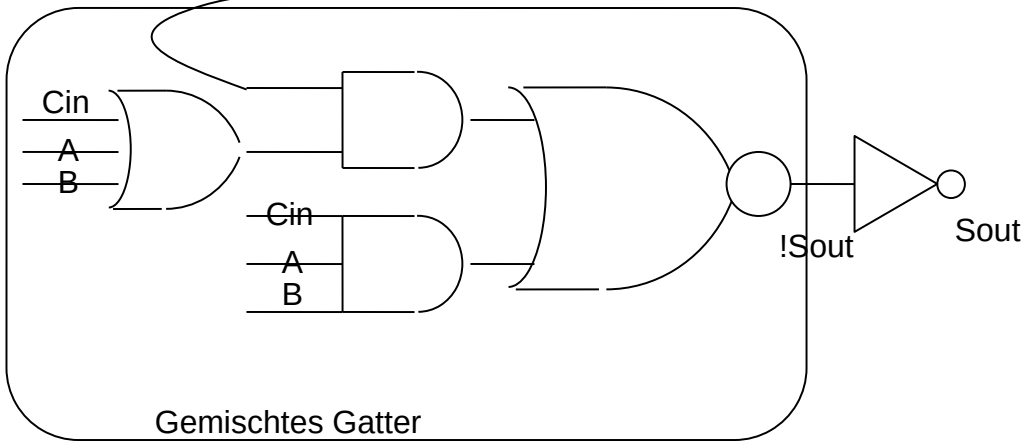
Optimierung: Polaritätswechsel

- Carry ist ein kritischer Pfad, da das Signal durch alle Bits läuft

$C_{out} = AB + (A+B)\,C_{in}$:



$S = ABC_{in} + (A + B + C_{in}) \cdot !C_{out}$:

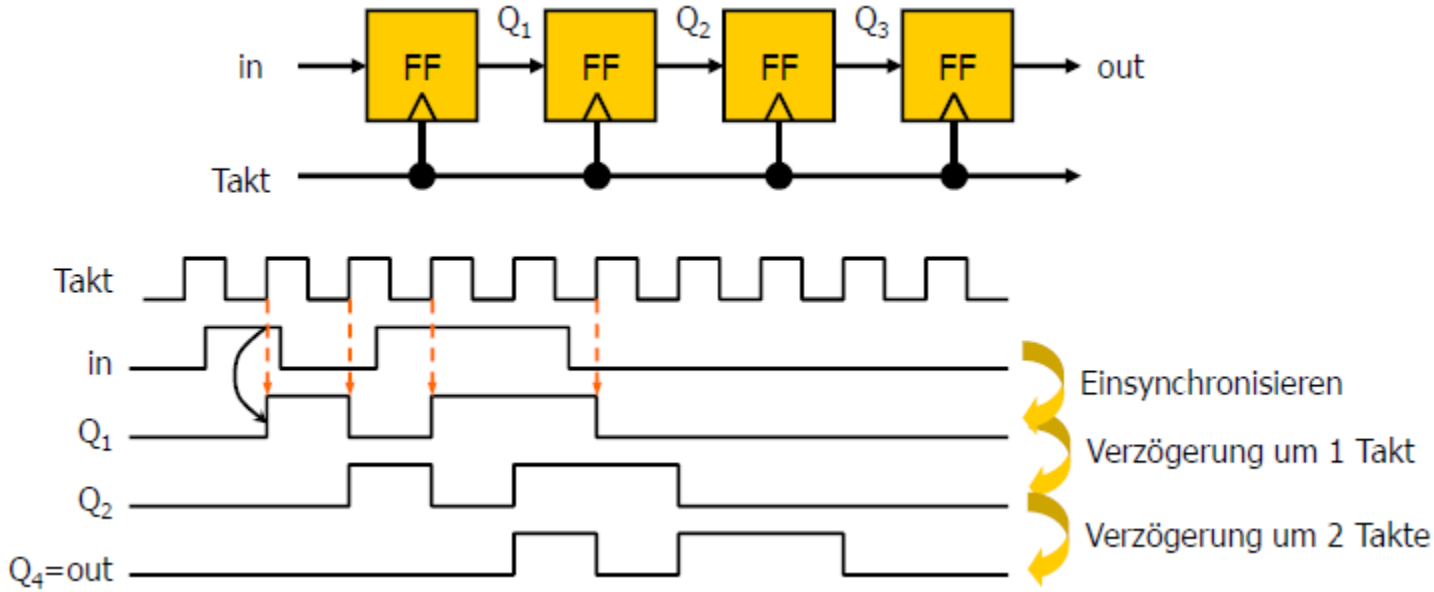


Getaktete Schaltungen

Schieberegister

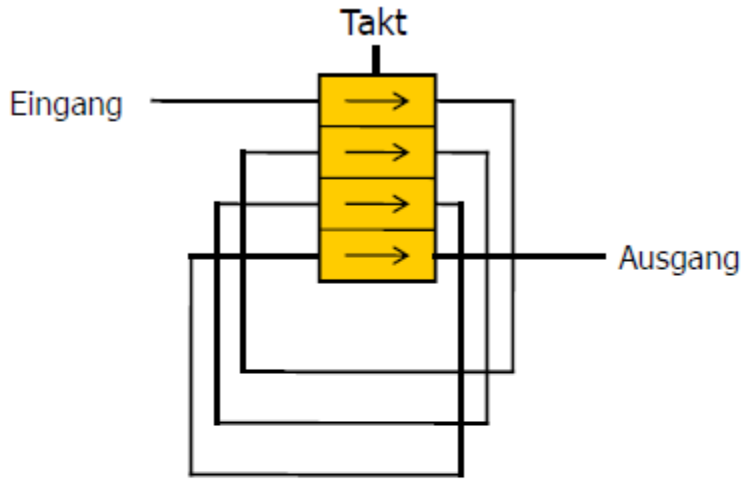
Schieberegister

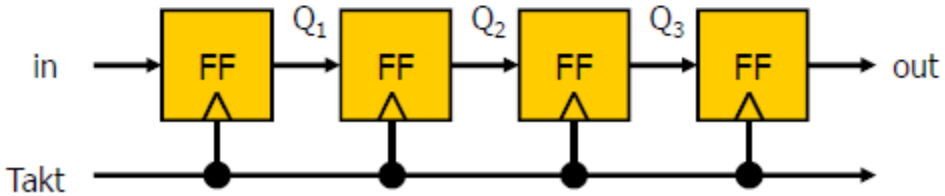
- Schieberegister entstehen durch Hintereinanderschalten von FFs.
- Zwischen den Stufen ist keine (wenig) Logik
- **Vorsicht**: Die Hold-Zeit kann leicht verletzt sein. Daher fügt man manchmal Verzögerungen (Inverterketten) in den Datenpfad ein.
- Anwendungen:
  - Verzögerung von Signalen (z.B. bei Pipelining)
  - Einfache Zustandscodierung
  - Spezielle Zähler (mit Rückkopplung)



