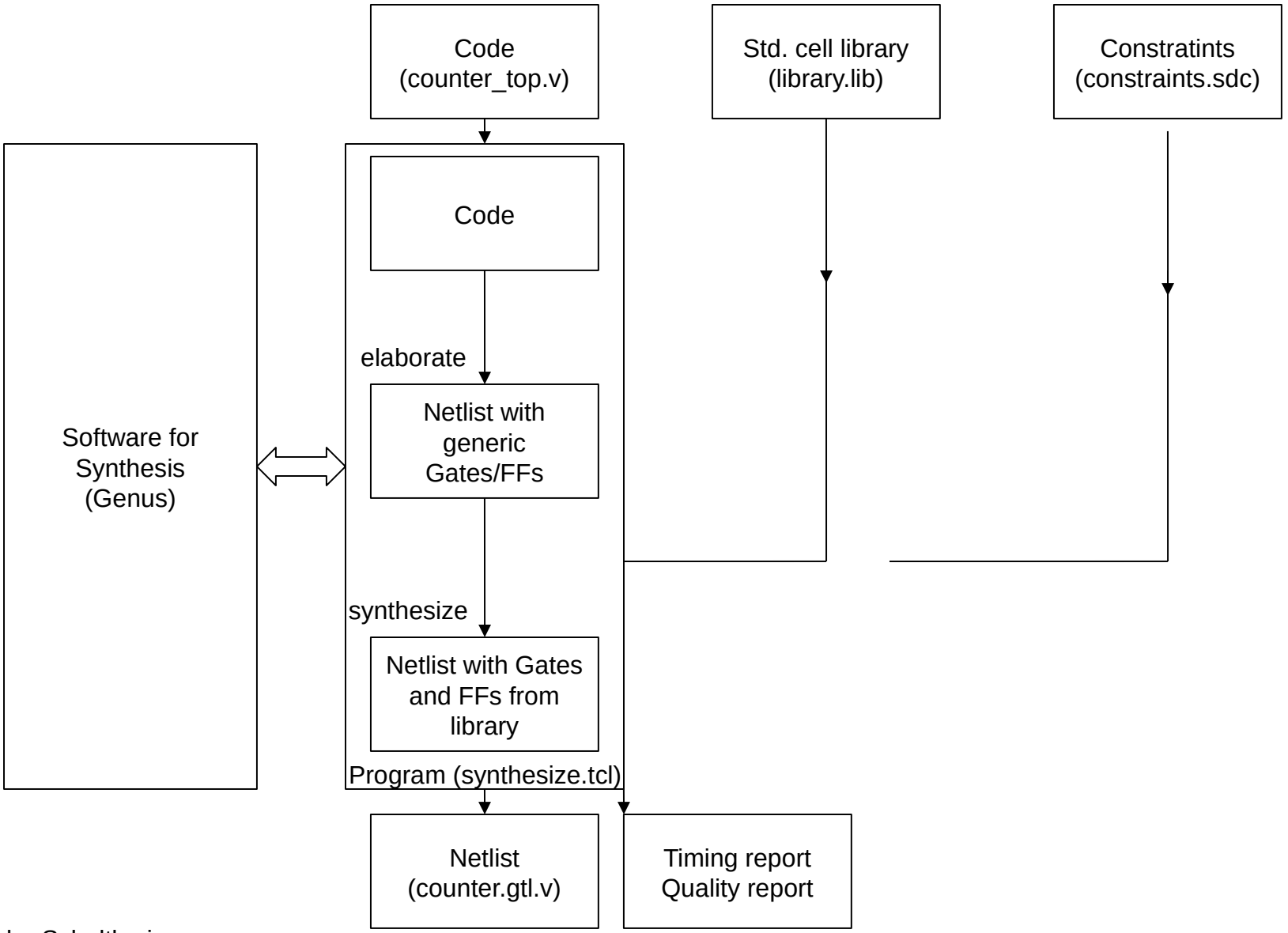
Design digitaler Schaltkreise

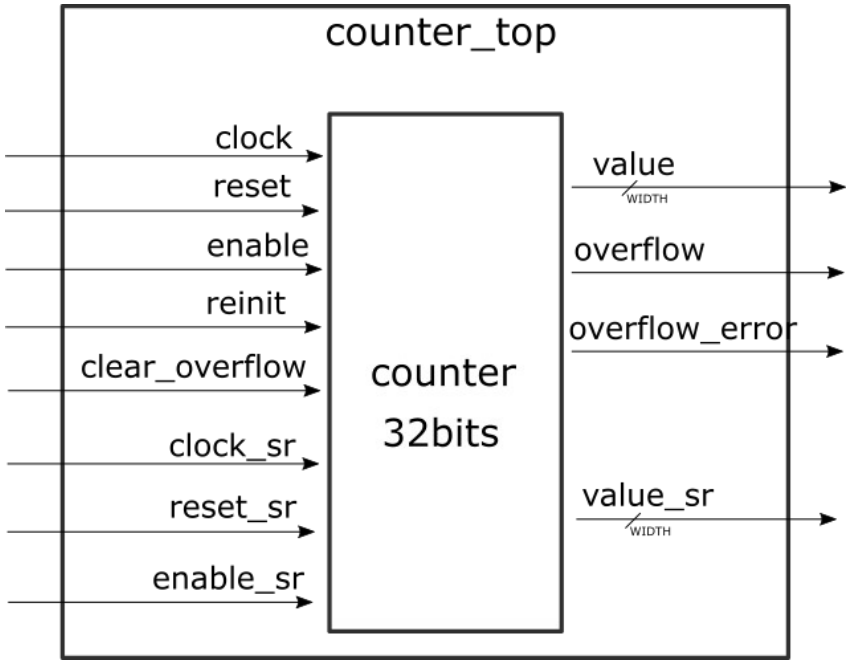
Übung 2
Synthese

Synthesis

- …


Code
(counter_top.v)
Std. cell library
(library.lib)
Constratints
(constraints.sdc)
Code
elaborate
Netlist with
generic
Gates/FFs
Software for
Synthesis
(Genus)
synthesize
Netlist with Gates
and FFs from
library
Program (synthesize.tcl)
Netlist
(counter.gtl.v)
Timing report
Quality report

- Counter
- counter_top contains an instance of the counter, and feed through the counter's input and outputs



- The Cadence tools are usually very complex
- We will present here only the minimal set of commands required to produce a result
- Commands callable from a TCL (tool command language) script
- Create a file called synthesize.tcl

Loading library

- To start with synthesis, we have to select a set of logic functions (gates and flipflops) to be used.
- The technology kit provides a set of functions, with various timing characterization option.
- TCL command: *set_db library /path/to/library.lib*

```
/*****************************************************************************************
 Module           : AND2X1_HV
 Cell Description : Combinational cell (AND2X1_HV) with drive strength X1
*****************************************************************************************/

cell (AND2X1_HV) {

  drive_strength  : 1;

  area                : 14.112000;
  pg_pin(gnd!) {
      voltage_name : gnd!;
      pg_type      : primary_ground;
  }

  pg_pin(vdd!) {
      voltage_name : vdd!;
      pg_type      : primary_power;
  }


  cell_leakage_power  : 1650.967087;

  leakage_power () {
      when           : "!A & !B";
      value          : 1110.011040;
  }
  leakage_power () {
      when           : "!A & B";
      value          : 1572.759967;
  }
```

Loading design and elaboration

- Loading design
- Command: *read_hdl -v2001 {counter_top.v ../assignment1/counter.v}*
- Elaboration
- Command: *elaborate*
- Elaboration is the process of implementing HDL description as a netlist with generic gates and flipflops

```
not g4 (n_176, value_sr[31]);
not g10 (n_319, reinit);
and g15 (n_318, n_240, enable);
and g16 (n_320, n_318, n_319);
or g17 (n_322, n_320, reinit);
and g21 (n_327, n_9, overflow);
and g22 (n_328, n_327, n_319);
or g23 (n_329, n_328, reinit);
and g24 (n_330, enable, n_319);
or g25 (n_331, n_330, reinit);
not g1 (n_240, overflow_error);
and g27 (n_9, n_315, enable);
CDN_flop \value_reg[0] (.clk (clock), .d (n_273), .sena (n_322),
    .aclr (1'b0), .apre (1'b0), .srl (reset), .srd (1'b0), .q
    (value[0]));
CDN_flop \value_reg[1] (.clk (clock), .d (n_274), .sena (n_322),
    .aclr (1'b0), .apre (1'b0), .srl (reset), .srd (1'b0), .q
    (value[1]));
CDN_flop \value_reg[2] (.clk (clock), .d (n_275), .sena (n_322),
    .aclr (1'b0), .apre (1'b0), .srl (reset), .srd (1'b0), .q
    (value[2]));
CDN_flop \value_reg[3] (.clk (clock), .d (n_276), .sena (n_322),
    .aclr (1'b0), .apre (1'b0), .srl (reset), .srd (1'b0), .q
    (value[3]));
CDN_flop \value_reg[4] (.clk (clock), .d (n_277), .sena (n_322),
    .aclr (1'b0), .apre (1'b0), .srl (reset), .srd (1'b0), .q
    (value[4]));
CDN_flop \value_reg[5] (.clk (clock), .d (n_278), .sena (n_322),
    .aclr (1'b0), .apre (1'b0), .srl (reset), .srd (1'b0), .q
    (value[5]));
CDN_flop \value_reg[6] (.clk (clock), .d (n_279), .sena (n_322),
    .aclr (1'b0), .apre (1'b0), .srl (reset), .srd (1'b0), .q
    (value[6]));
```

Constraints

- Defining the constraints is done by calling a set of commands, using a standard called SDC (Synopsis Design Constraints).
- SDC commands are supported by tools from Cadence, Synopsis and the newest Xilinx Vivado Suite, which makes design constraining easy to learn and reuse among various technologies.
- constraints.sdc

Constraints

- Defining the clocks is the first step when writing a constraints file. Mostly three parameters are required:
- A clock frequency or waveform (if the clock is not symetrical)
- A target wire to apply it to
- A name to identify the clock. It is usually set to the name of the wire, for clarity
- Commands in constraints.sdc
- *create_clock -name clock -period 1 [get_port clock]*
- Further commands:
- *set_clock_uncertainty 0.1 -setup clock*
- (reduces setup slack by 0.1 ns)
- *set_clock_uncertainty 0.1 -hold clock*
- (reduces setup slack by 0.1 ns)
- *set_output_delay -clock clock 0.5 [get_ports {value*}]*
- Load command in synthesize.tcl: *read_sdc constraints.sdc*

Configuring timing output

- Timing groups are a feature of the tool to group the logic paths depending on their types and help make timing analysis clearer
- Input to register (I2C)
- Register to output (C2O)
- Register to Register (C2C)
- Example:
- *set all_regs [all des seqs -clock clock]*
- *define_cost_group -name C2C*
- *path_group -from $all_regs -to $all_regs -group C2C -name C2C*

Synthesis

- Commands for synthesis:
- *synthesize -to_mapped -effort medium*
- Incremental optimization:
- *synthesize -to_mapped -incr -effort medium*
- Command for writing the output netlist: *write_hdl*
- Command for writing of timing output: *report timing*

Netlist

- …

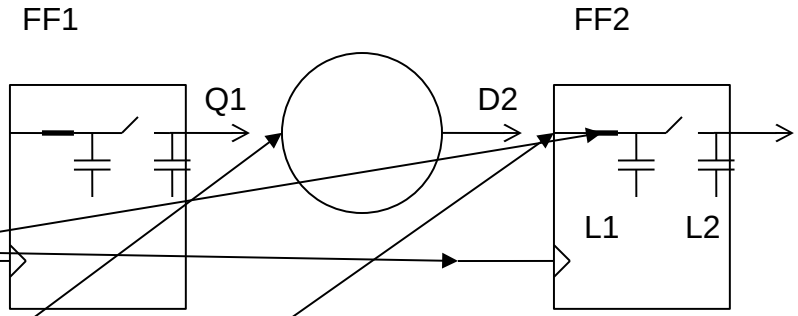
```
BUFX24_HV g1416(.A (\value[7]_570 ), .Q (value[7]));
BUFX24_HV g1409(.A (\value[6]_569 ), .Q (value[6]));
BUFX32_HV g1408(.A (\value_sr[26]_623 ), .Q (value_sr[26]));
BUFX24_HV g1415(.A (\value[5]_568 ), .Q (value[5]));
BUFX24_HV g1425(.A (\value[4]_567 ), .Q (value[4]));
DFX4_HV overflow_error_reg(.CP (clock), .D (n_277), .Q
    (overflow_error_596), .QN (UNCONNECTED0));
DFX4_HV overflow_reg(.CP (clock), .D (n_276), .Q (overflow_595), .QN
    (n_58));
DFX6_HV \value_reg[0] (.CP (clock), .D (n_608), .Q (\value[0]_563 ),
    .QN (UNCONNECTED1));
DFX6_HV \value_reg[1] (.CP (clock), .D (n_246), .Q (\value[1]_564 ),
    .QN (UNCONNECTED2));
DFX6_HV \value_reg[2] (.CP (clock), .D (n_493), .Q (\value[2]_565 ),
    .QN (UNCONNECTED3));
DFX6_HV \value_reg[3] (.CP (clock), .D (n_243), .Q (\value[3]_566 ),
    .QN (UNCONNECTED4));
DFX6_HV \value_reg[4] (.CP (clock), .D (n_604), .Q (\value[4]_567 ),
    .QN (UNCONNECTED5));
DFX6_HV \value_reg[5] (.CP (clock), .D (n_241), .Q (\value[5]_568 ),
    .QN (UNCONNECTED6));
```

Timing report

```
Path 1: VIOLATED (-588 ps) Setup Check with Pin dut/value_reg[14]/CP->D
          Group: C2C
     Startpoint: (R) dut/value_reg[11]/CP
          Clock: (R) clock
       Endpoint: (R) dut/value_reg[14]/D
          Clock: (R) clock

                    Capture       Launch
       Clock Edge:+    1000            0
       Src Latency:+      0            0
       Net Latency:+      0 (I)        0 (I)
           Arrival:=   1000            0

             Setup:-    174
       Uncertainty:-    100
     Required Time:=    726
      Launch Clock:-      0
         Data Path:-   1314
             Slack:=   -588
```

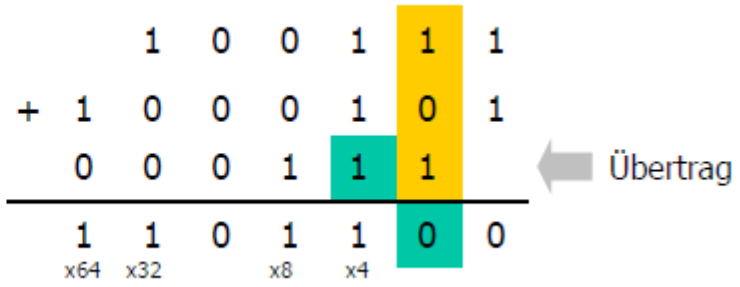


Timing Point	Flags	Arc	Edge	Cell	Fanout	Load (fF)	Trans (ps)	Delay (ps)	Arrival (ps)
dut/value_reg[11]/CP	-	-	R	(arrival)	66	-	0	-	0
dut/value_reg[11]/Q	-	CP->Q	F	DFX6_HV	5	43.0	111	459	459
dut/inc_add_34_54/g6008/Q	-	A->Q	F	BUFX24_HV	3	23.2	45	156	615
dut/inc_add_34_54/g5977/Q	-	B->Q	F	AND2X8_HV	1	16.3	47	141	756
dut/inc_add_34_54/g5963/Q	-	A->Q	R	NAND2X12_HV	4	47.3	126	100	856
dut/inc_add_34_54/g5933/Q	-	A->Q	F	NOR2X8_HV	1	11.0	56	47	903
dut/inc_add_34_54/g5886/Q	-	A->Q	R	NAND2X6_HV	1	10.9	82	72	975
dut/inc_add_34_54/g5830/Q	-	A->Q	R	XNOR2X2_HV	1	8.8	207	176	1150
dut/g6663/Q	-	A1->Q	R	AO22X6_HV	1	7.1	63	164	1314
dut/value_reg[14]/D	<<<	-	R	DFX6_HV	1	-	-	0	1314

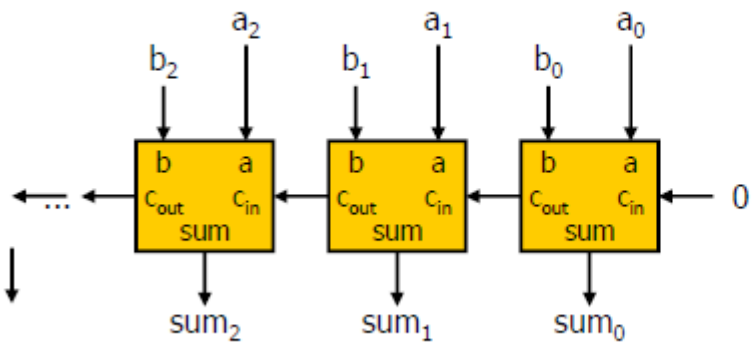
Vorlesung 6
Addition von Binärzahlen

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):



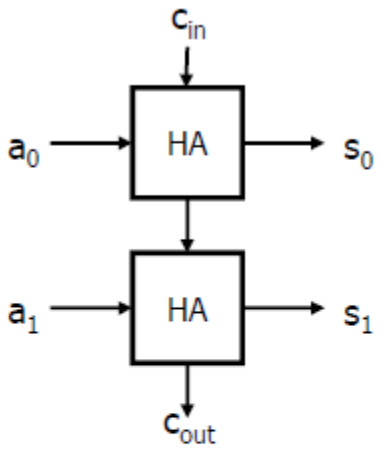
- Die Addition erfolgt stellenweise wie bei Dezimalzahlen mit einem **Übertrag (carry)**:
- In jeder Stufe werden also aus den **3 Eingängen** a,b,cin die **Ausgänge** sum und cout erzeugt.
- Man nennt diesen wichtigen Schaltungsblock den **Volladdierer** (full adder, FA):



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

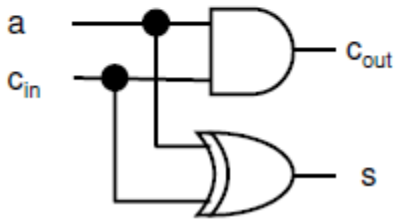
Halbaddierer

- 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)



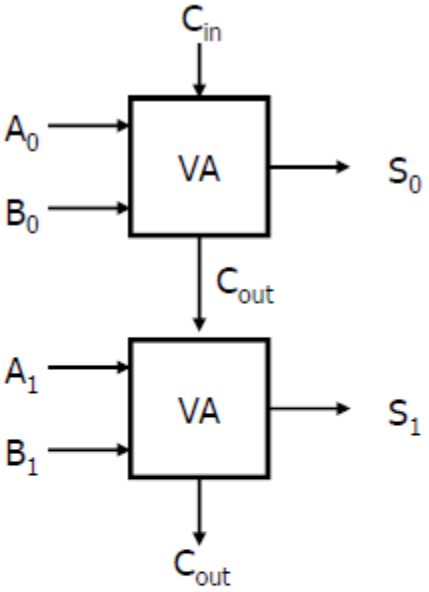
c_{in}	a	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

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



- …

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}:

C_{in} \ A,B	00	01	11	10
0	0	0	**1**	0
1	0	**1**	**1**	**1**

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

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

Sum:

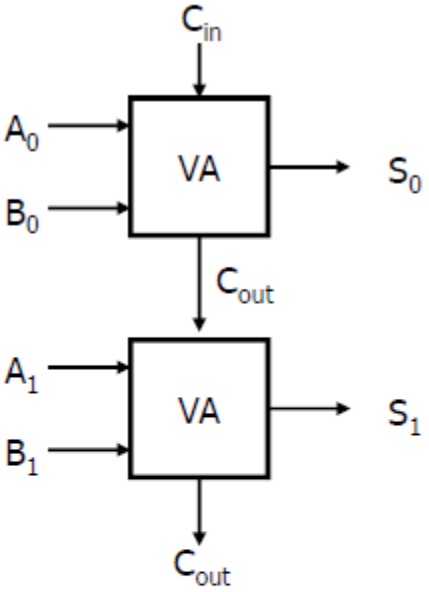
C_{in} \ A,B	00	01	11	10
0	0	**1**	0	**1**
1	**1**	0	**1**	0

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

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

- …

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}:

C_{in} \ A,B	00	01	11	10
0	0	0	**1**	0
1	0	**1**	**1**	**1**

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

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

Sum:

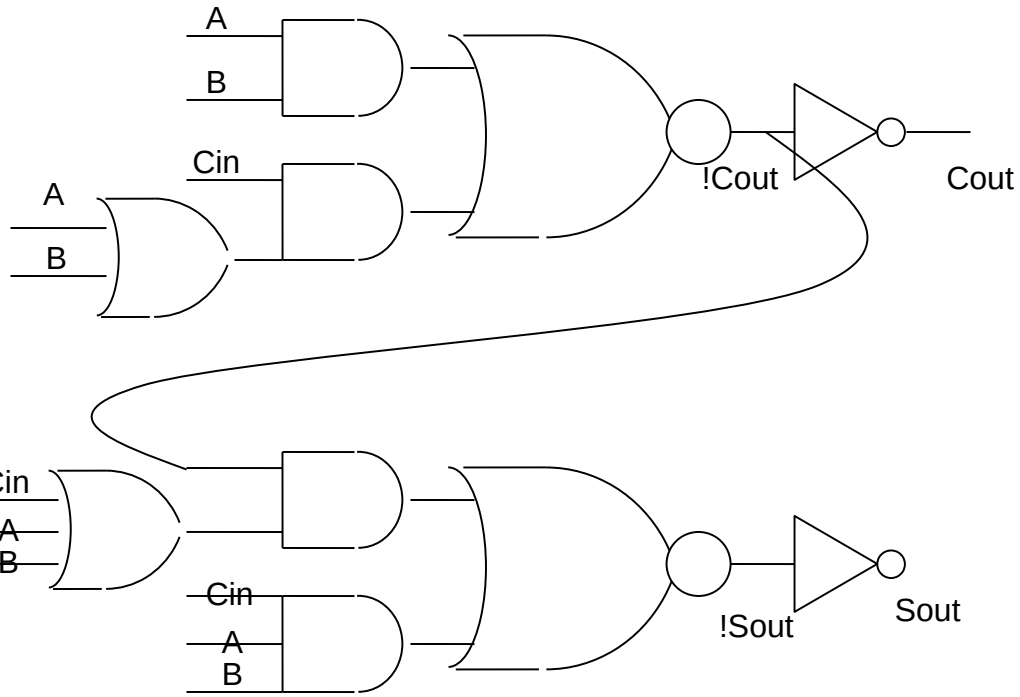
C_{in} \ A,B	00	01	11	10
0	0	**1**	0	**1**
1	**1**	0	**1**	0

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

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

- …

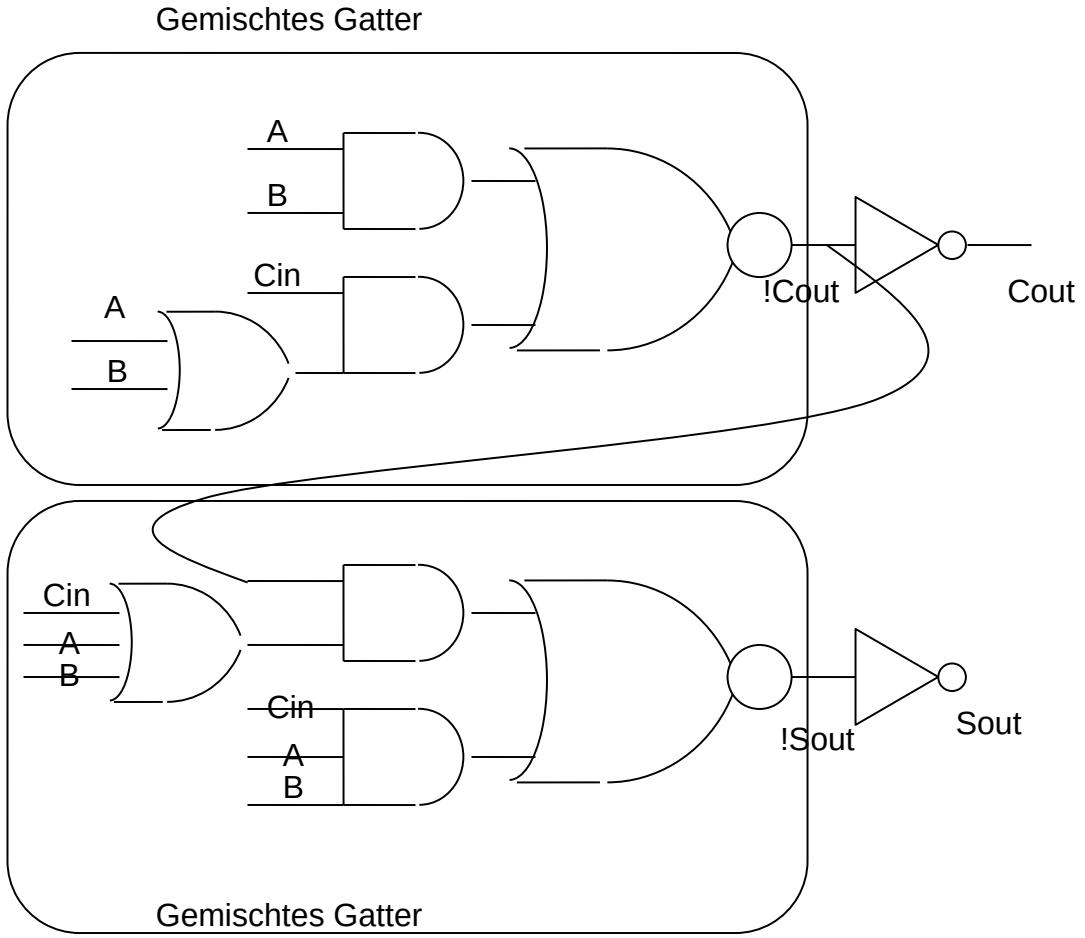
$C_{out} = AB + (A+B)\,C_{in}$:



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

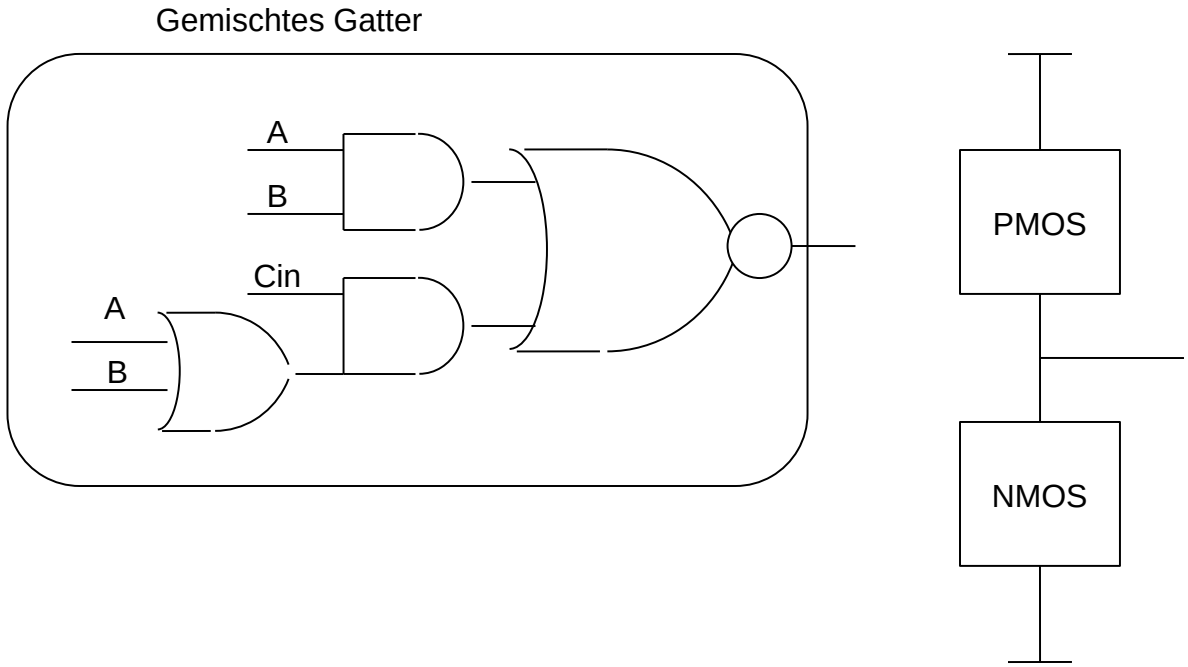
- …

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

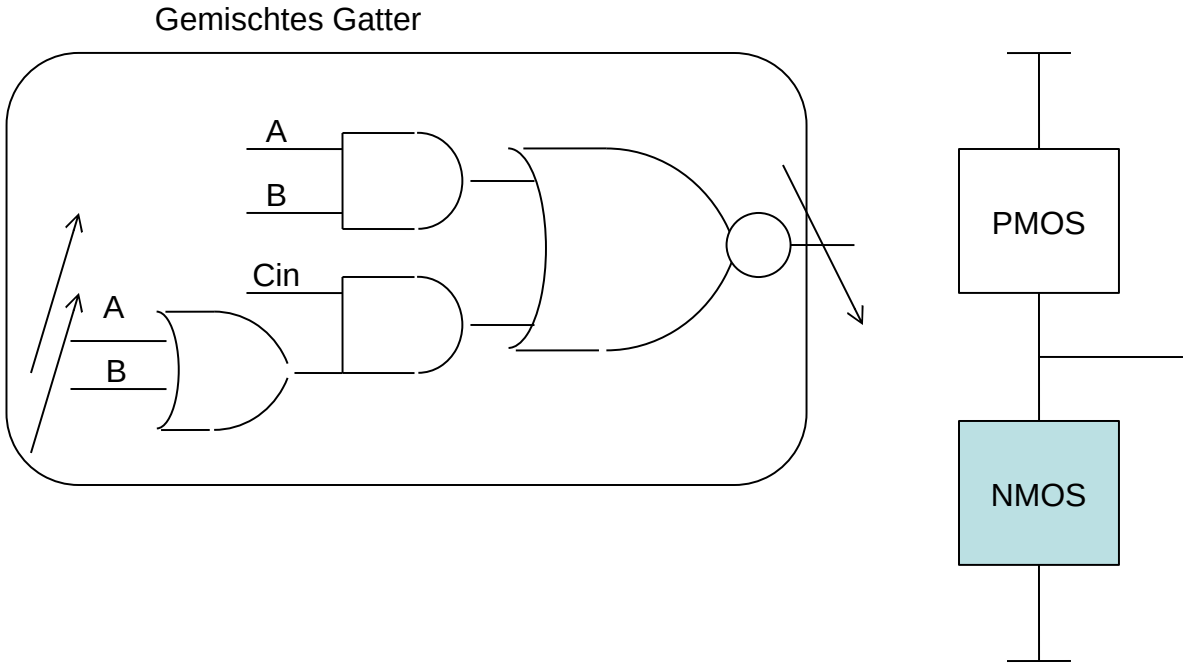


$S = ABC_{in} + (A + B + C_{in}) \cdot !C_{out}$:

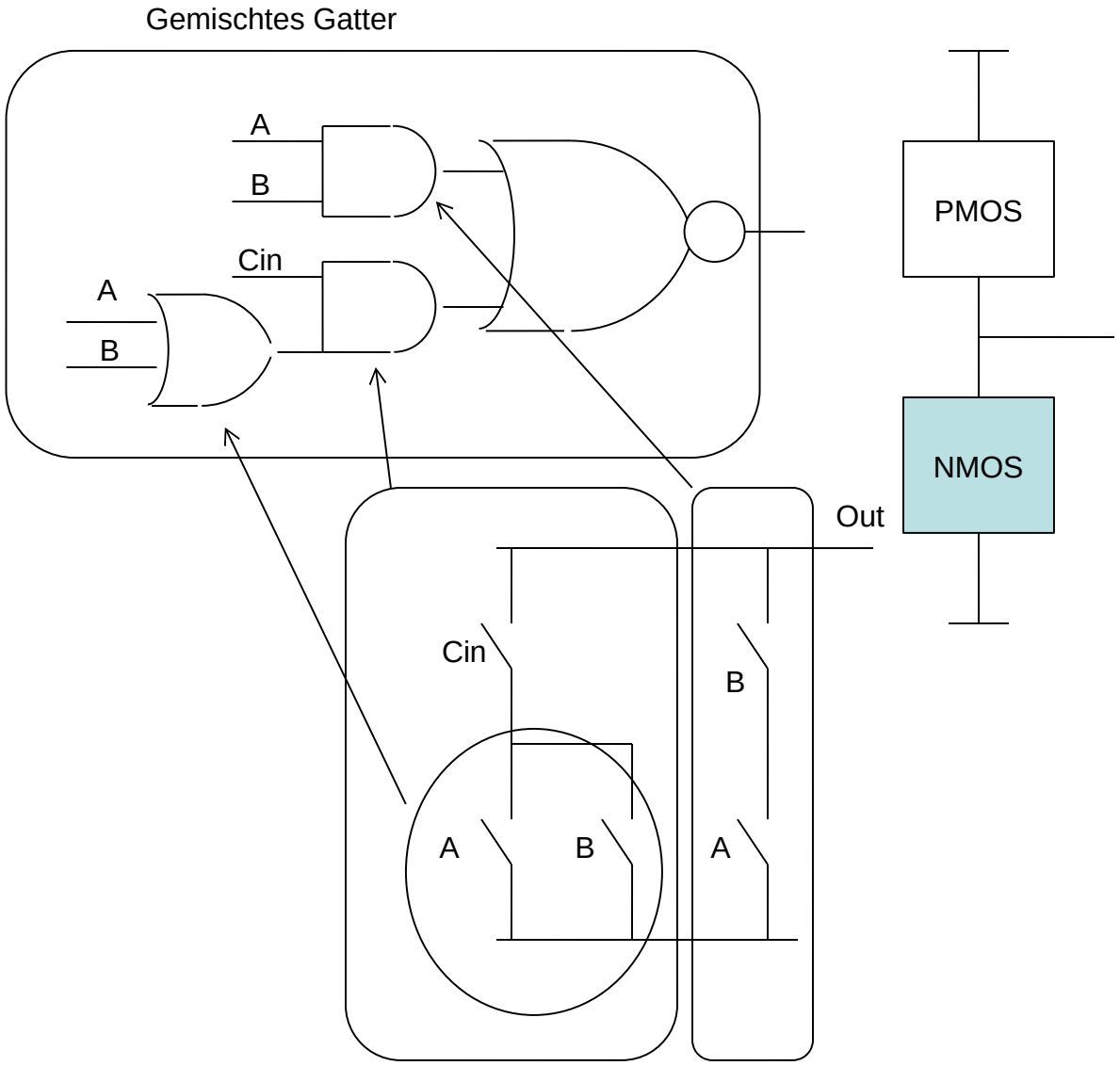
- …


Gemischtes Gatter
A
B
Cin
A
B
PMOS
NMOS

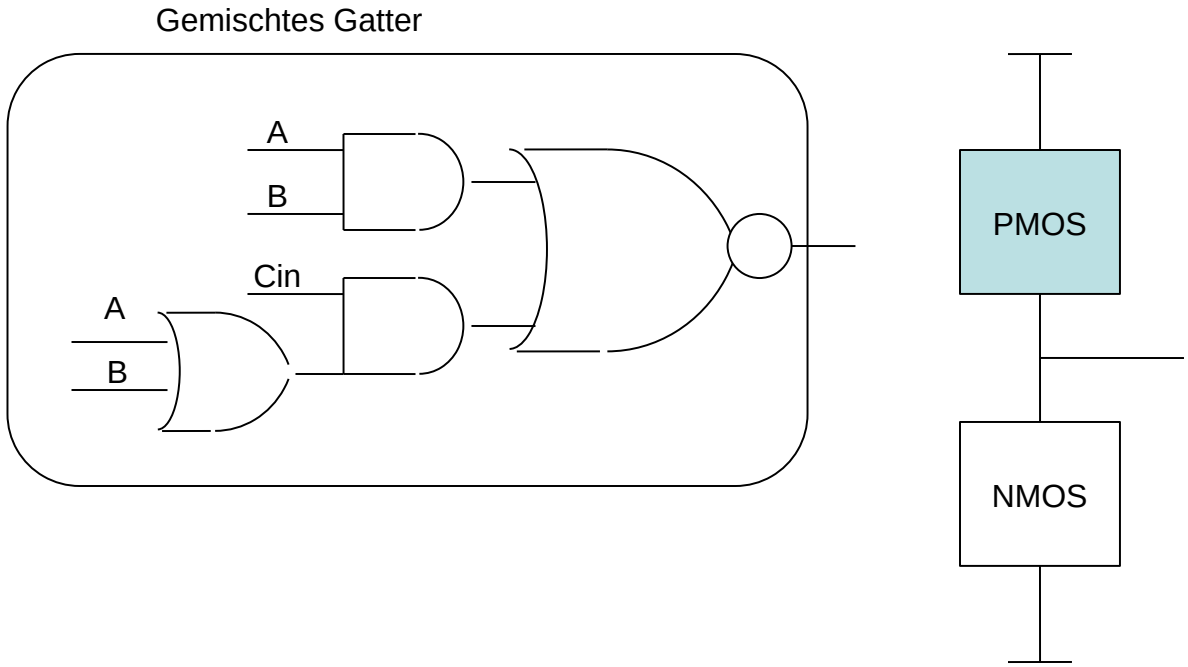
- …


Gemischtes Gatter
A
B
Cin
A
B
PMOS
NMOS

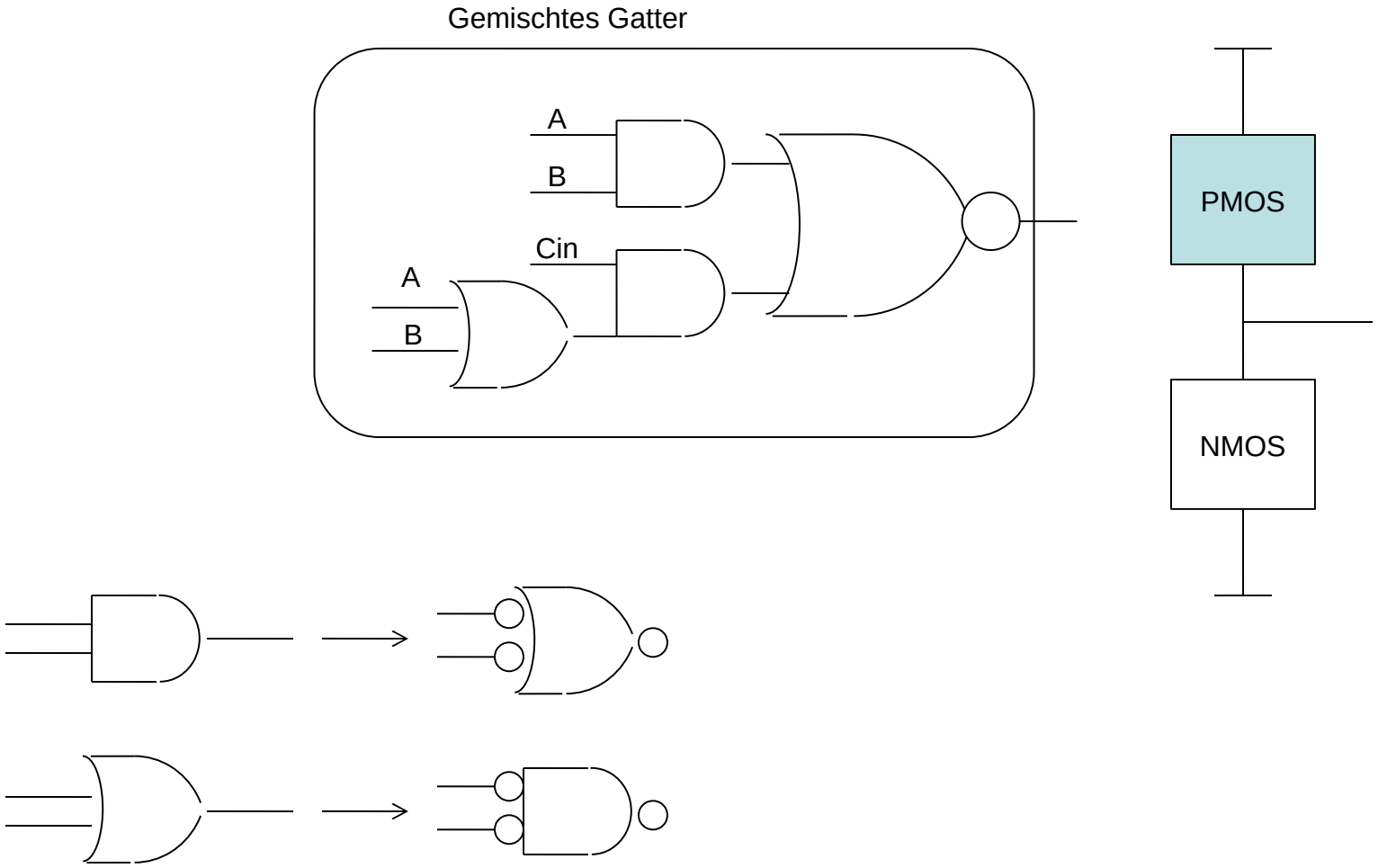
- …


Gemischtes Gatter
A
B
Cin
A
B
PMOS
NMOS
Out
Cin
B
A
B
A

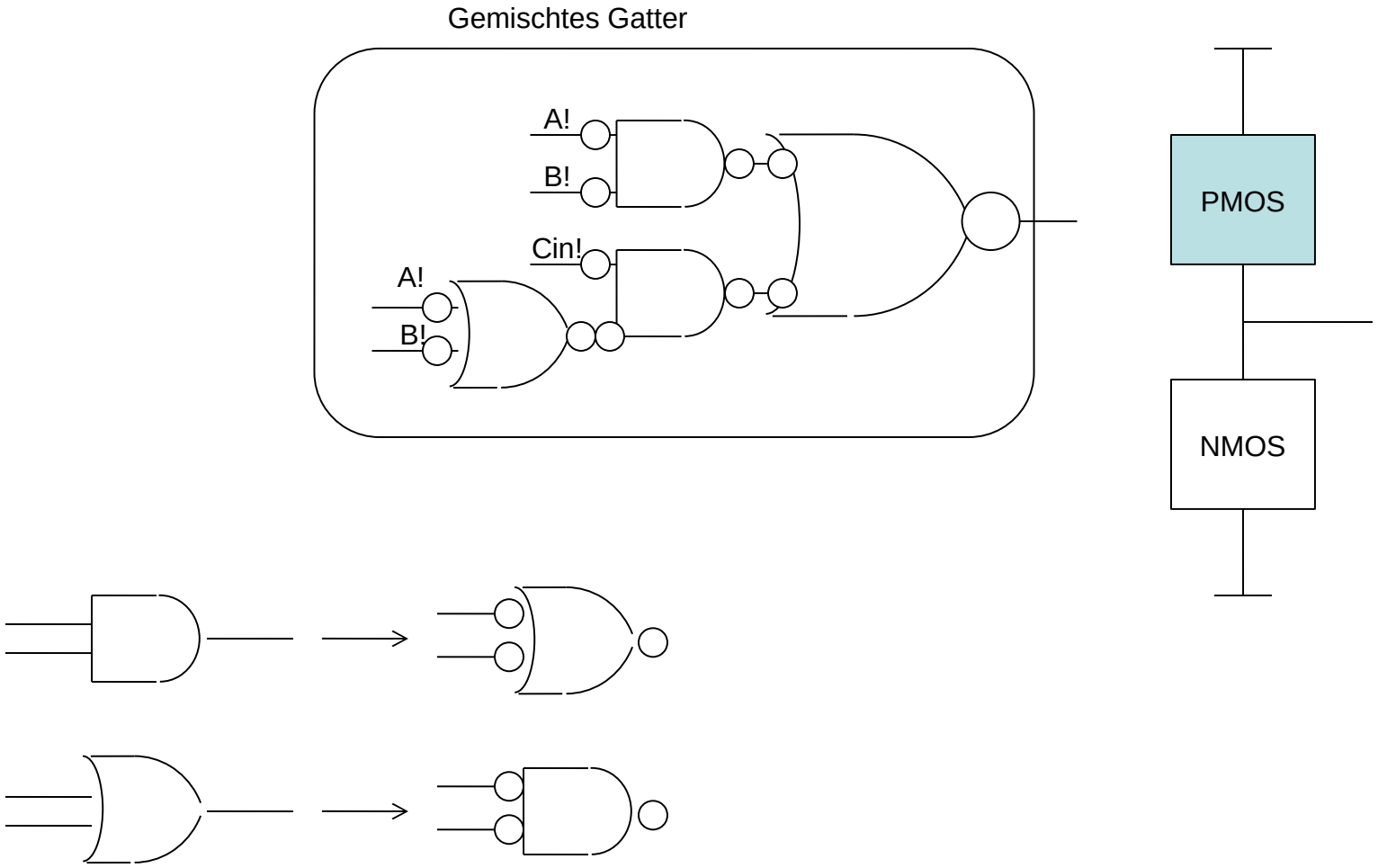
- …


Gemischtes Gatter
A
B
Cin
A
B
PMOS
NMOS

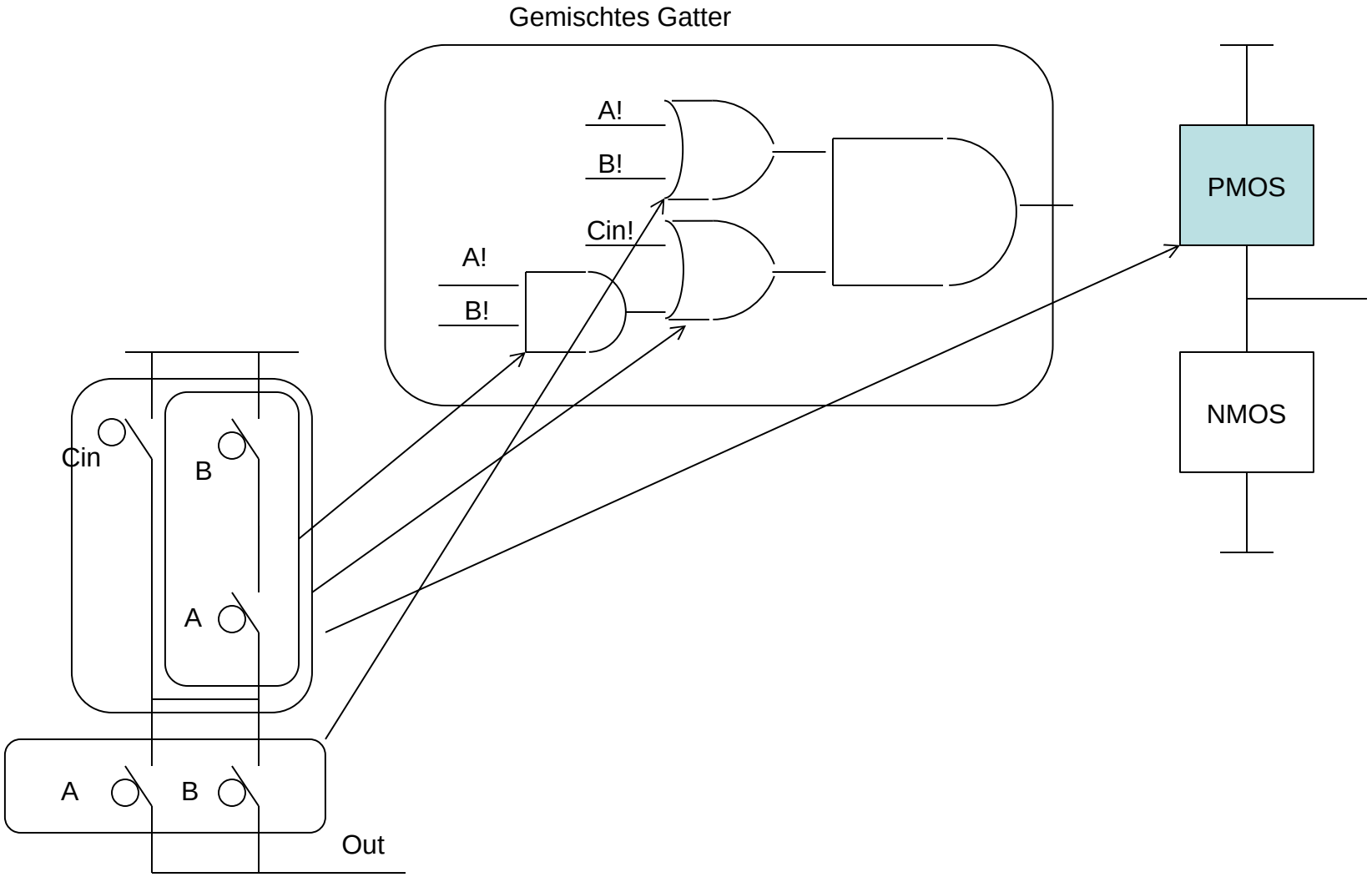
- …


Gemischtes Gatter
A
B
Cin
A
B
PMOS
NMOS

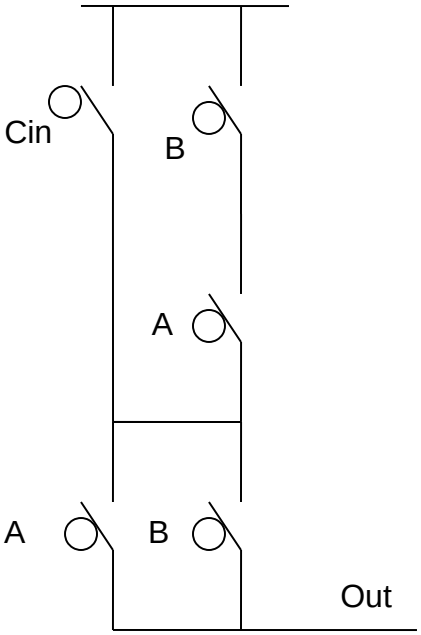
- …

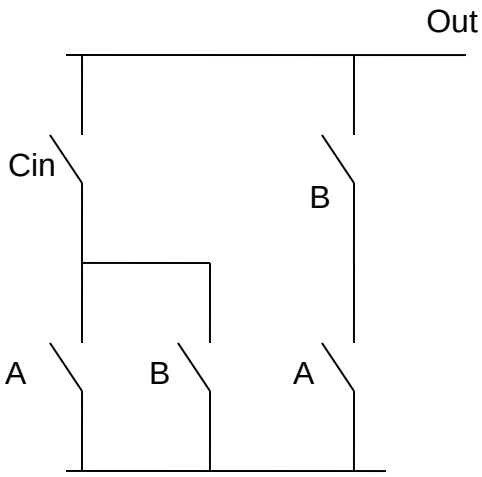

Gemischtes Gatter
A!
B!
Cin!
A!
B!
PMOS
NMOS

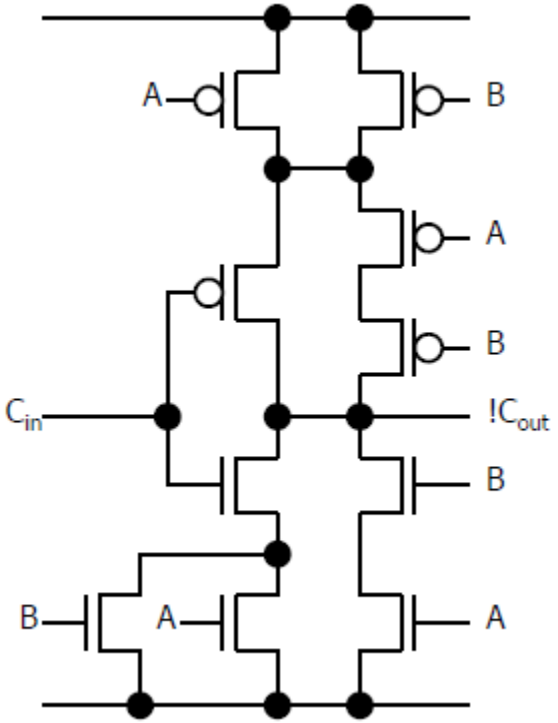
- …


Gemischtes Gatter
A!
B!
Cin!
A!
B!
PMOS
NMOS
Cin
B
A
A
B
Out