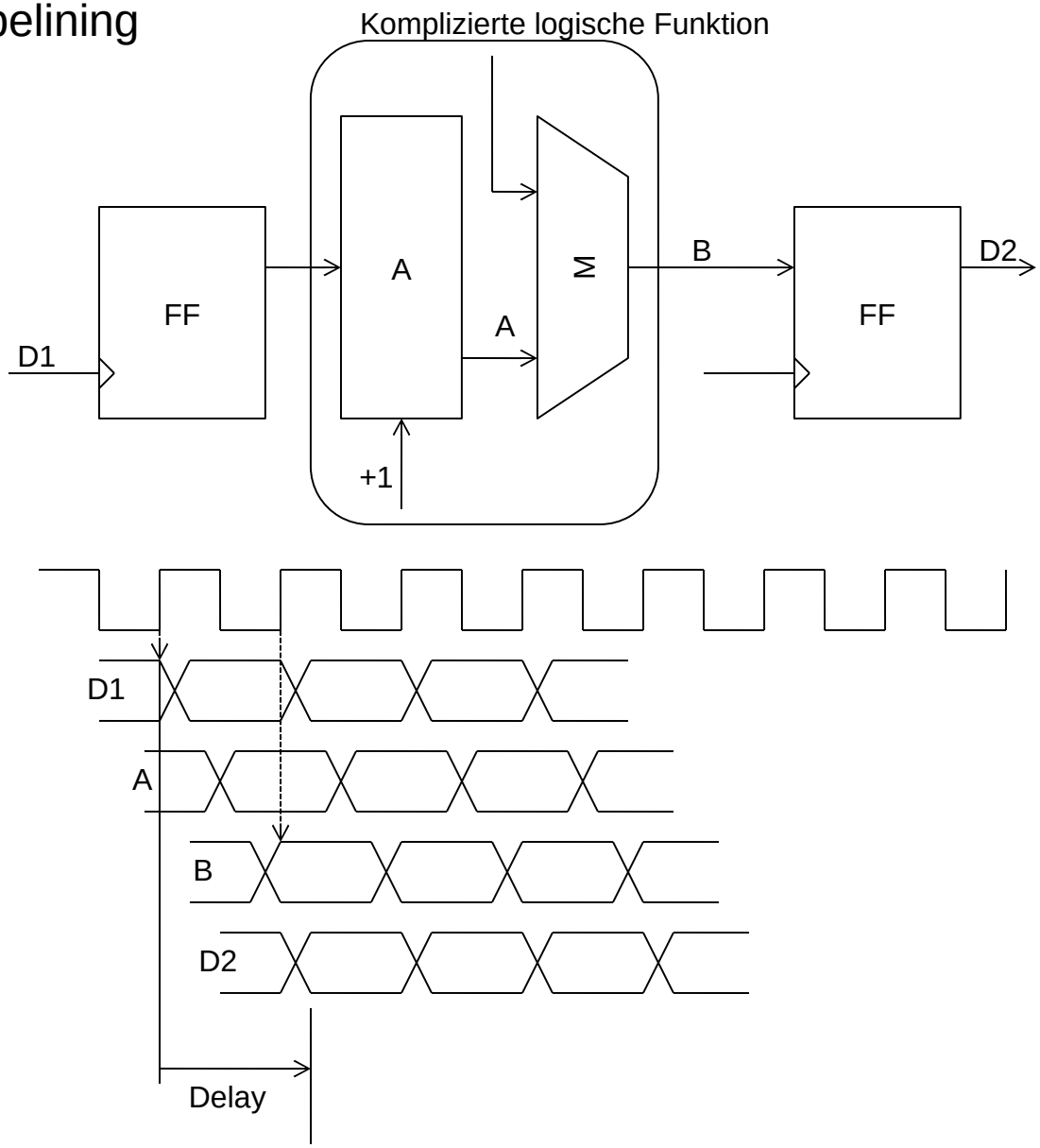
Schieberegister

- Schieberegister: Sehr einfach, keine Logik, ein Eingang, ein Ausgang

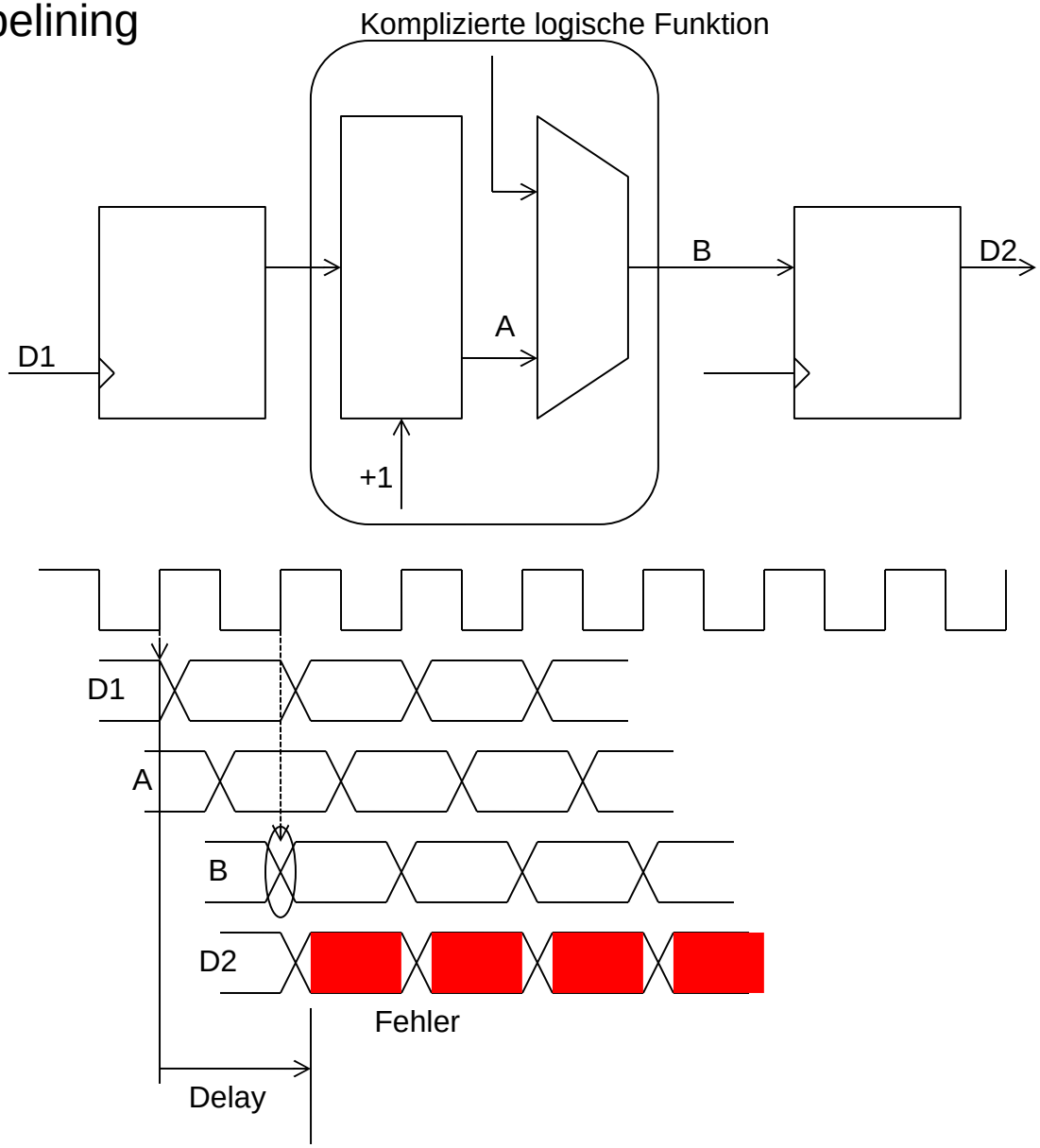




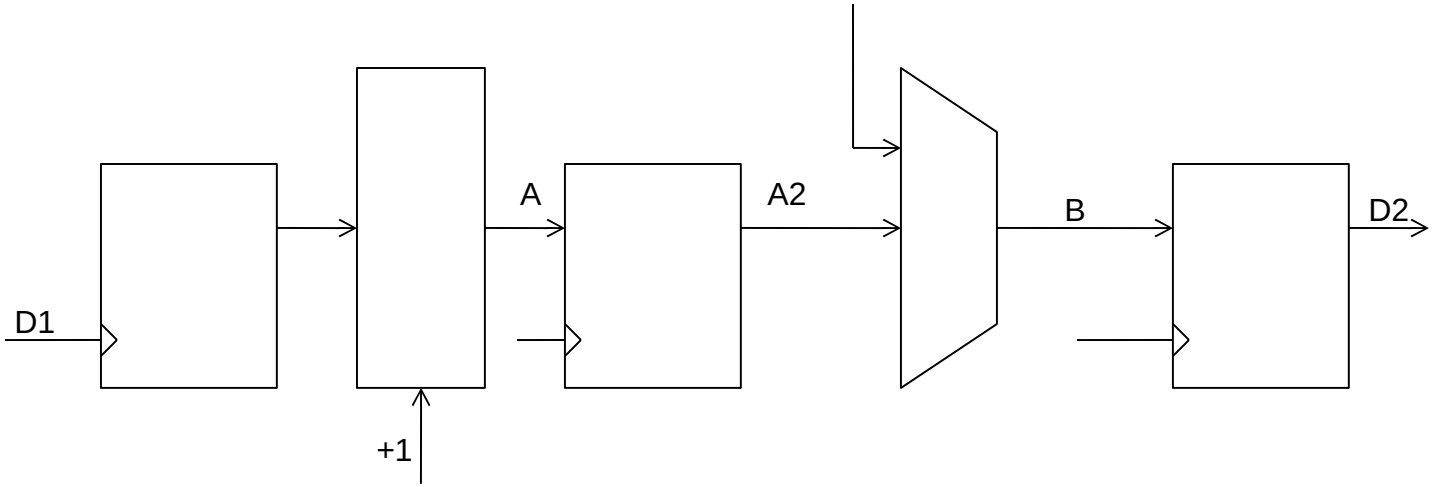
- Pipelining

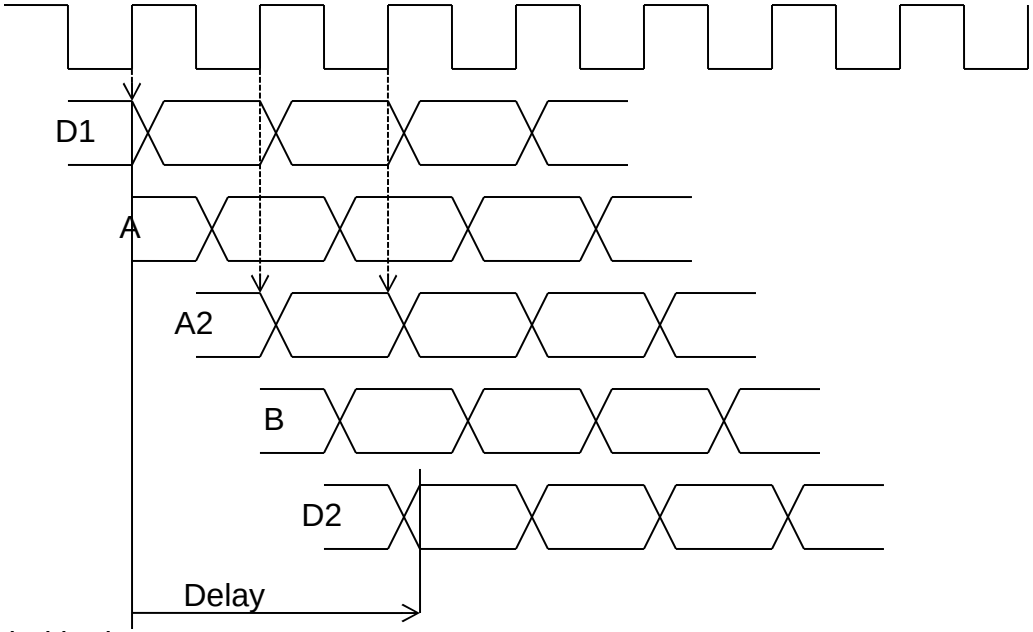

Komplizierte logische Funktion
FF
D1
A
A
M
+1
B
FF
D2
D1
A
B
D2
Delay

- Pipelining


Komplizierte logische Funktion
D1
A
+1
B
D2
D1
A
B
D2
Fehler
Delay

- Pipelining

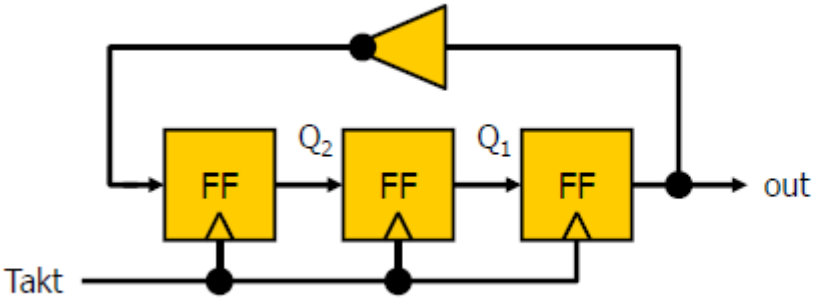


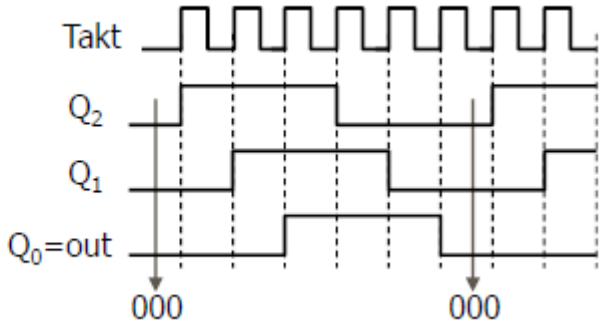


Zähler

Linear Feedback Shift Register (LFSR)

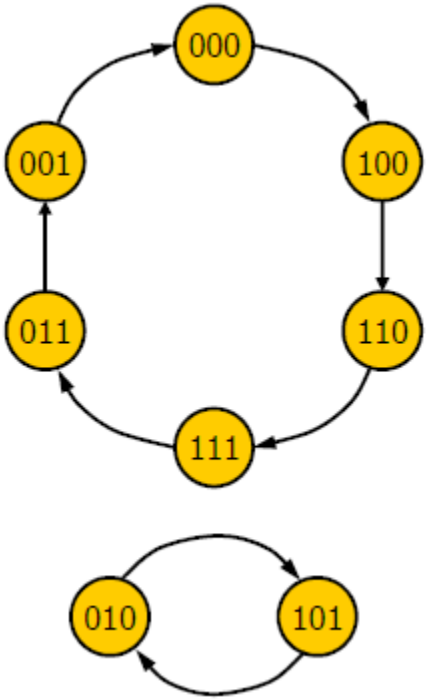
- Sehr **einfach aufgebaute Zähler** werden durch **Linear Feedback Shift Register (LFSR)** erzeugt
- Beim ‚Johnson Zähler' wird der Ausgang über einen Inverter zum Eingang rückgekoppelt
- Der Zähler hat dadurch 2N Zustände…





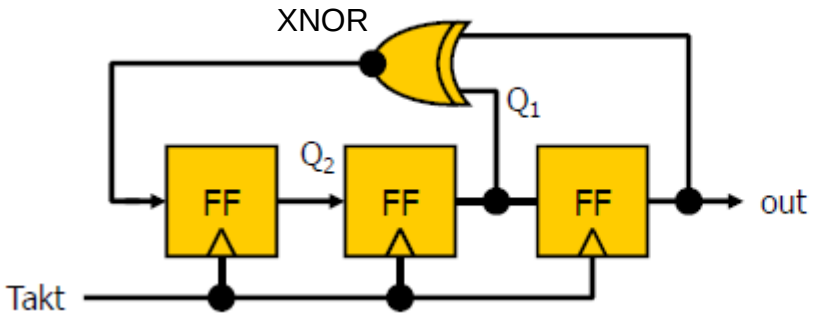
Johnson Zähler: Sprungdiagramm

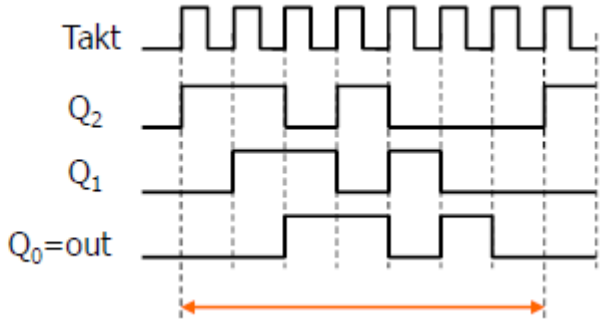
- Bei N=3 gibt es $2^3 = 8$ mögliche Zustände.
- 6 davon werden vom Johnson Zähler durchlaufen:
- Die verbleibenden beiden Zustände bilden einen eigenen Zyklus.
- Man muss mit einem Reset vermeiden hier zu starten!