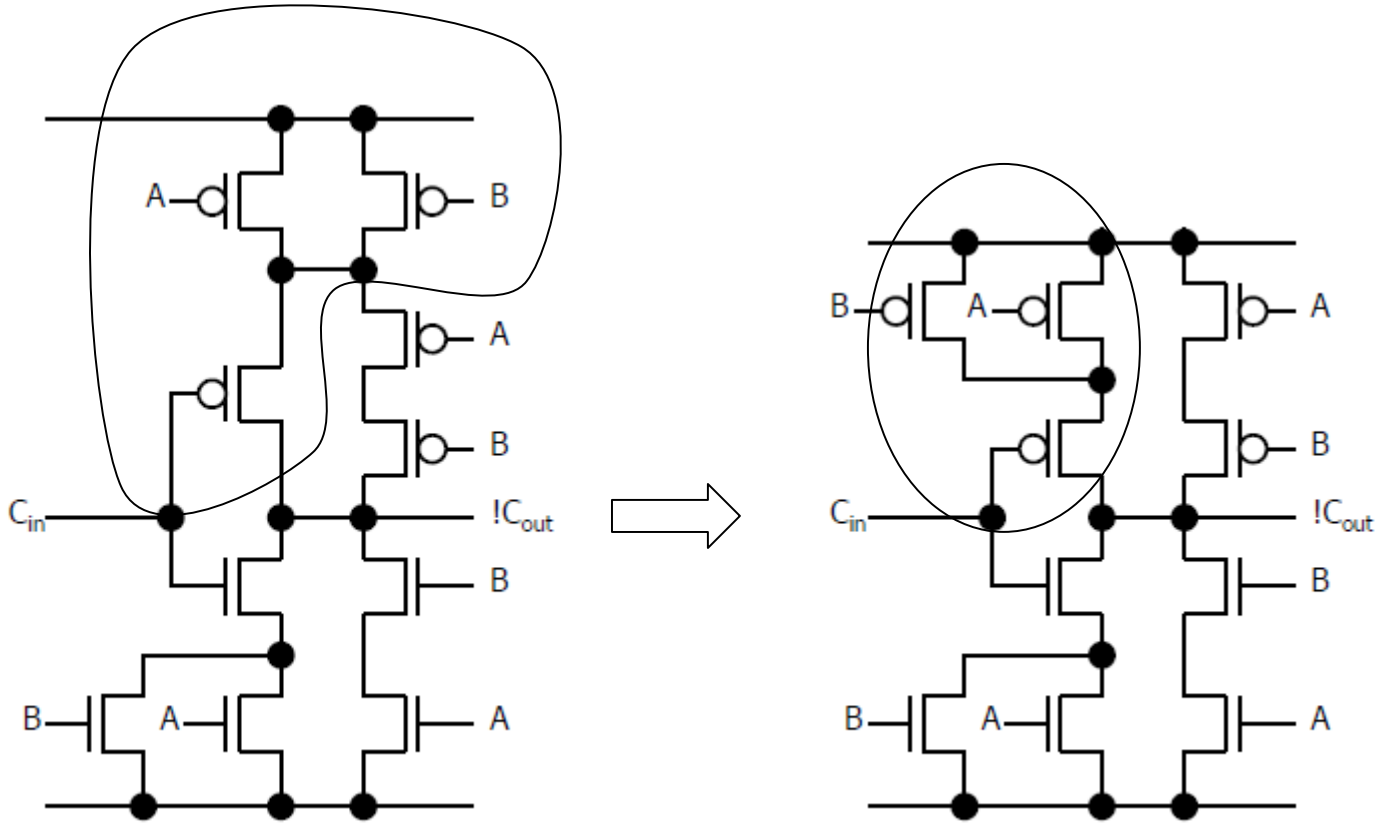
- Das Carry wird durch Cout = AB + (A+B)Cin gegeben.
- Diese Funktion kann mit dem gemischten Gatter Y = !(AB+(A+B)Cin) implementiert werden
- Problem: 3 PMOS übereinander ('Stack height' = 3)



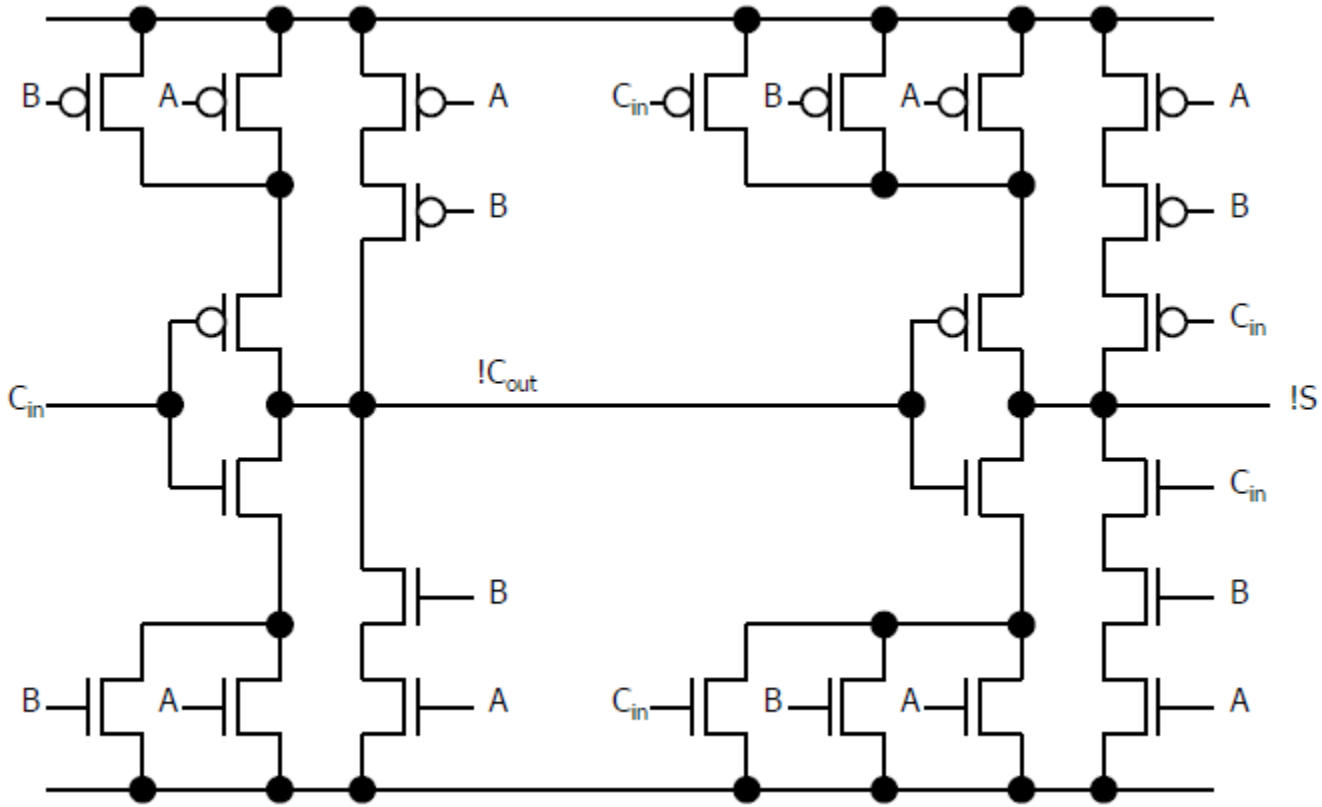




- Der PMOS Zweig kann umgeformt werden:



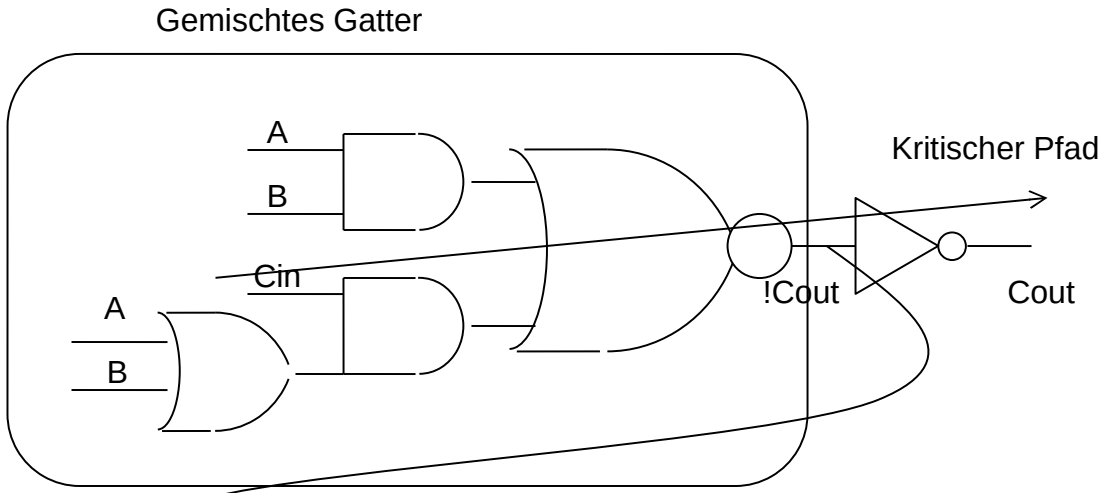
- -> Optimierter Volladierer



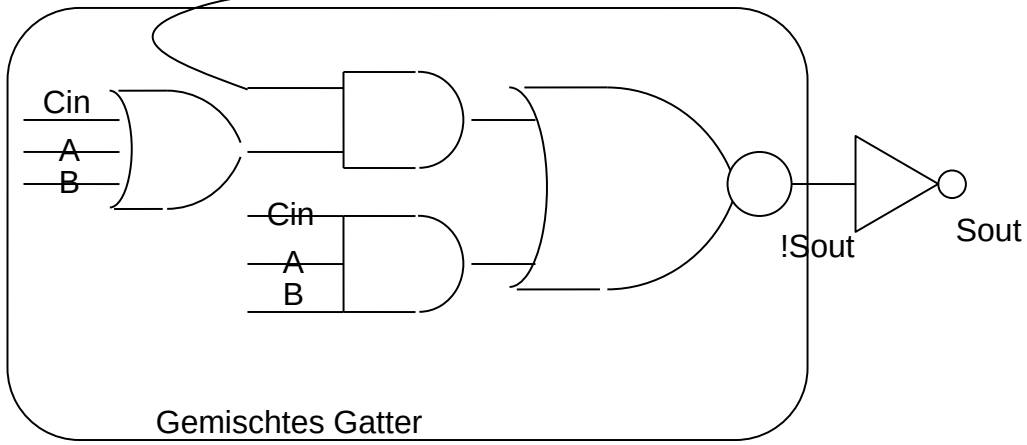
Optimierung: Polaritätswechsel

- …

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

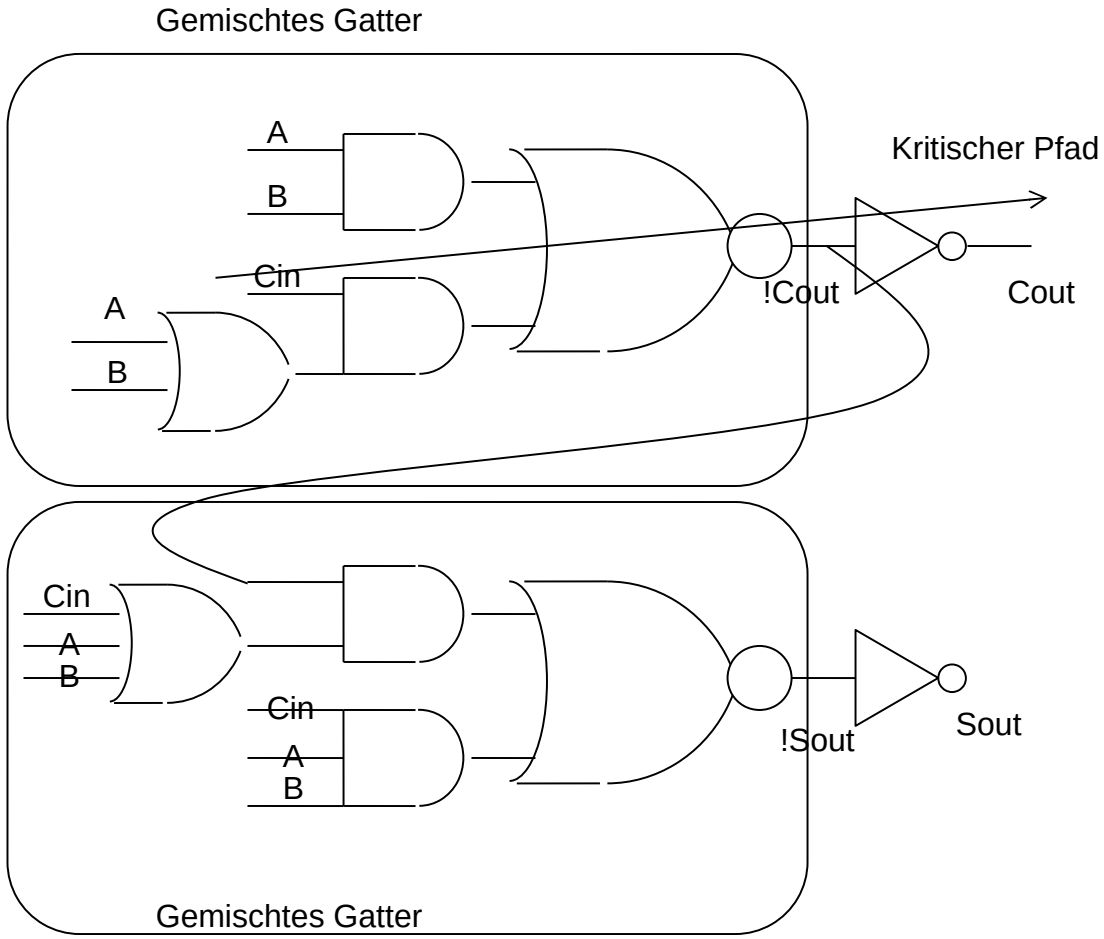


$S = ABC_{in} + (A + B + C_{in}) \cdot !C_{out}$:



- …

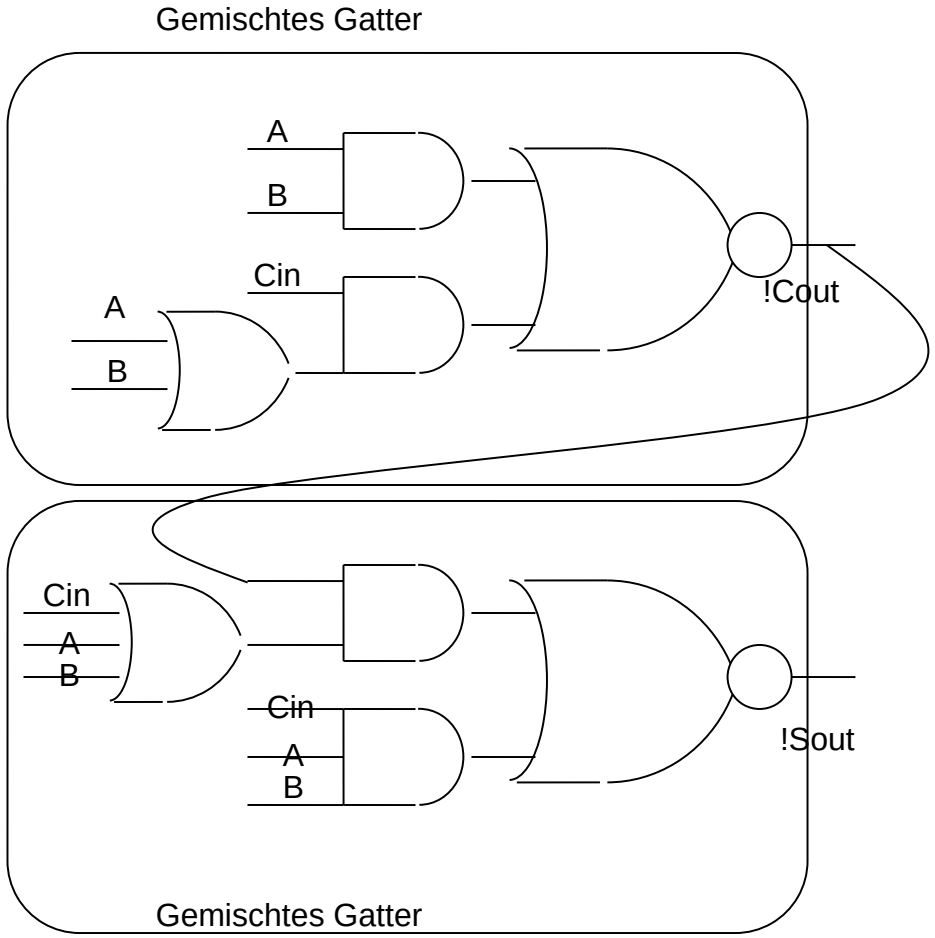
$C_{out} = AB + (A+B)\,C_{in}$:



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

Eine offensichtliche Geschwindigkeitsverbesserung bekommt man, indem man $!C_{out}$ weitergibt und in jeder zweiten Stufe Duale Logik benutzt

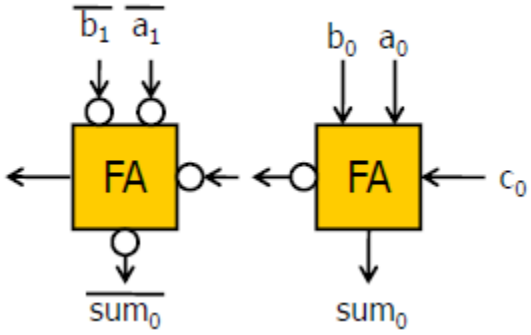
- …



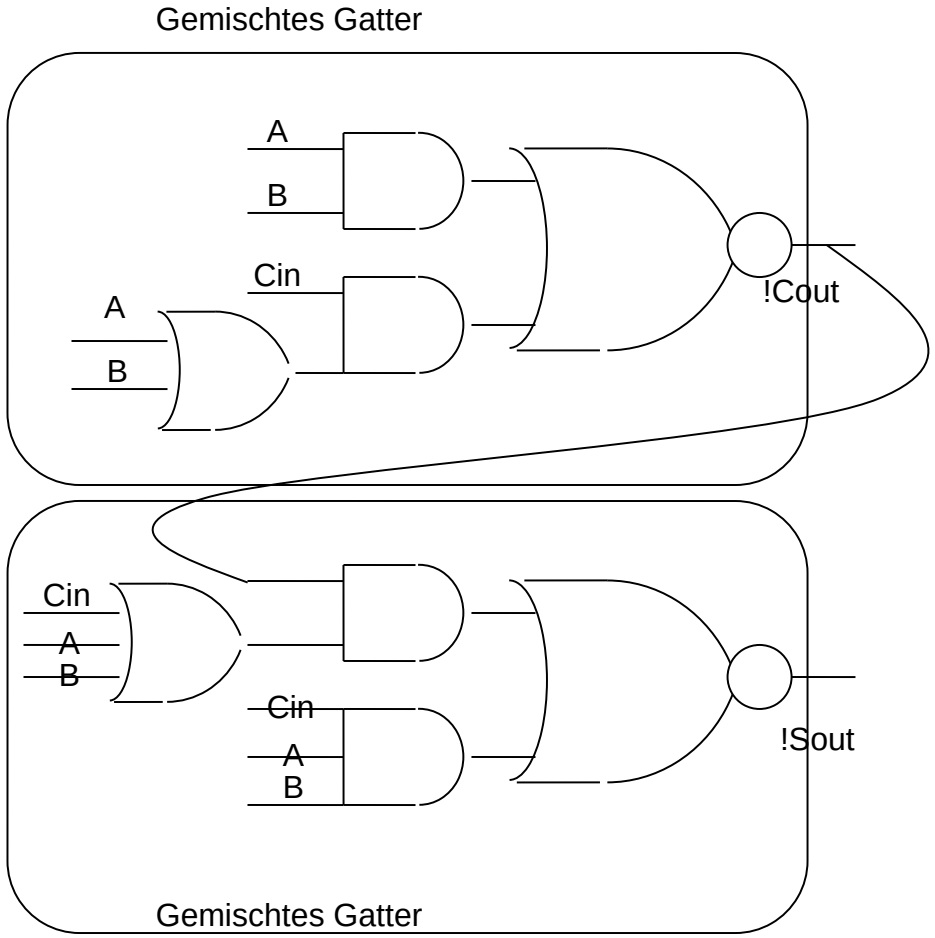
Eine offensichtliche Geschwindigkeitsverbesserung bekommt man, indem man $!C_{out}$ weitergibt und in jeder zweiten Stufe Duale Logik benutzt

Optimierung: Polaritätswechsel

- …

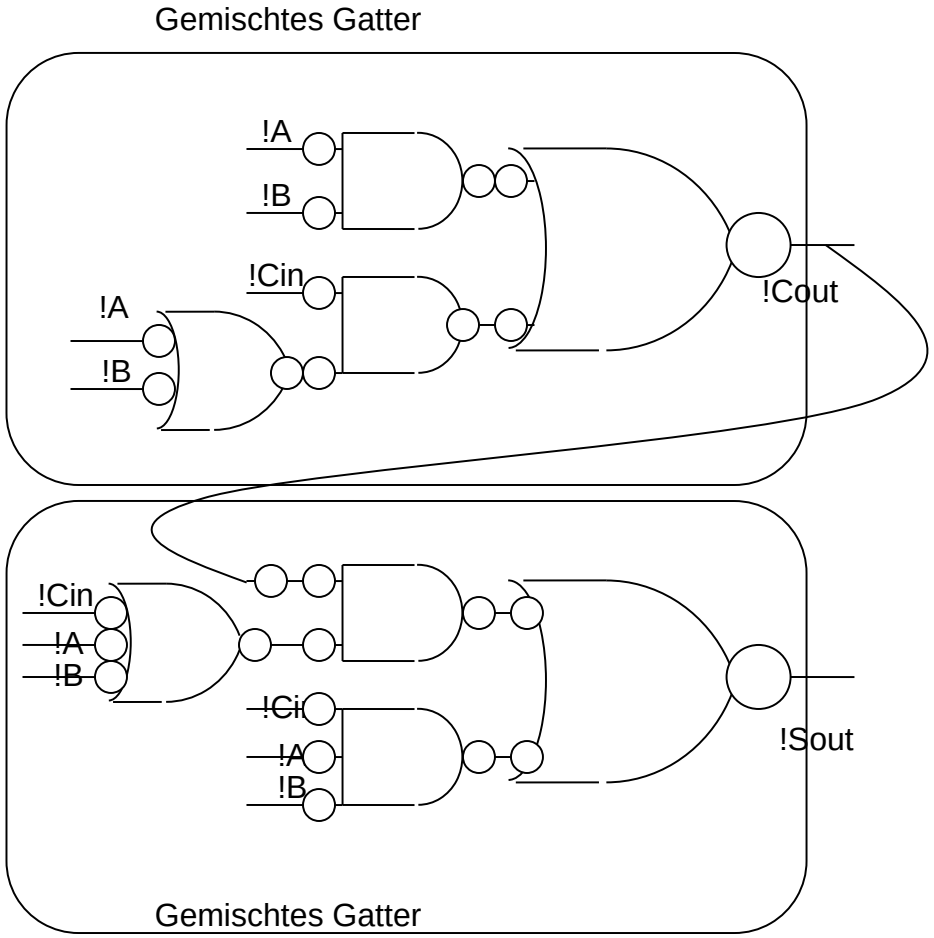


- …



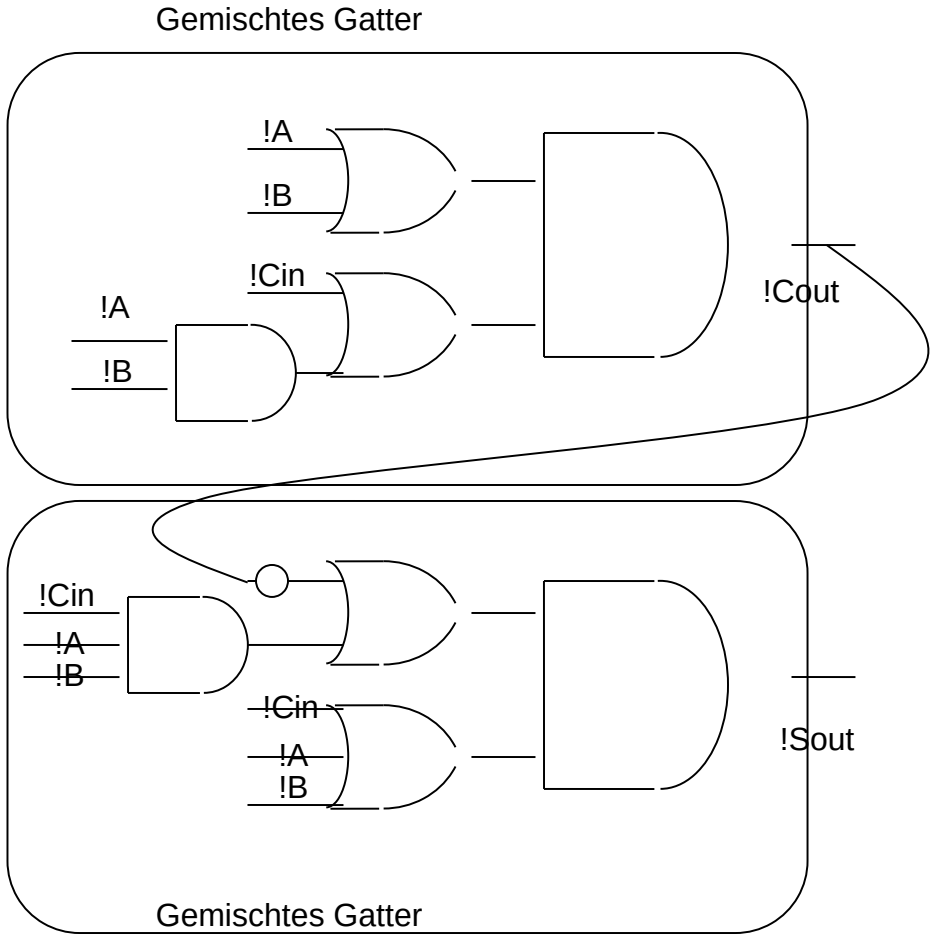
Eine offensichtliche Geschwindigkeitsverbesserung bekommt man, indem man $!C_{out}$ weitergibt und in jeder zweiten Stufe Duale Logik benutzt

- …



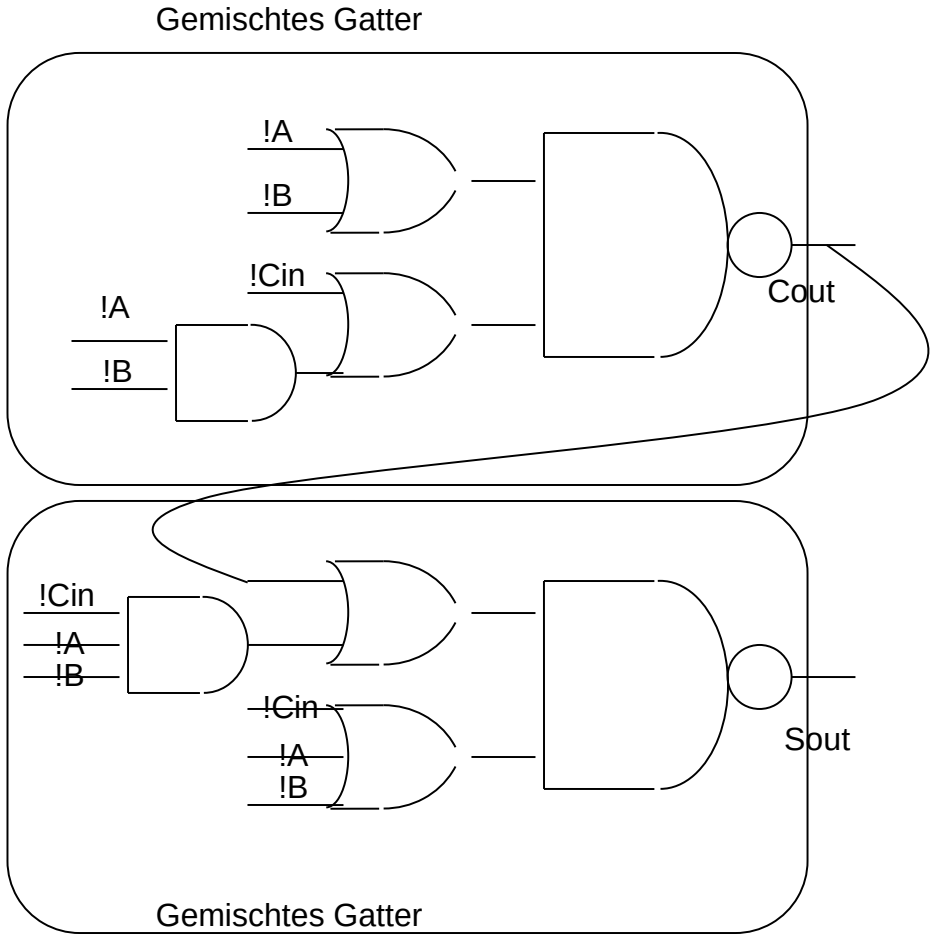
Eine offensichtliche Geschwindigkeitsverbesserung bekommt man, indem man $!C_{out}$ weitergibt und in jeder zweiten Stufe Duale Logik benutzt

- …



Eine offensichtliche Geschwindigkeitsverbesserung bekommt man, indem man $!C_{out}$ weitergibt und in jeder zweiten Stufe Duale Logik benutzt

- …



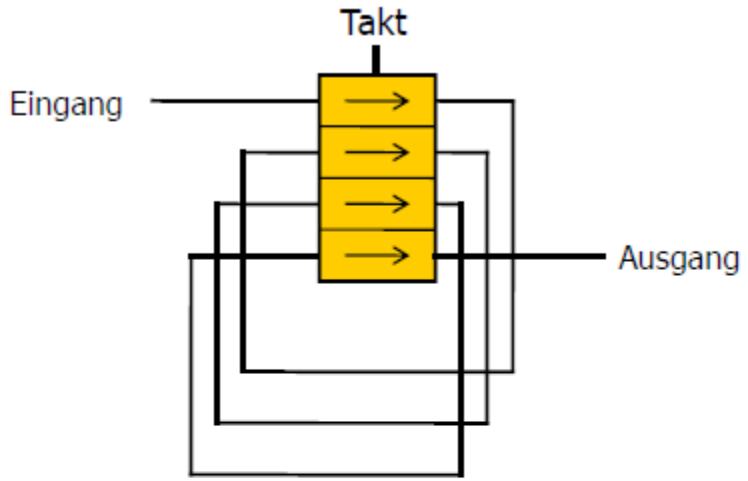
Eine offensichtliche Geschwindigkeitsverbesserung bekommt man, indem man $!C_{out}$ weitergibt und in jeder zweiten Stufe Duale Logik benutzt

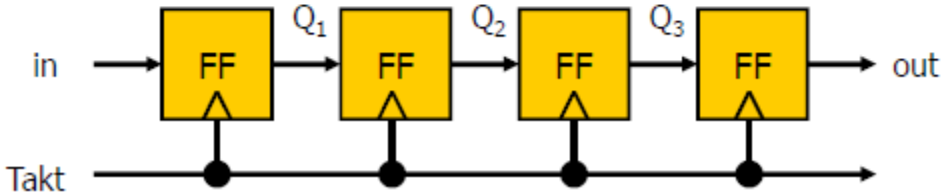
Getaktete Schaltungen

Schieberegister

Schieberegister

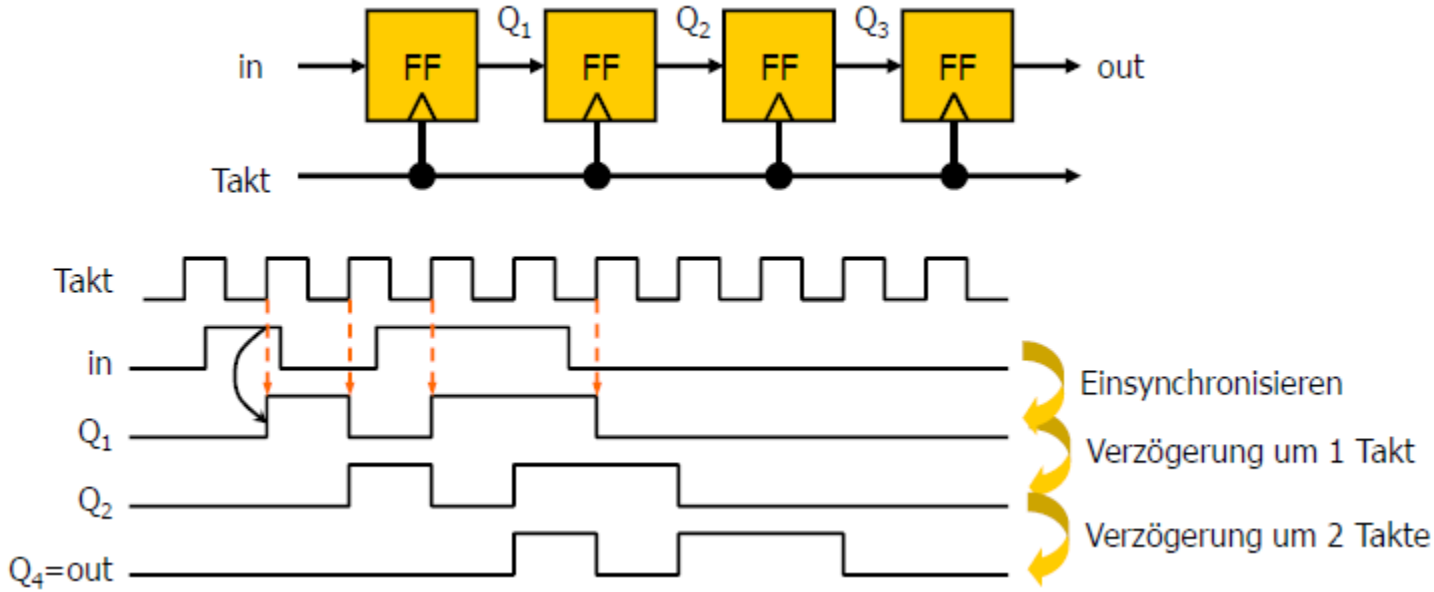
- Schieberegister: Sehr einfach, keine Logik, ein Eingang, ein Ausgang