- Das Zurücksetzen in einen Anfangszustand kann durch sync/async. Reset der FFs erfolgen



Zähler aus Schieberegistern: PRBS

- Durch Rückkopplung des Ausgangs und eines (oder mehrerer) geeigneten Abgriffs (‚tap‘) kann bei N Flipflops eine Bitsequenz mit der Periode 2N-1 entstehen (‚maximum length‘)
- Die Bitsequenz hat keine erkennbare Struktur und wird daher als Pseudo-Random-Bit-Sequence (PRBS) bezeichnet





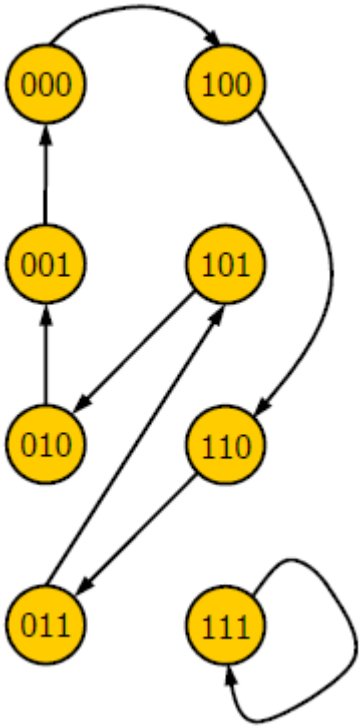
PRBS

- Einige Eigenschaften:
- In der gesamten Sequenz kommt nur genau eine Eins weniger vor als Nullen
- Die Hälfte aller zusammenhängenden Einser-Blöcke ist einen Takt lang, ein Viertel ist zwei Takte lang, etc. (bis auf maximale Sequenzen von Einsen).
- Gleiches gilt für die Nullen.
- 000000111110111100111010110000101110001101101001000100110010101

N	Abgriffe	Länge	Maximal ?
3	Q_1	7	ja
4	Q_1	15	ja
5	Q_2	31	ja
15	Q_1	32767	ja
16	Q_{12}, Q_3, Q_1	65535	ja
16	Q_7		96.8%
39	Q_4	$5x10^{11}$	

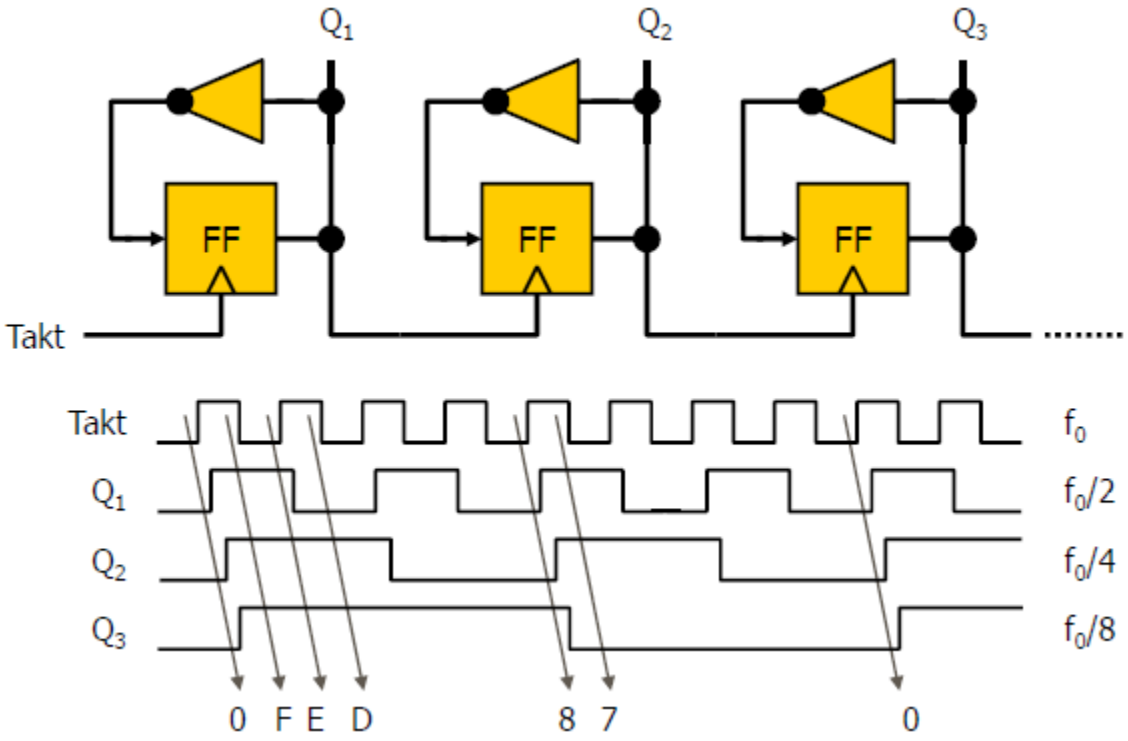
PRBS

- Bei N=3 gibt es 2^3 = 8 mögliche Zustände.
- 7 davon werden durchlaufen
- Zustand 111 ist (bei XNOR feedback) immer stabil



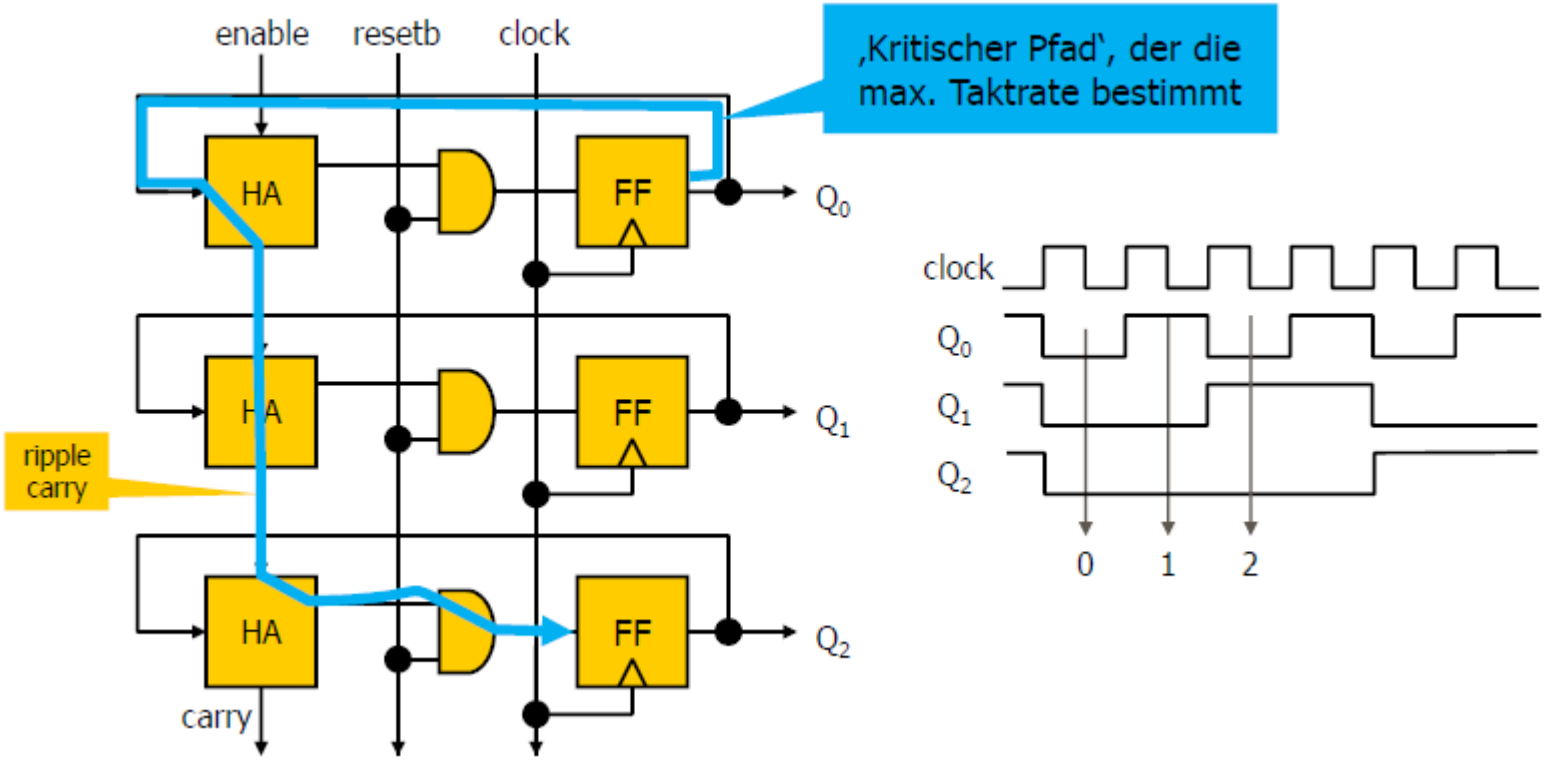
Asynchrone Binärzähler (Ripple Counter)

- Rückkopplung von !Q auf D erzeugt 'Toggle-FFs', die bei jedem Takt den Zustand ändern (0->1->0->...)
- Der Q-Ausgang eines Bits steuert das nächste Bit an (hier Rückwärtszähler):
- Wegen der Verzögerung der einzelnen Stufen sind die Flanken **nicht gleichzeitig** (daher async. Zähler)
- Sollte daher normalerweise vermieden werden. Anwendung: Frequenzteiler



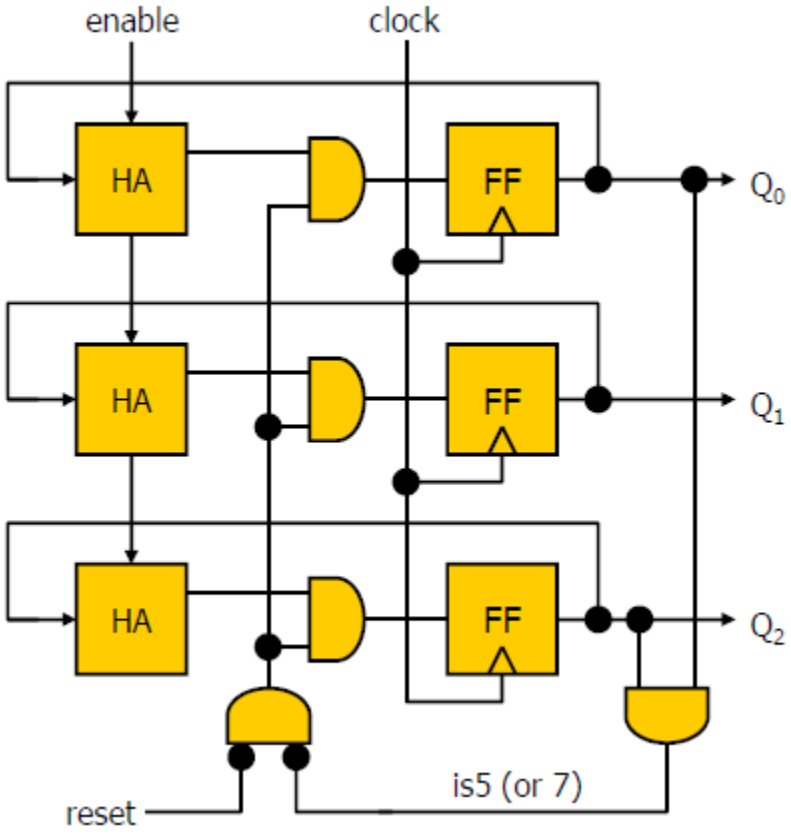
Synchrone Binärzähler

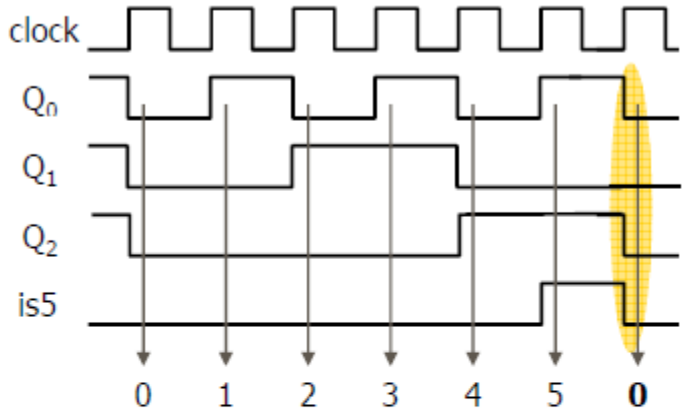
- Alle FFs werden gleichzeitig getaktet
- Die Eingänge werden so beschaltet, daß sich (z.B.) aufsteigend Binärzahlen ergeben
- Implementierung mit Halbaddierern (mit enable und reset)
- Max. Taktfrequenz ist durch die Laufzeit des 'ripple' Carry begrenzt.