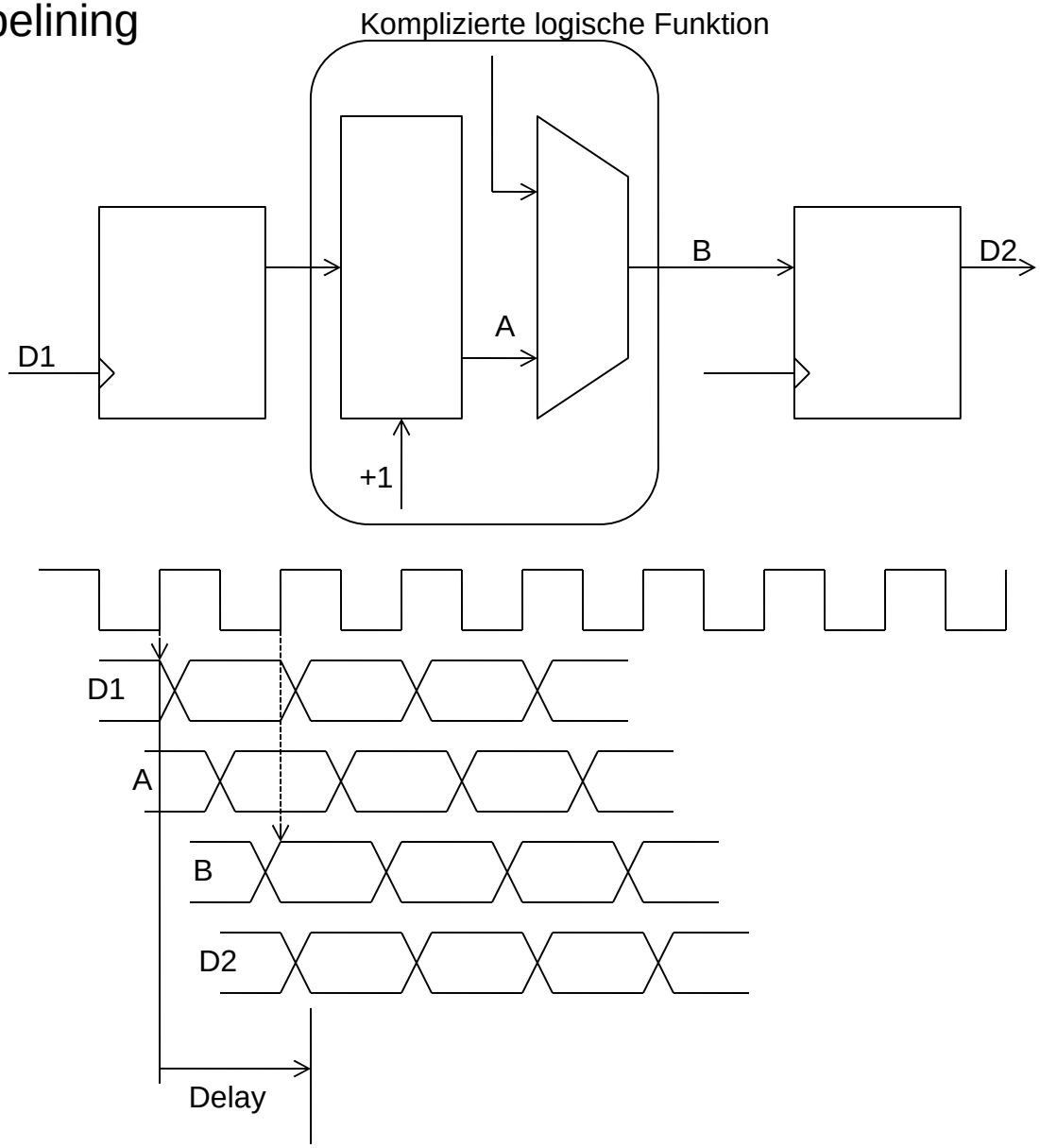


Schieberegister

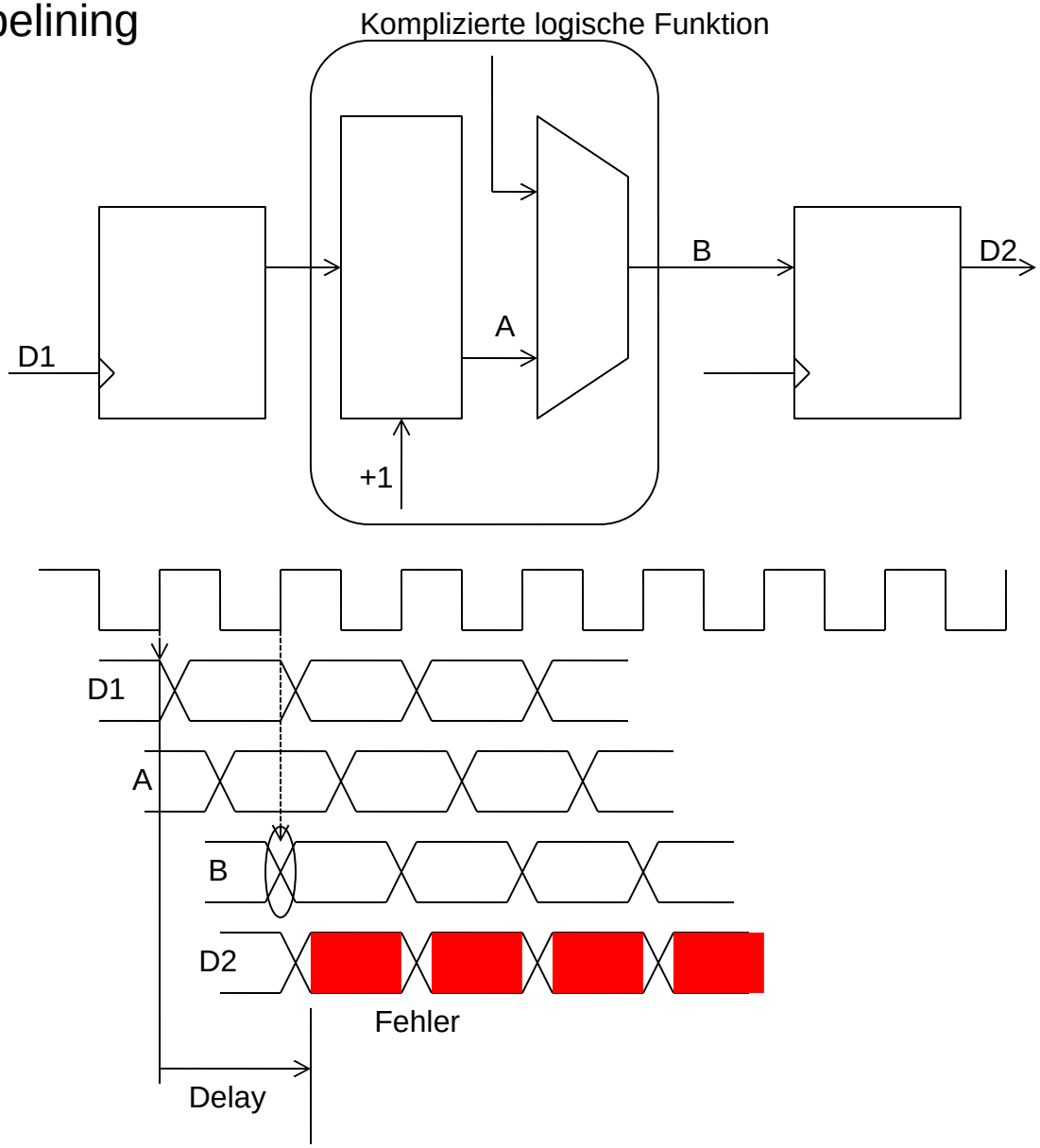
- Schieberegister entstehen durch Hintereinanderschalten von FFs.
- Zwischen den Stufen ist keine (wenig) Logik
- **Vorsicht**: Die Hold-Zeit kann leicht verletzt sein. Daher fügt man manchmal Verzögerungen (Inverterketten) in den Datenpfad ein.
- Anwendungen:
  - Verzögerung von Signalen (z.B. bei Pipelining)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



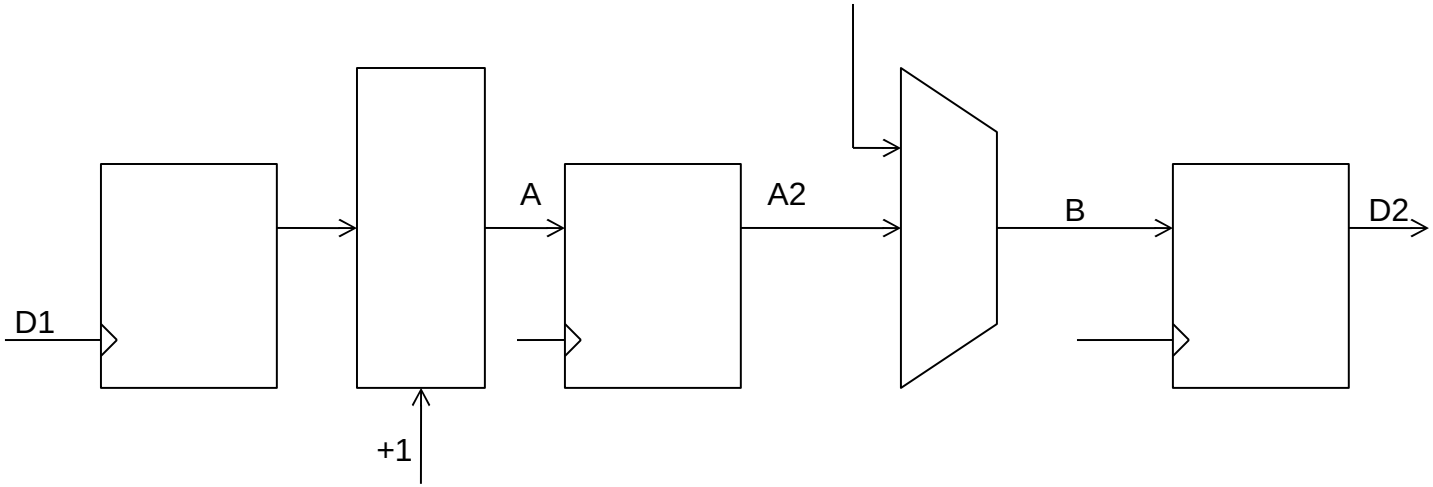
- Pipelining

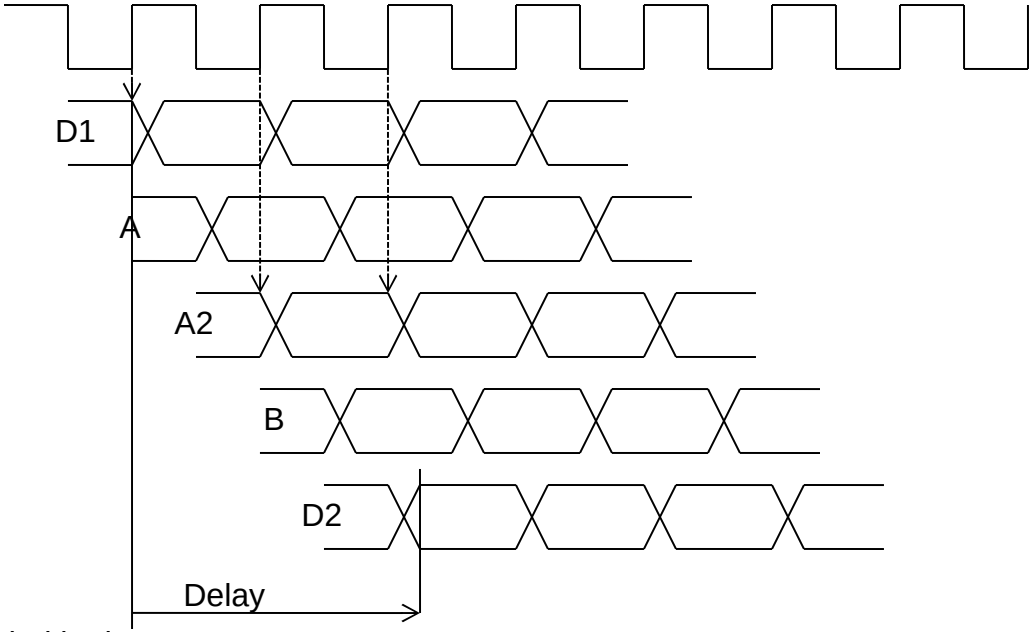


- Pipelining


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

- Pipelining

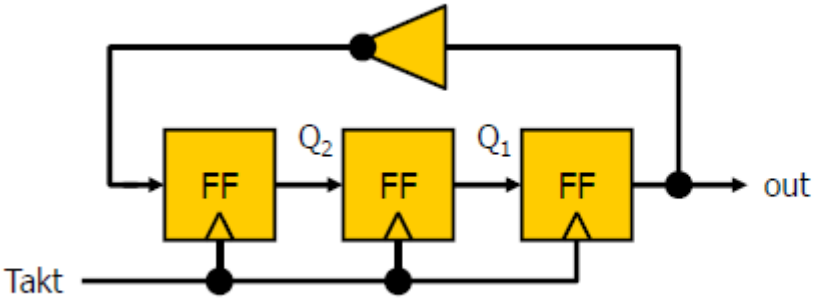


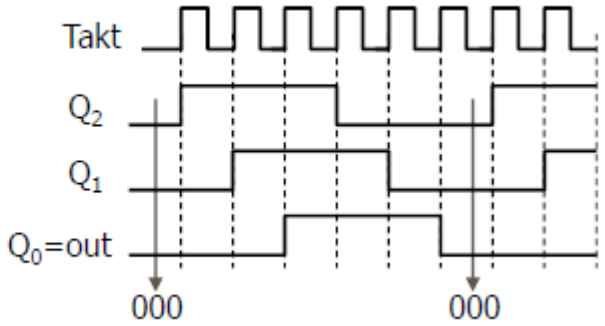


Zähler

Linear Feedback Shift Register (LFSR)

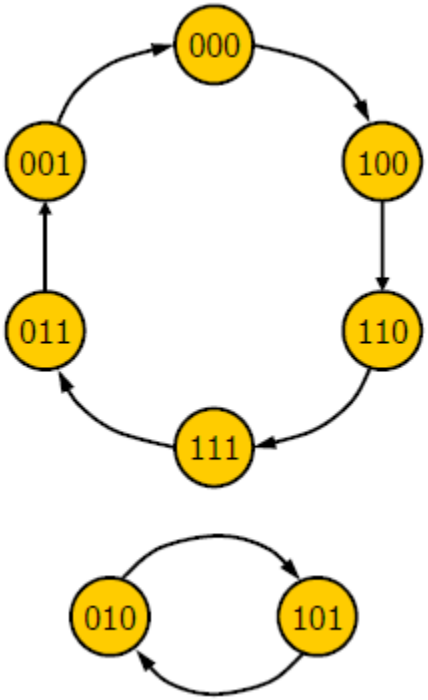
- Sehr **einfach aufgebaute Zähler** werden durch **Linear Feedback Shift Register (LFSR)** erzeugt
- Das Zurücksetzen in einen Anfangszustand kann durch sync/async. Reset der FFs erfolgen
- Beim ‚Johnson Zähler' wird der Ausgang über einen Inverter zum Eingang rückgekoppelt.
- Der Zähler hat dadurch 2N Zustände





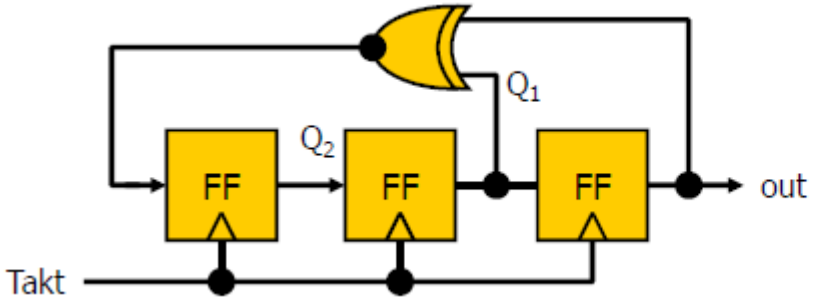
Johnson Zähler: Sprungdiagramm

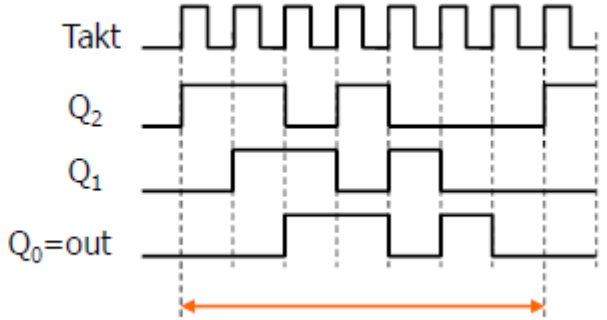
- Bei N=3 gibt es 2^3 = 8 mögliche Zustände.
- 6 davon werden vom Johnson Zähler durchlaufen:
- Die verbleibenden beiden Zustände bilden einen eigenen Zyklus.
- Man muss mit einem Reset vermeiden hier zu starten!