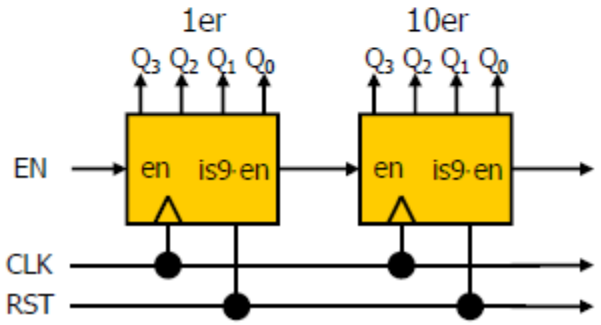


Kürzere synchrone Binärzähler (z.B. BCD Zähler)

- Gibt man das (synchrone) Reset-Signal bei einem bestimmten Zählerstand, so wird die Periode verkürzt.
- Anwendung: BCD Zähler (Periode 10). 'is9 × en' gibt nächste Stufe frei.

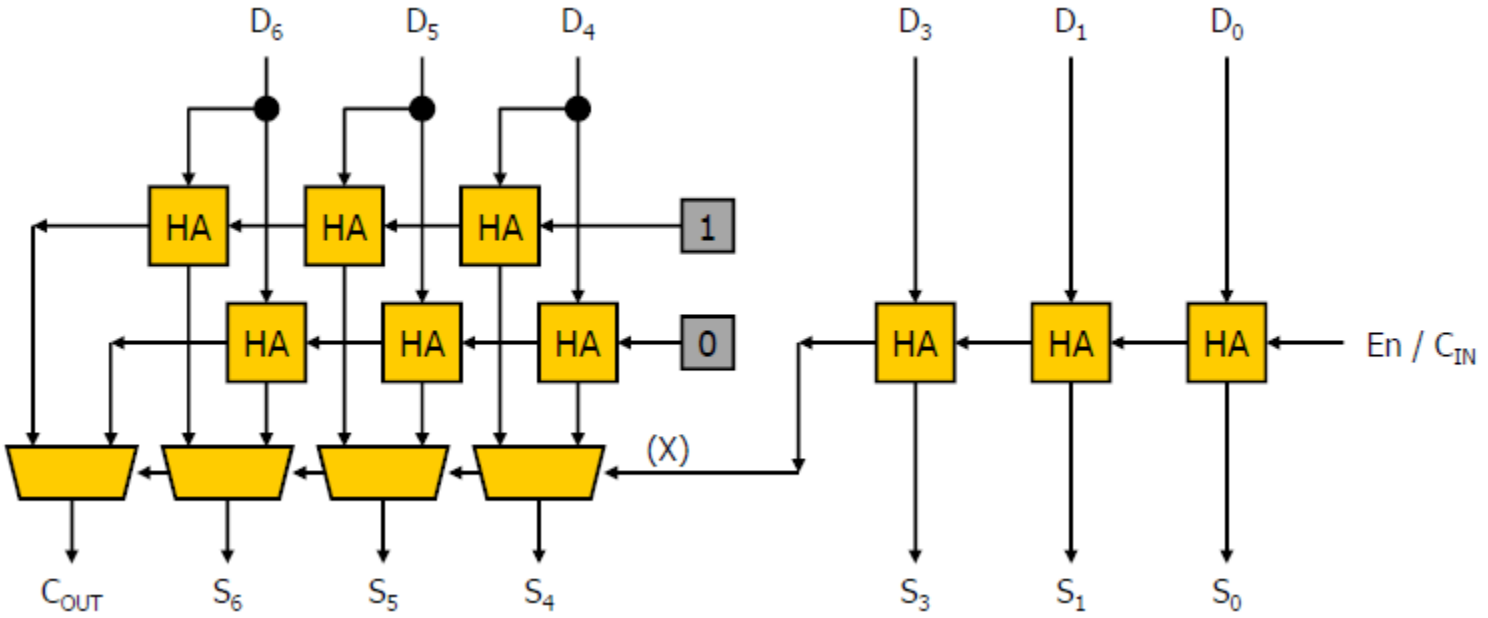






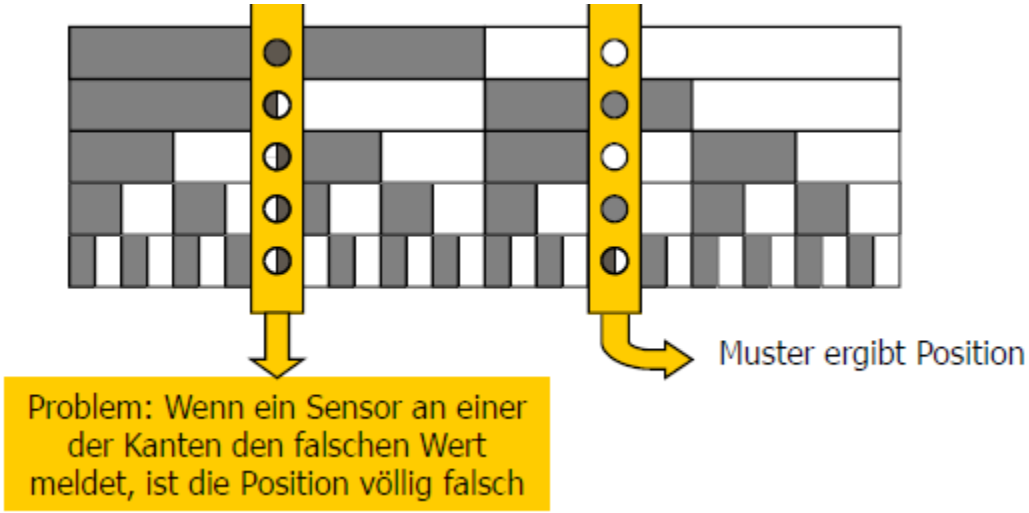
Schnellere Zähler / Addierer

- Bei sehr großen Wortbreiten N muss das Carry-Signal sehr lange durch den Halbaddierer rippeln (N Stufen) und die Schaltung wird langsam.
- Es gibt viele Tricks, um das zu beschleunigen, z.B. den Carry-Select Addierer
  - Berechne für Gruppen von Bits das COUT unter den ZWEI Annahmen CIN = 0 oder CIN =1. Das benötigt ZWEI Addierer.
  - Das COUT (X) der vorangehenden Gruppe wählt dann aus, welches Ergebnis benutzt wird
  - Im Fall von zwei Gruppen a N/2 reduziert sich der Delay auf etwa N/2+1



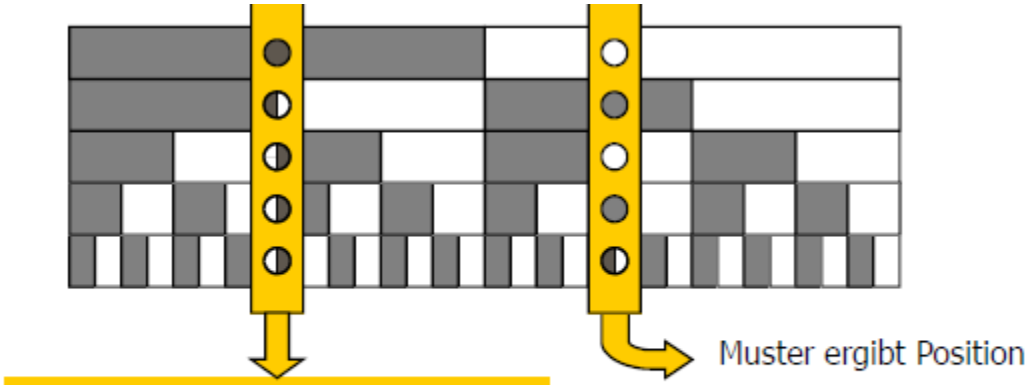
Gray Zähler: Implementierung

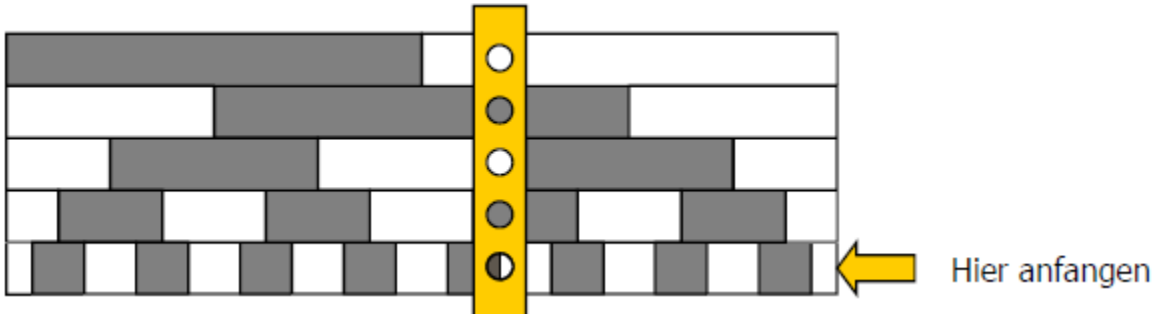
- **Gray Zähler**
- Betrachten wir z.B. einen linearen Maßstab zur Positionsmessung mit binärer Kodierung und Photosensor:



Gray Zähler: Implementierung

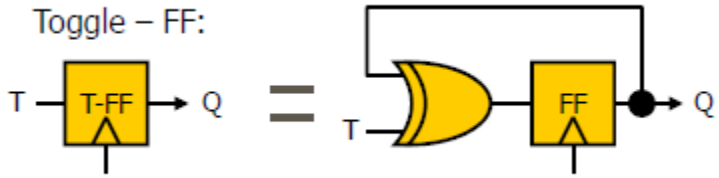
- Lösung: An jeder Kante darf sich nur ein Bit ändern. z.B.: Gray Code: Ändere das niedrigste mögliche Bit

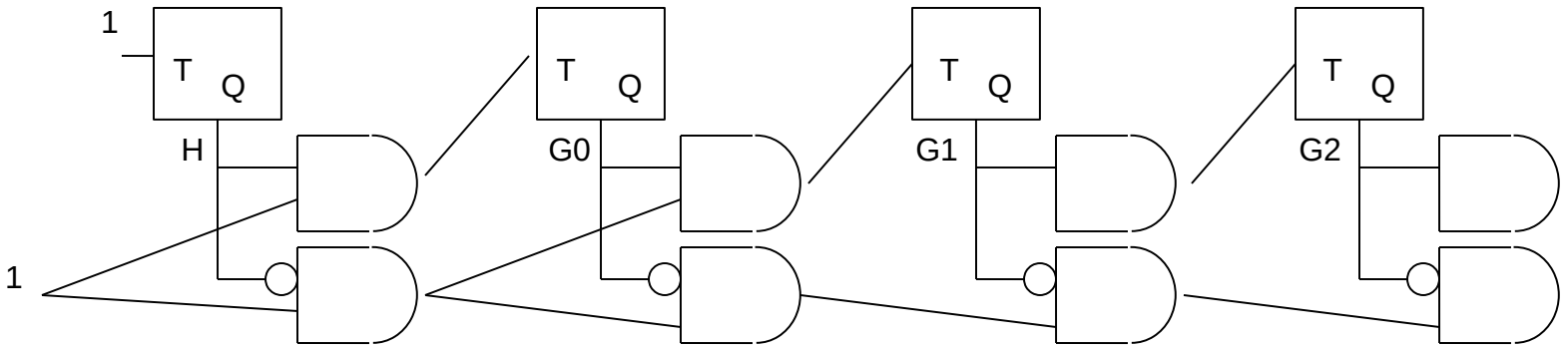




Gray Zähler: Implementierung

- Grey

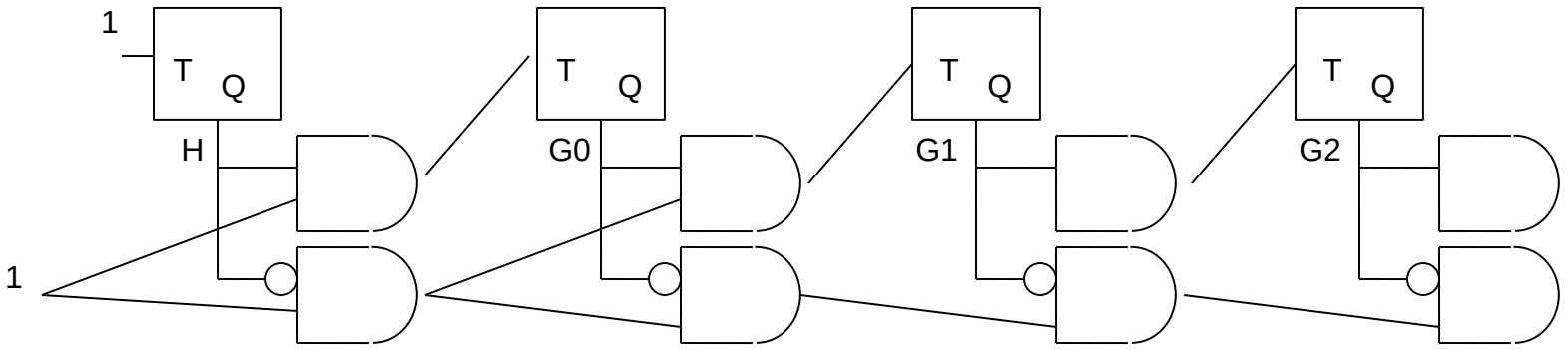




Grey-Zähler

- Grey

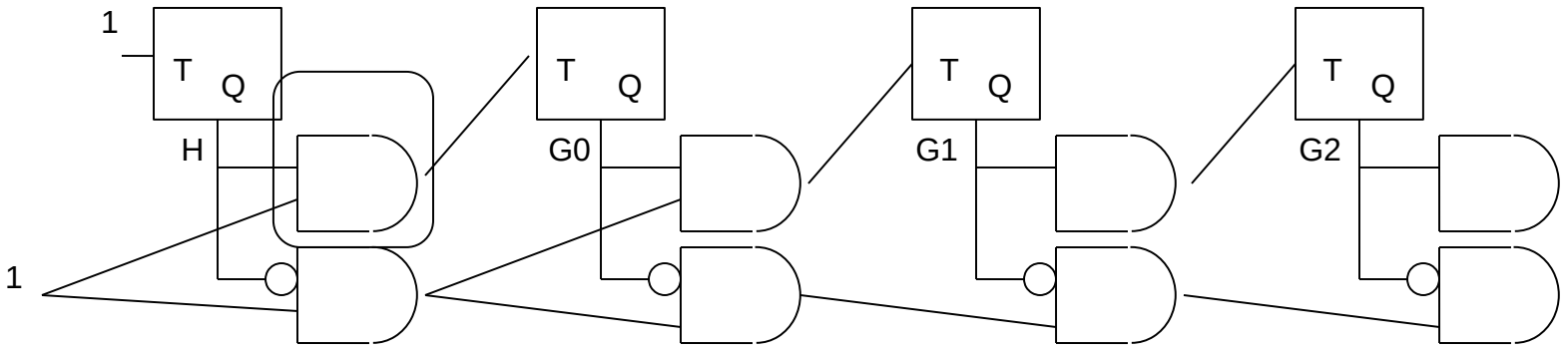
	G2	G1	G0	H
0	0	0	0	1
1	0	0	1	0
2	0	1	1	1
3	0	1	0	0
4	1	1	0	1
5	1	1	1	0
6	1	0	1	1
7	1	0	0	0



Grey-Zähler

- Grey

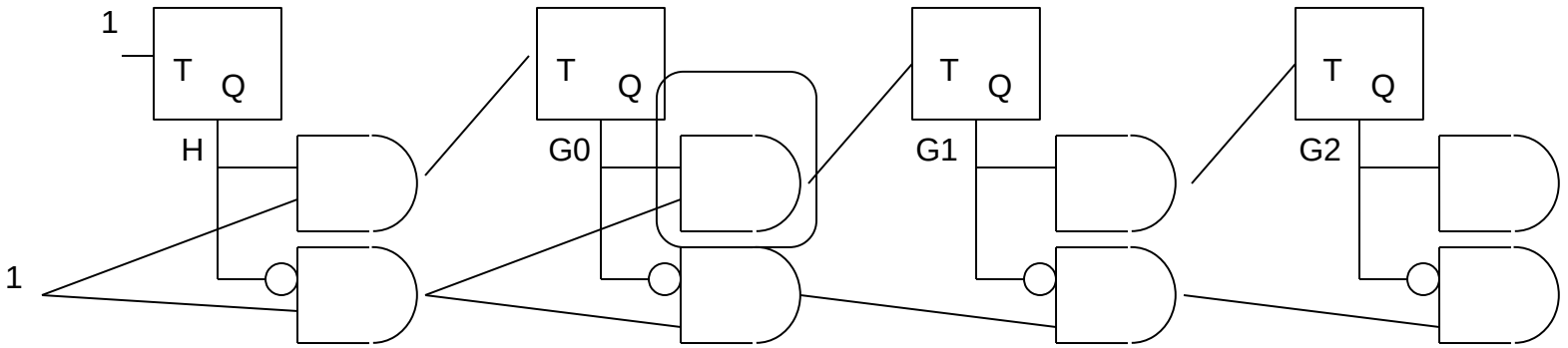
	G2	G1	G0	H
0	0	0	0	1
1	0	0	**1**	0
2	0	1	1	1
3	0	1	**0**	0
4	1	1	0	1
5	1	1	**1**	0
6	1	0	1	1
7	1	0	**0**	0



Grey-Zähler

- Grey

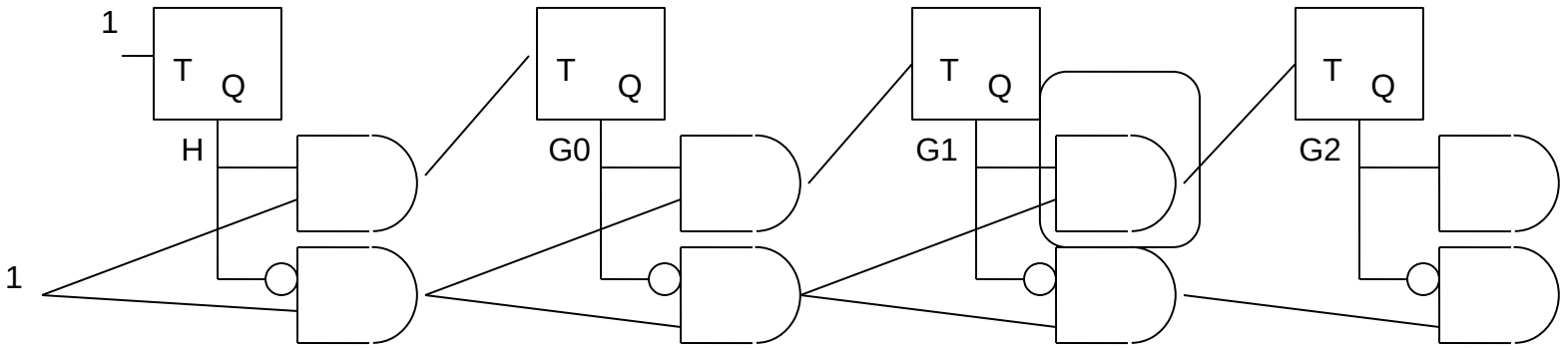
	G2	G1	G0	H
0	0	0	0	1
1	0	0	1	0
2	0	**1**	1	1
3	0	1	0	0
4	1	1	0	1
5	1	1	1	0
6	1	**0**	1	1
7	1	0	0	0



Grey-Zähler

- Grey

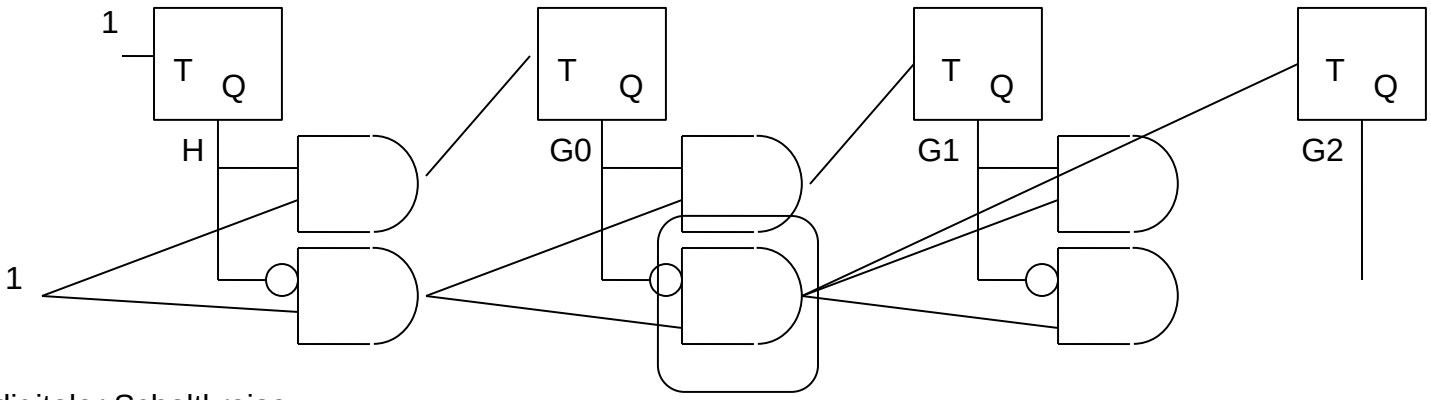
	G2	G1	G0	H
0	0	0	0	1
1	0	0	1	0
2	0	**1**	1	1
3	0	1	0	0
4	**1**	1	0	1
5	1	1	1	0
6	1	**0**	1	1
7	1	0	0	0



Grey-Zähler

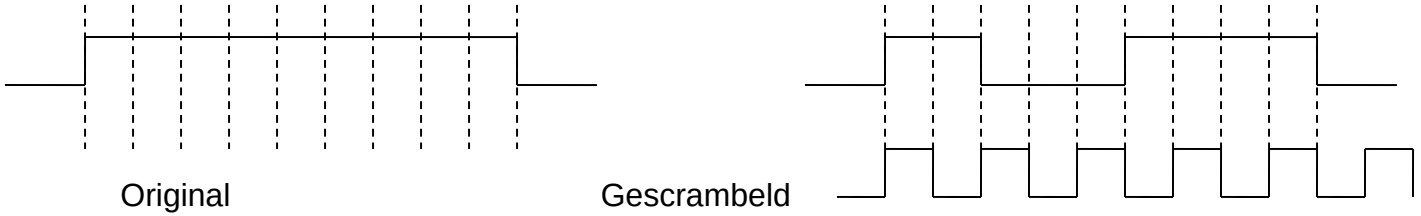
- Grey

	G2	G1	G0	H
0	0	0	0	1
1	0	0	1	0
2	0	**1**	1	1
3	0	1	0	0
4	**1**	1	0	1
5	1	1	1	0
6	1	**0**	1	1
7	1	0	0	0



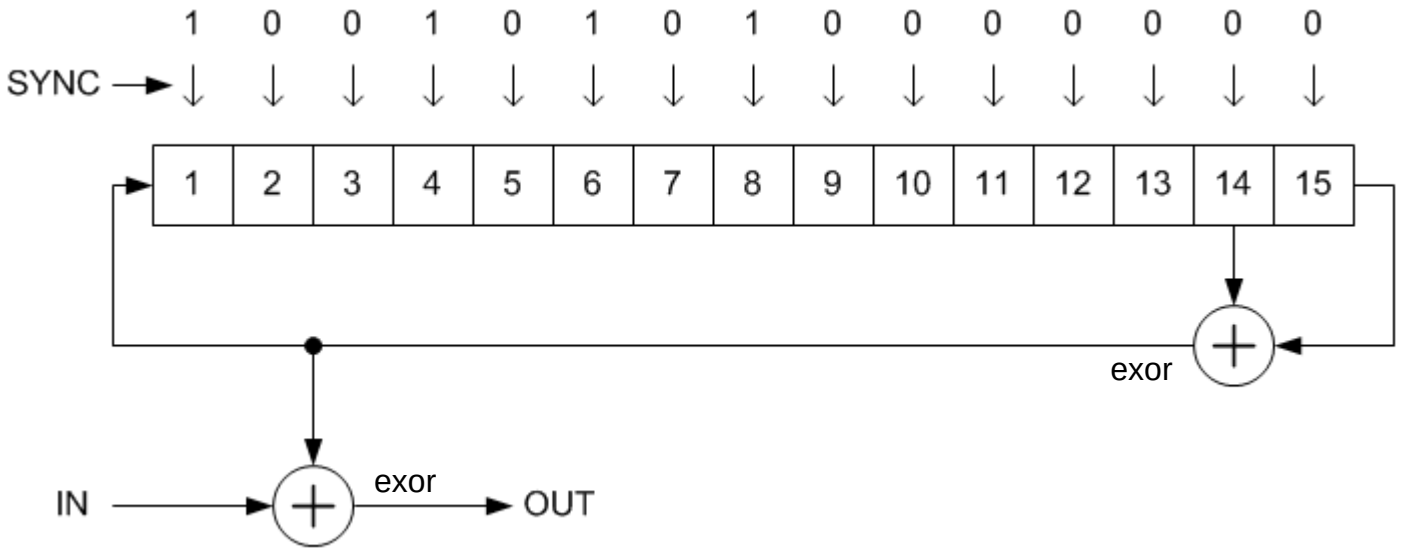
Scrambler

- Ein **Scrambler** (deutsch *Verwürfler*) verwendet linear rückgekoppelte Schieberegister (LFSR), um ein Digitalsignal umkehrbar umzustellen
- Ein Scrambler basierend auf LFSR stellt wegen der einfachen und bekannten Verfahren keine brauchbare Verschlüsselung von Daten dar.
- Ein Scrambler wird durch linear rückgekoppelte Schieberegister (LFSR) realisiert. Dabei wird meistens die pro Schieberegisterlänge maximal mögliche Codelänge verwendet



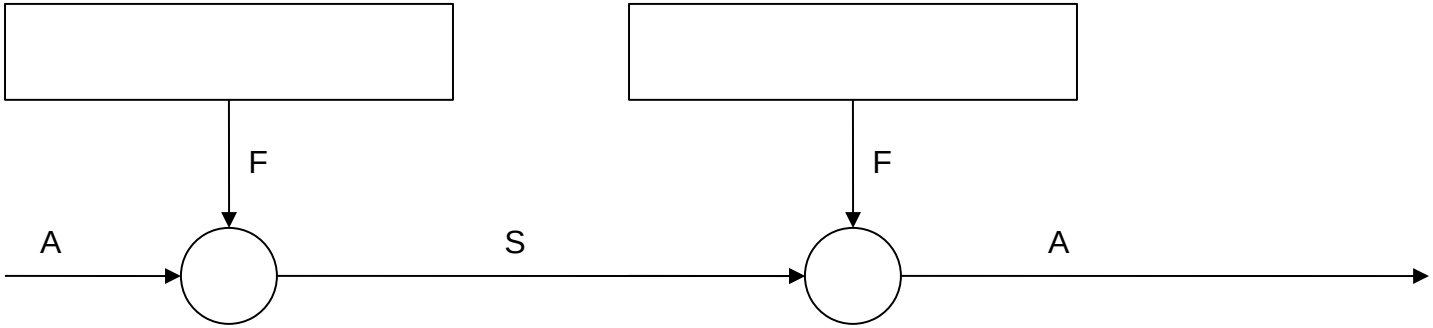
Synchrone (additive) Scrambler

- Synchrone oder auch additive Scrambler benötigen einen definierten Startwert ungleich 0 im LFS-Register, und der Empfänger muss durch geeignete Maßnahmen, wie beispielsweise einem speziellen Sync-Wort, die genaue Codephasenlage des Senders mitgeteilt bekommen.
- Ist dem Empfänger die korrekte Codephasenlage nicht bekannt, kann er das gescrambelte Datensignal nicht richtig dekodieren.
- Vorteil: Fehler werden nicht multipliziert
- Nachteil: Synchronisierung nötig