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





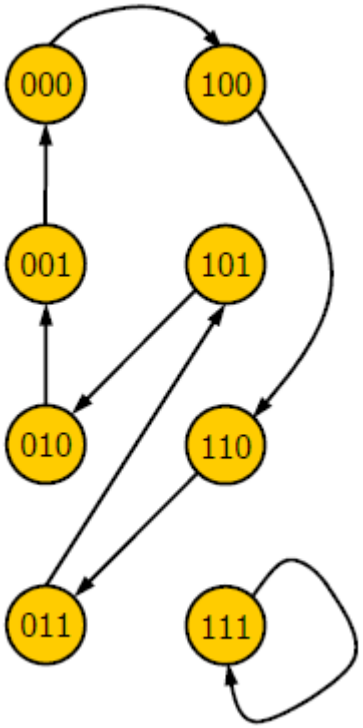
PRBS

- 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.
- Beispiel für N = 6 (Periode 63)
  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	$5x10^{11}$	

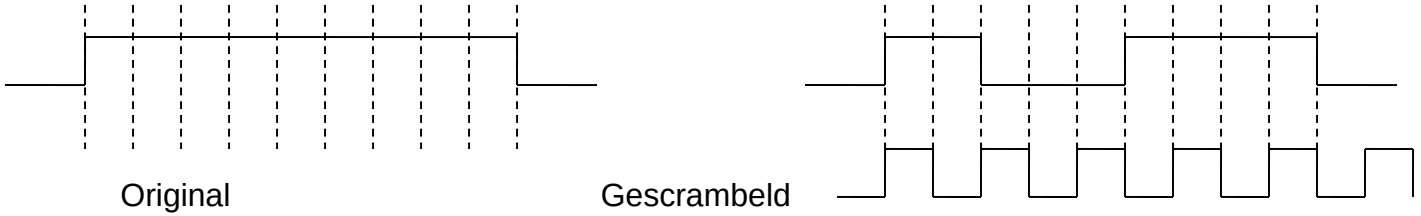
PRBS

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



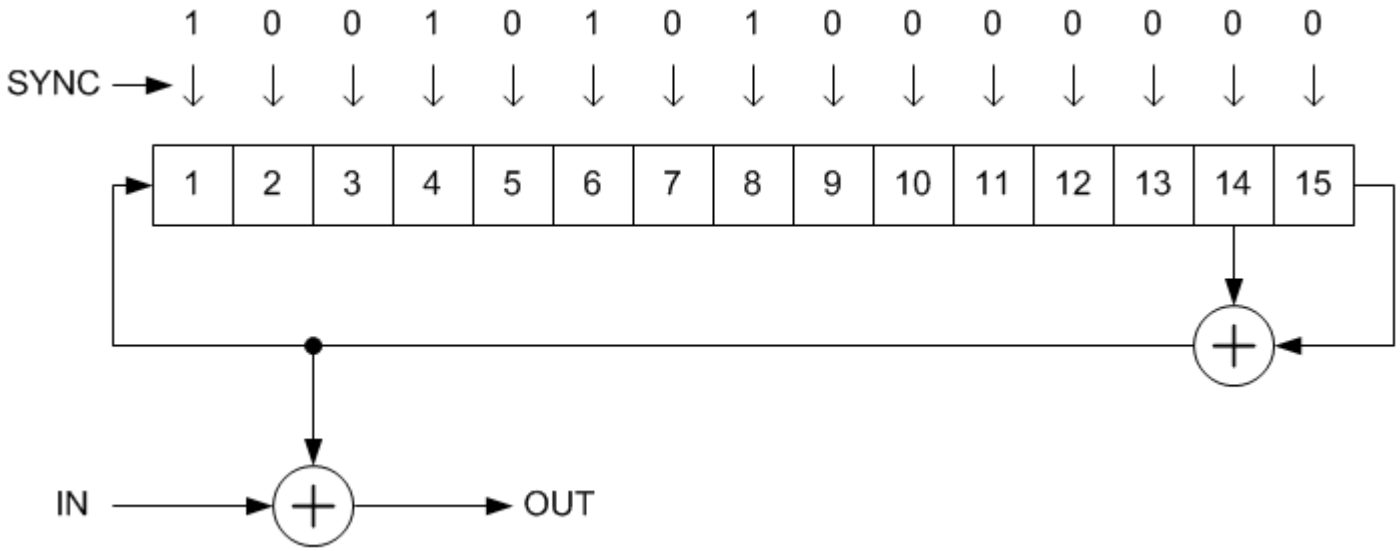
Scrambler

- Ein **Scrambler** (deutsch *Verwürfler*) verwendet linear rückgekoppelte Schieberegister (LFSR) oder fixe Tabellen, um ein Digitalsignal nach einem relativ einfachen Algorithmus umkehrbar umzustellen.
- Ein Scrambler basierend auf fixen Tabellen bzw. 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

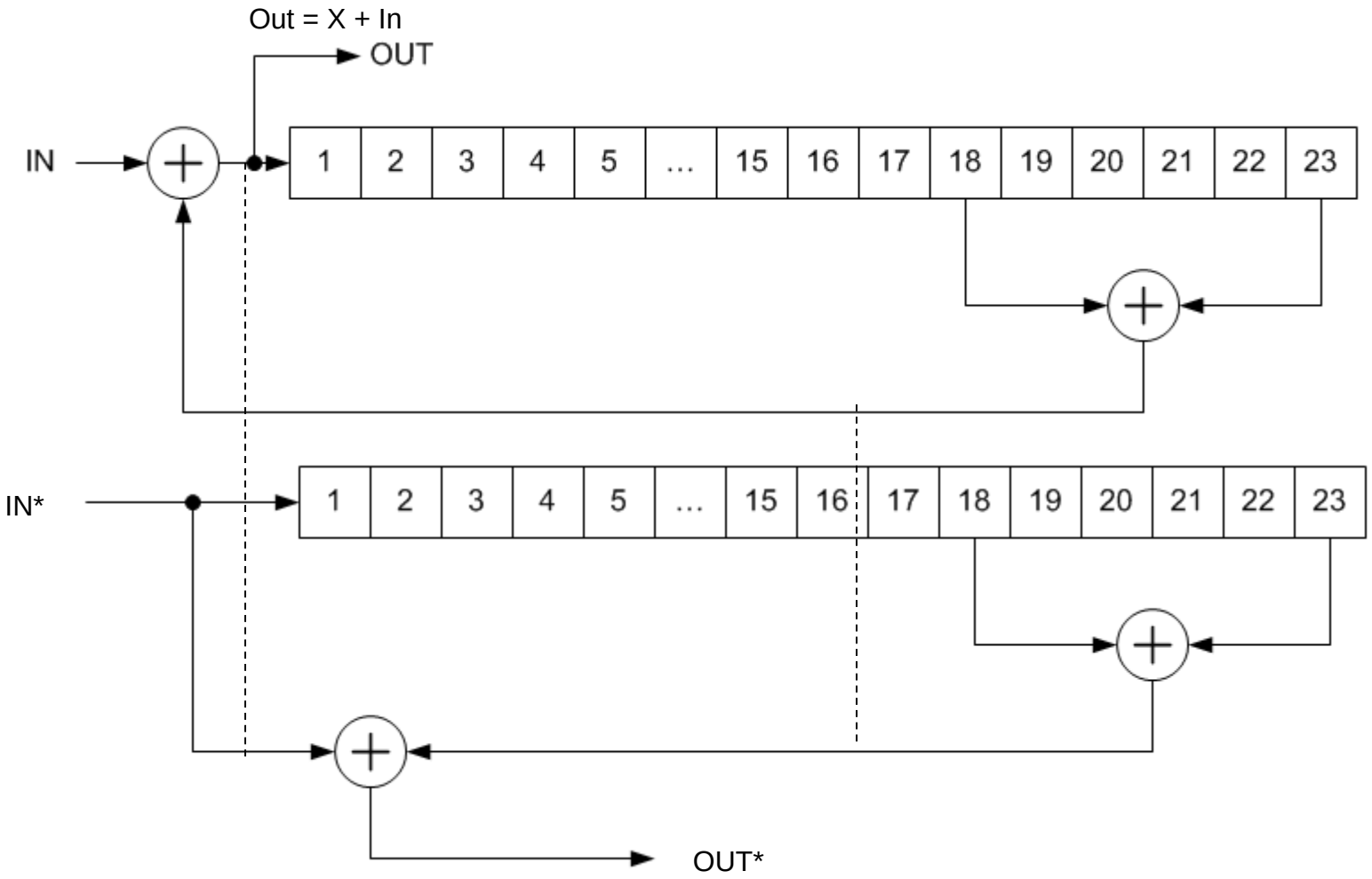


Selbstsynchronisierende (multiplikative) Scrambler

- 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

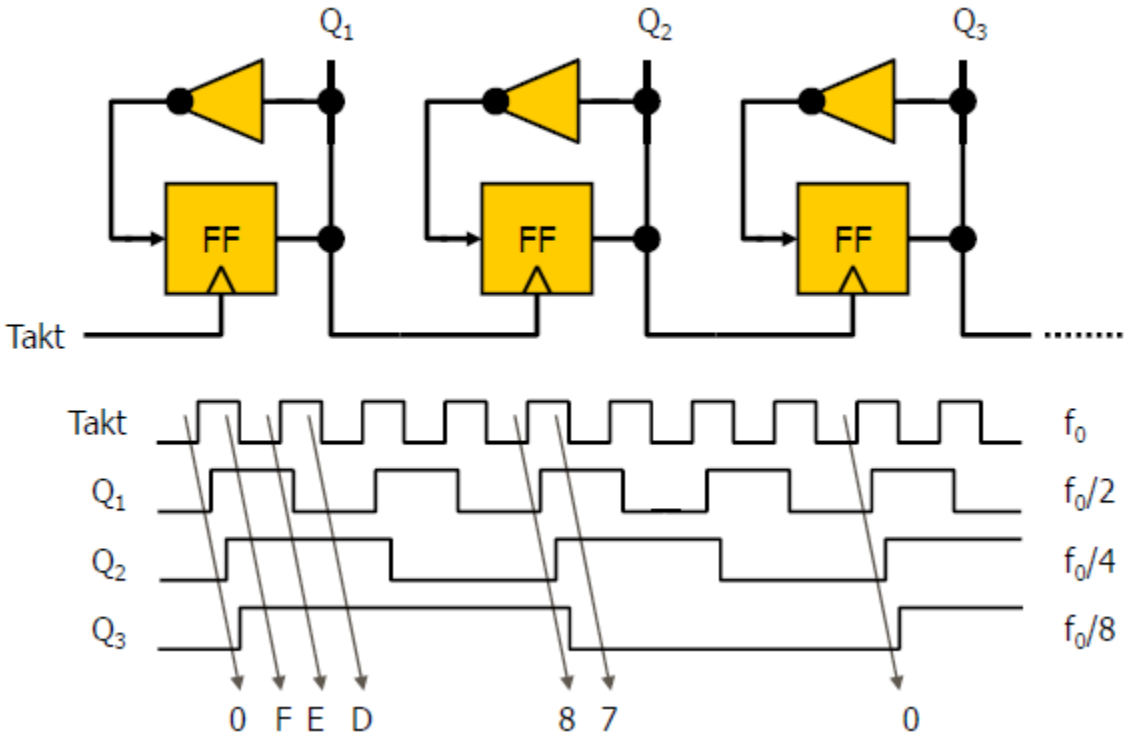
- …



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

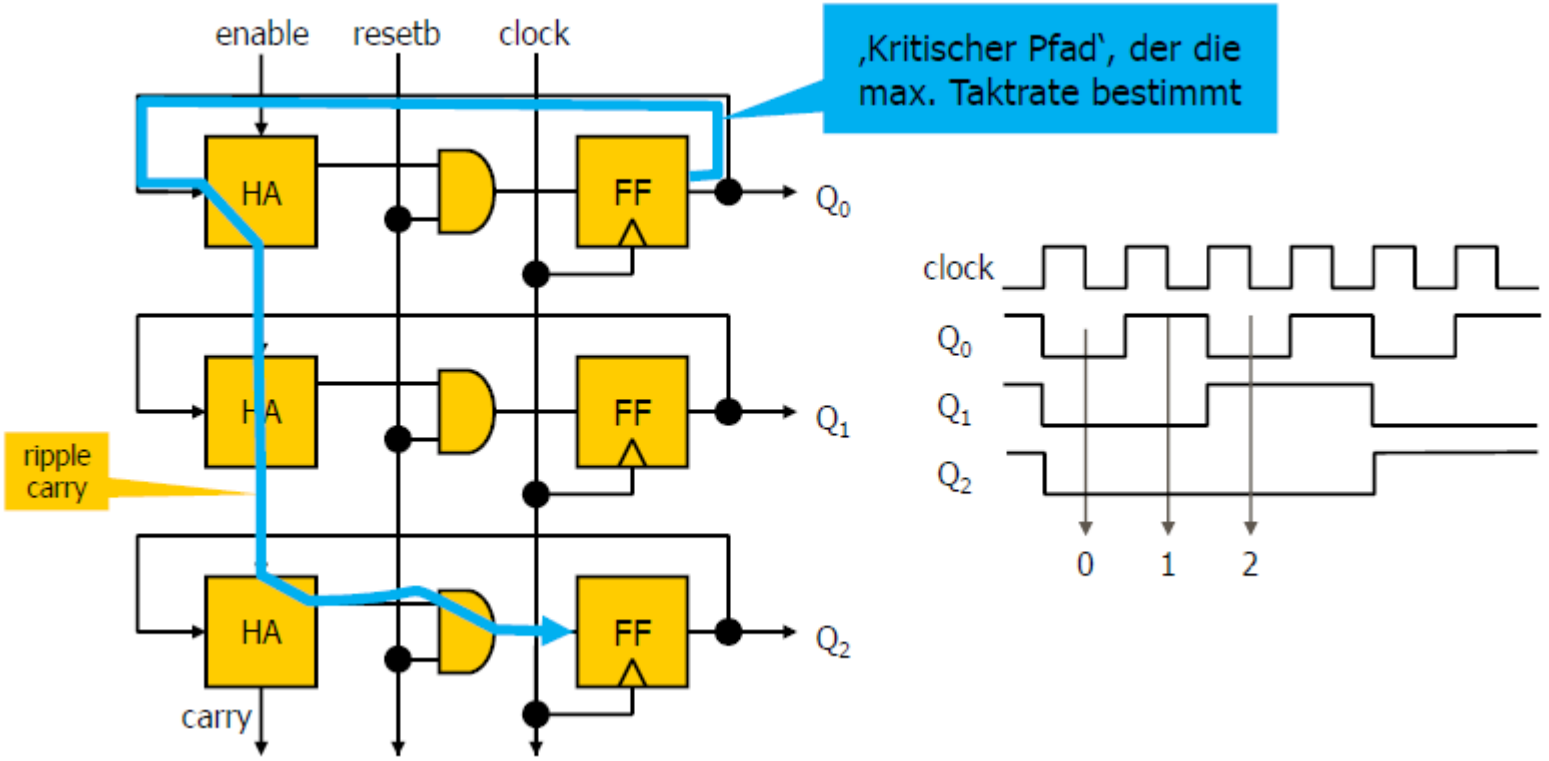
Asynchrone Binärzähler (Ripple Counter)