Synchrone (additive) Scrambler

- Descrambling basiert auf der Gleichheit A = A exor F exor F
- Wenn man A zweimal mit gleicher Zahl F „ex-odert“ bekommt man A

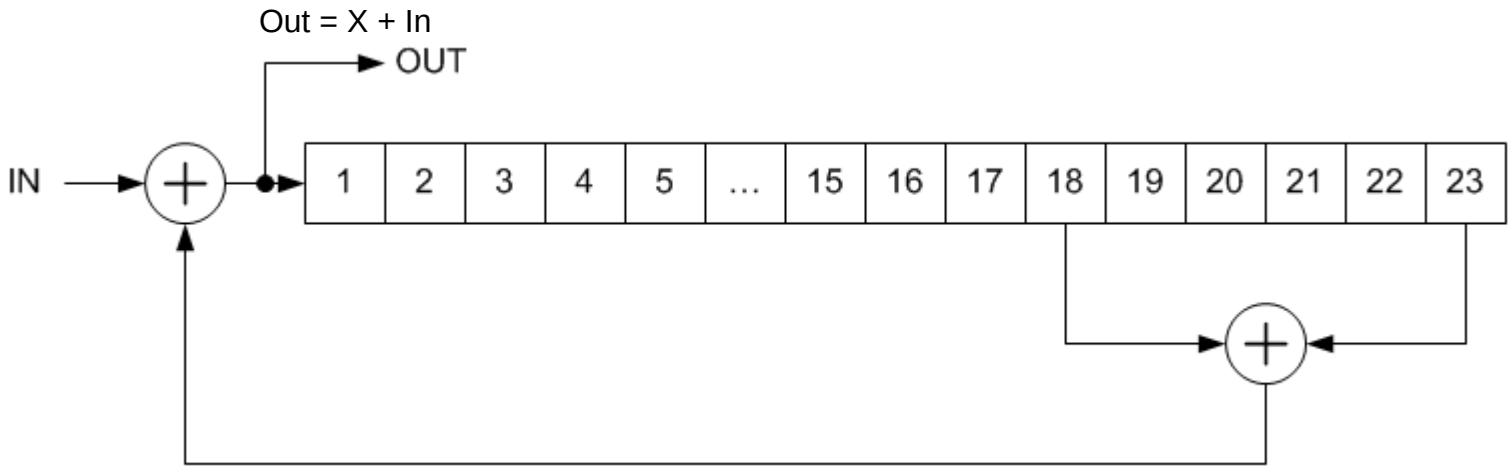


Selbstsynchronisierende (multiplikative) Scrambler

- Selbstsynchronisierende oder auch multiplikative Scrambler benötigen keinen definierten Startwert und auch kein Sync-Wort, um die Codephase des Empfängers mit der Codephase des Senders abzugleichen. Auch kann der Startwert des LFSR beliebig sein.
- Erreicht wird die Funktion der Selbstsynchronität dadurch, dass die Nutzdatenfolge direkt auf den Inhalt des LFSR einwirkt
- Nachteilig ist die Abhängigkeit des Scramblers von der Nutzdatenfolge. So können bestimmte Nutzdatenfolgen den Scrambler vollständig "auslöschen"
- Darüber hinaus pflanzen sich Übertragungsfehler bei selbstsynchronisierenden Scramblern fort

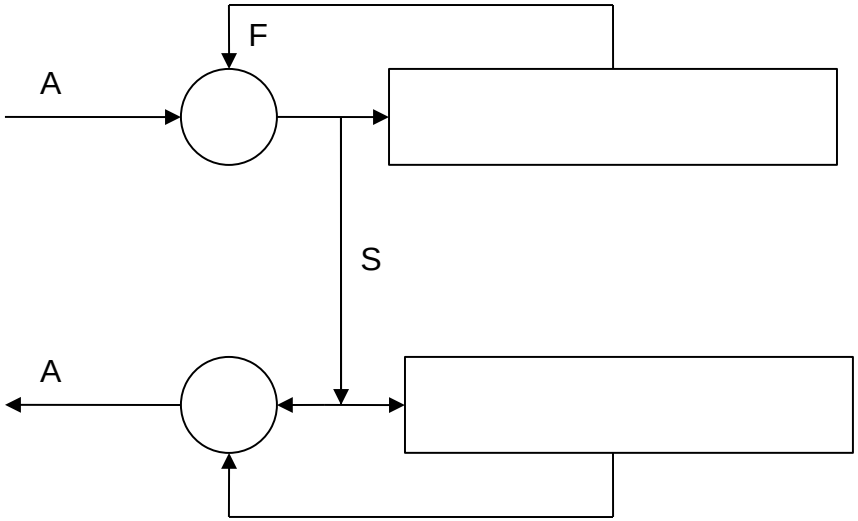
Selbstsynchronisierende (multiplikative) Scrambler

- …



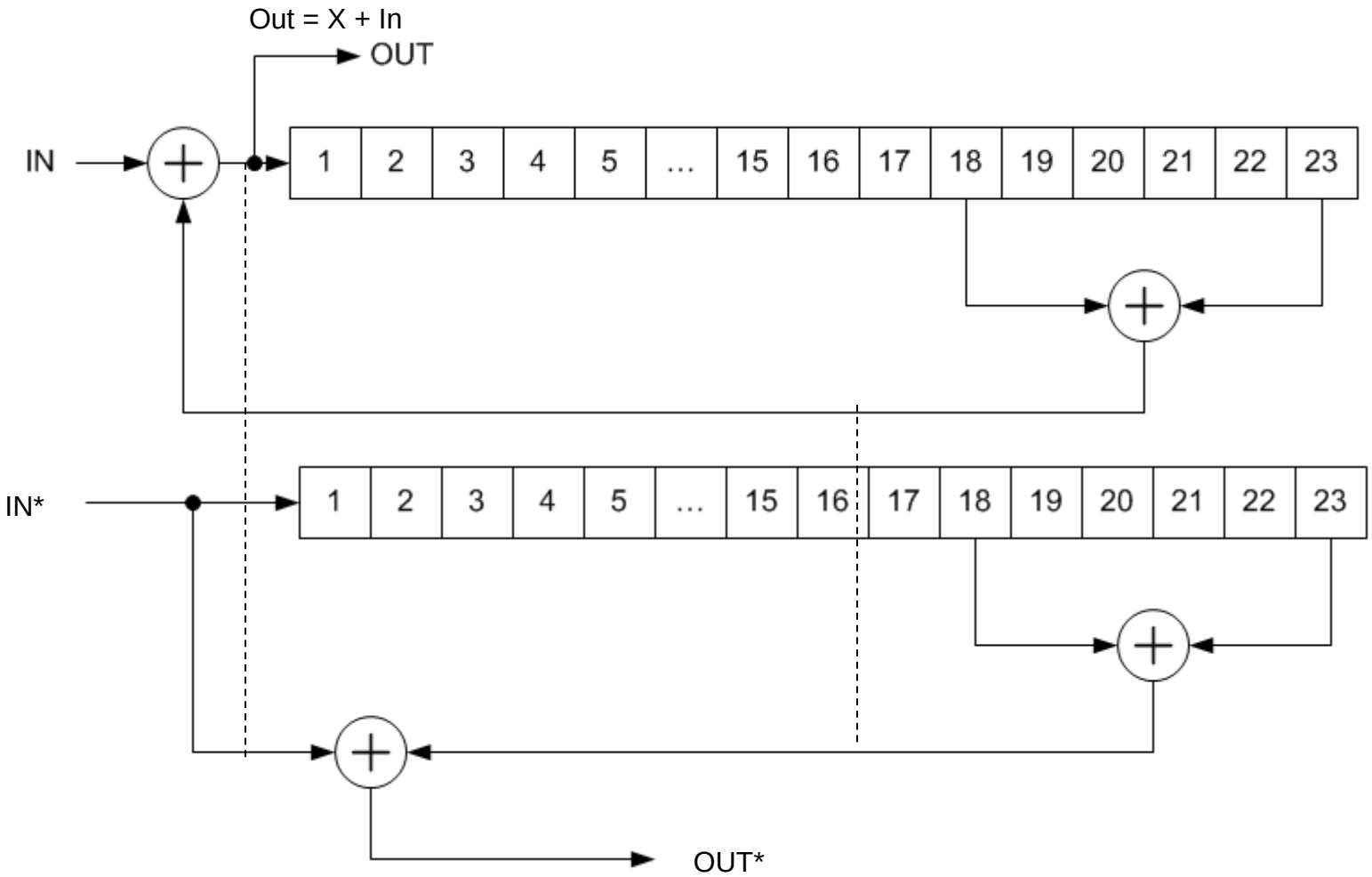
Selbstsynchronisierende (multiplikative) Scrambler

- Descrambling basiert auf der Gleichheit A = A exor F exor F
- Wenn man A zweimal mit gleicher Zahl F „ex-odert“ bekommt man A



Selbstsynchronisierende (multiplikative) Scrambler

- …



Out* = X + Out = X + X + In = In