- 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



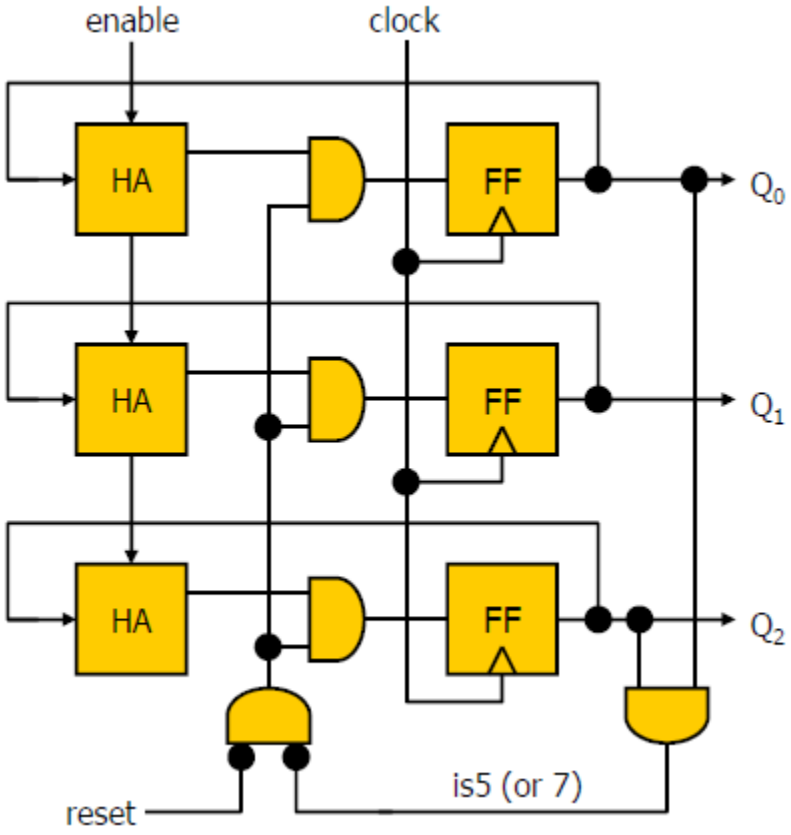
Synchrone Binärzähler

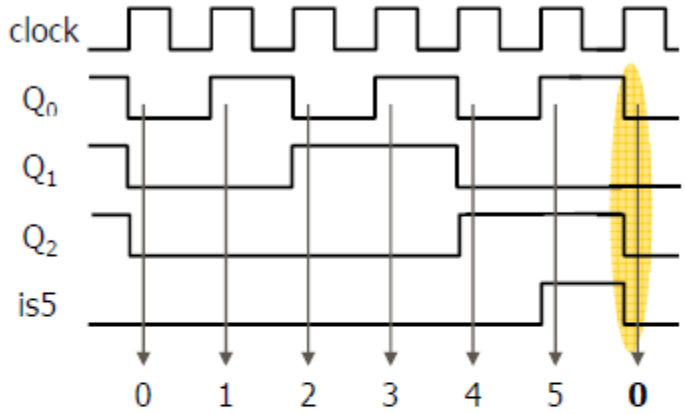
- Alle FFs werden gleichzeitig getaktet
- Die Eingänge werden so beschaltet, daß sich (z.B.) aufsteigend Binärzahlen ergeben
- Implementierung mit Halbaddierern (mit enable und reset)
- Max. Taktfrequenz ist durch die Laufzeit des 'ripple' Carry begrenzt.

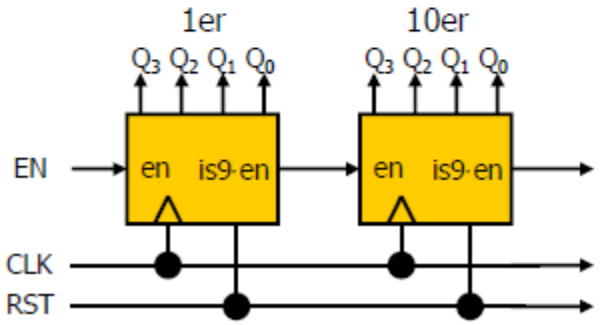


Kürzere synchrone Binärzähler (z.B. BCD Zähler)

- 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.

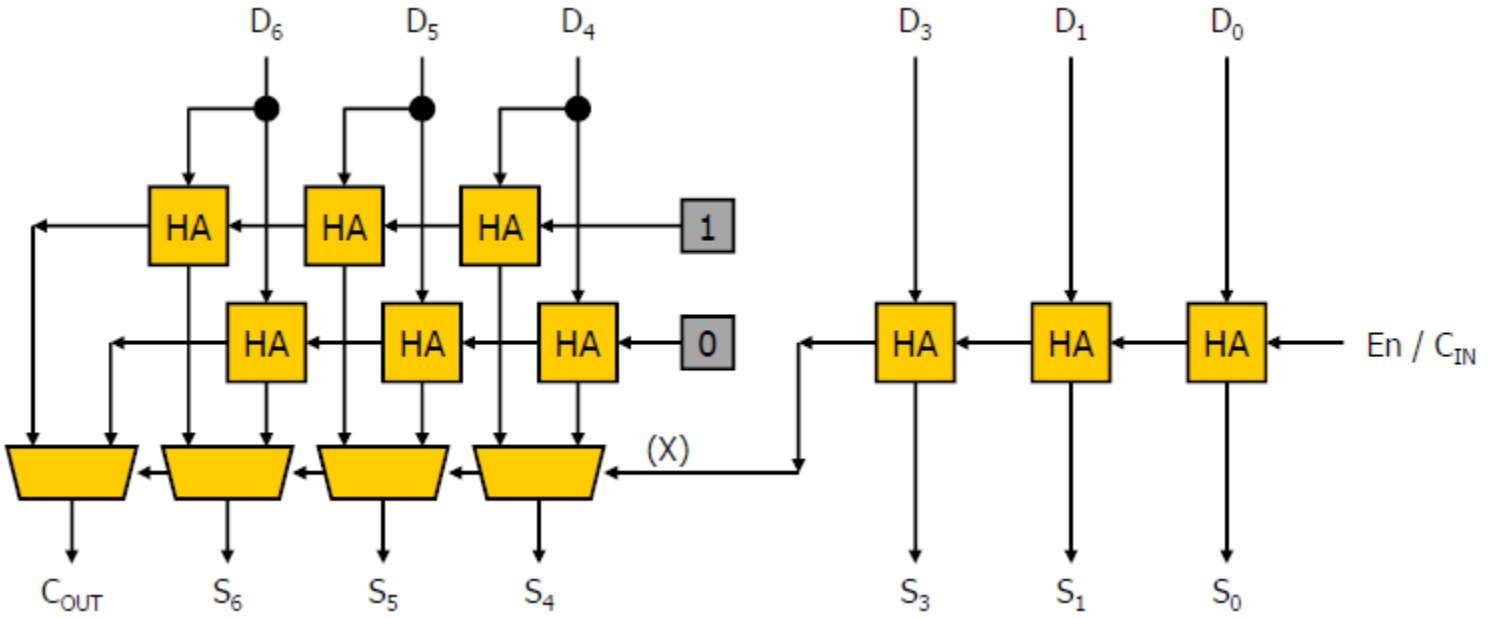






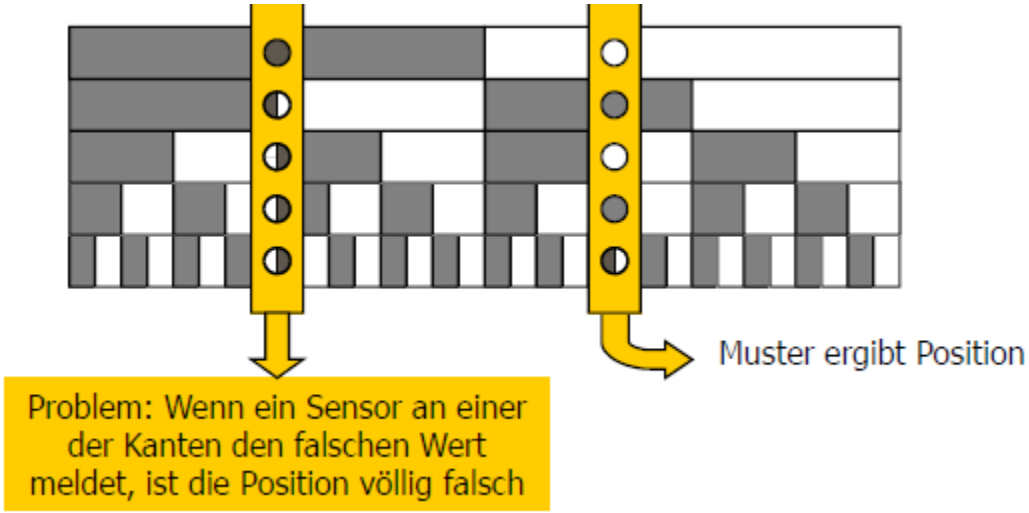
Schnellere Zähler / Addierer

- 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



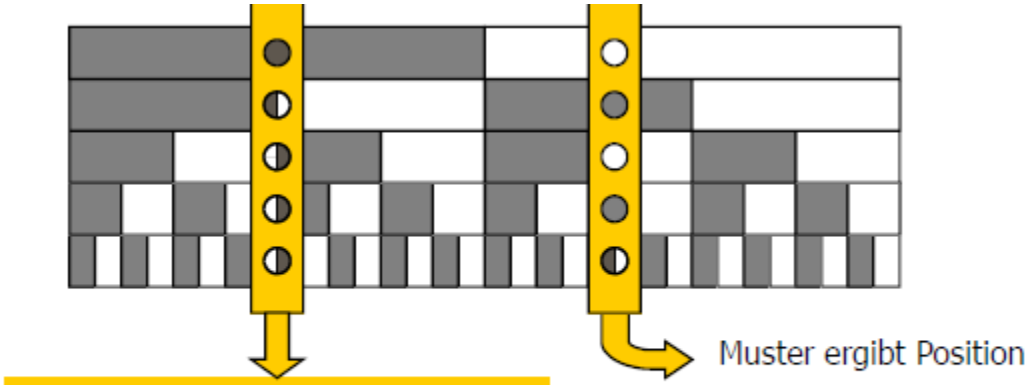
Gray Zähler: Implementierung

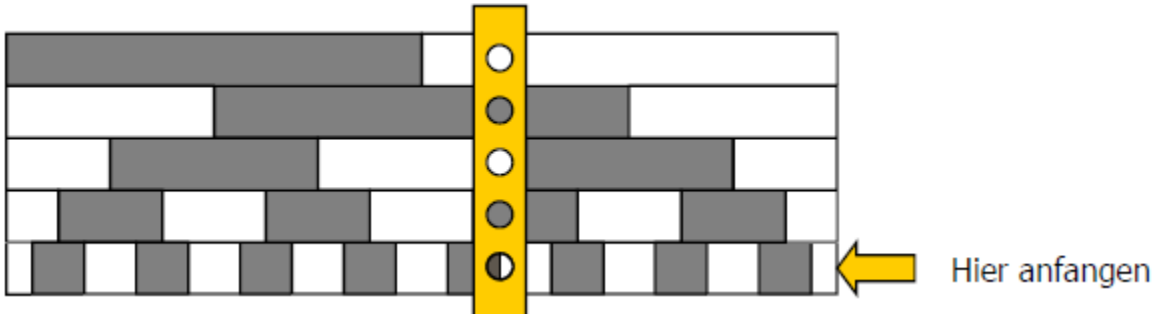
- **Gray Zähler**
- Betrachten wir z.B. einen linearen Maßstab zur Positionsmessung mit binärer Kodierung und Photosensor:



Gray Zähler: Implementierung

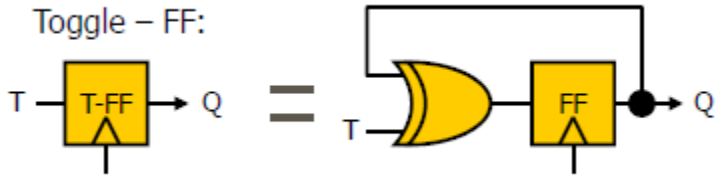
- Lösung: An jeder Kante darf sich nur ein Bit ändern. z.B.: Gray Code: Ändere das niedrigste mögliche Bit

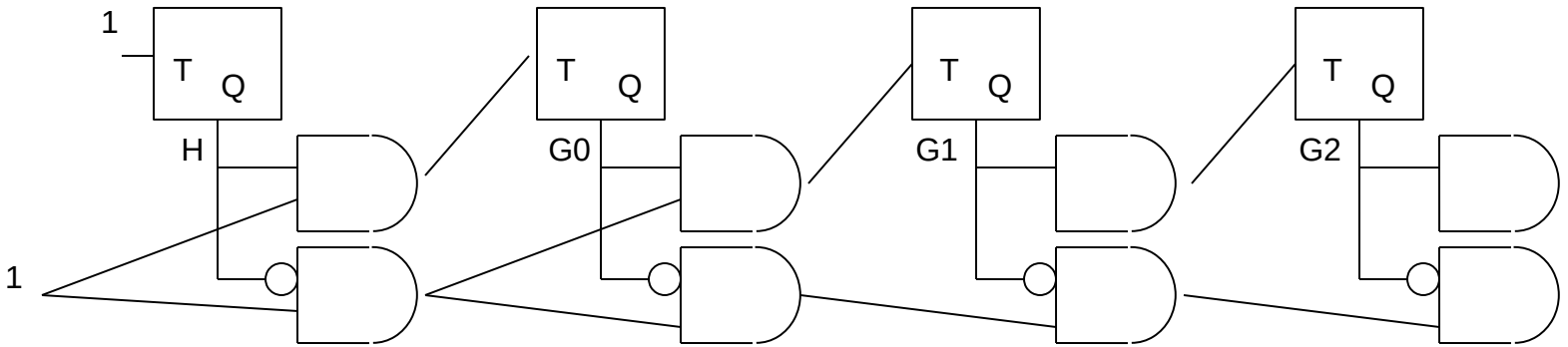




Gray Zähler: Implementierung

- Grey

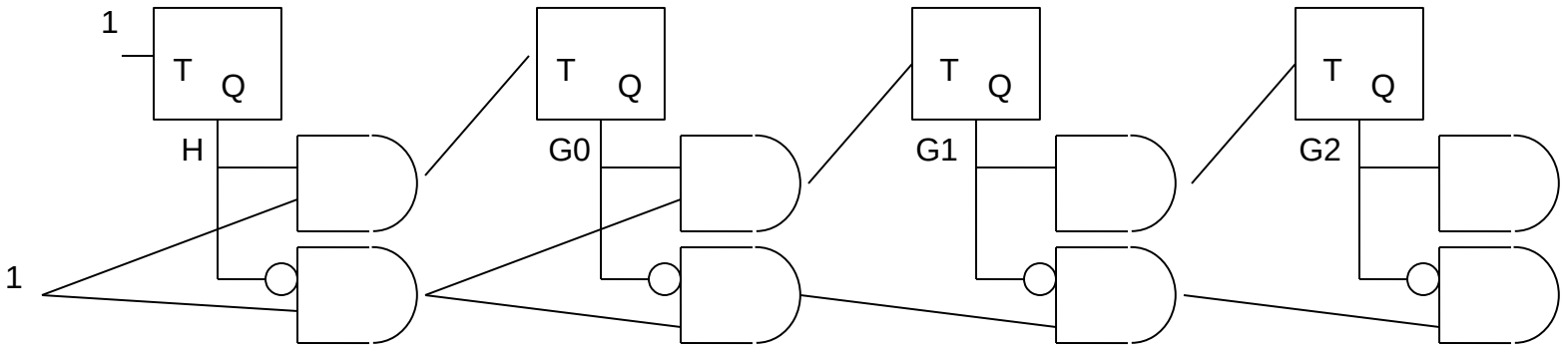




Grey-Zähler

- Grey

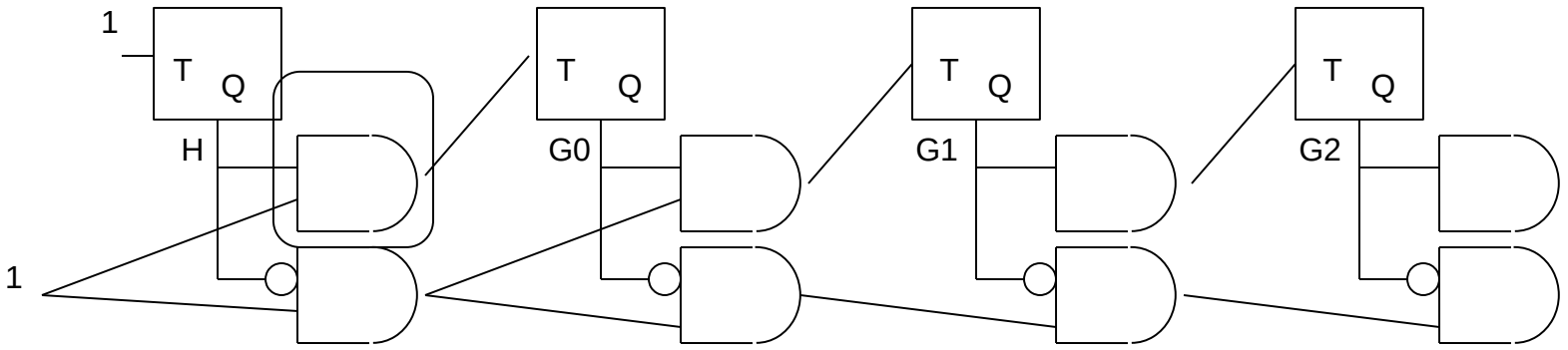
	G2	G1	G0	H
0	0	0	0	1
1	0	0	1	0
2	0	1	1	1
3	0	1	0	0
4	1	1	0	1
5	1	1	1	0
6	1	0	1	1
7	1	0	0	0



Grey-Zähler

- Grey

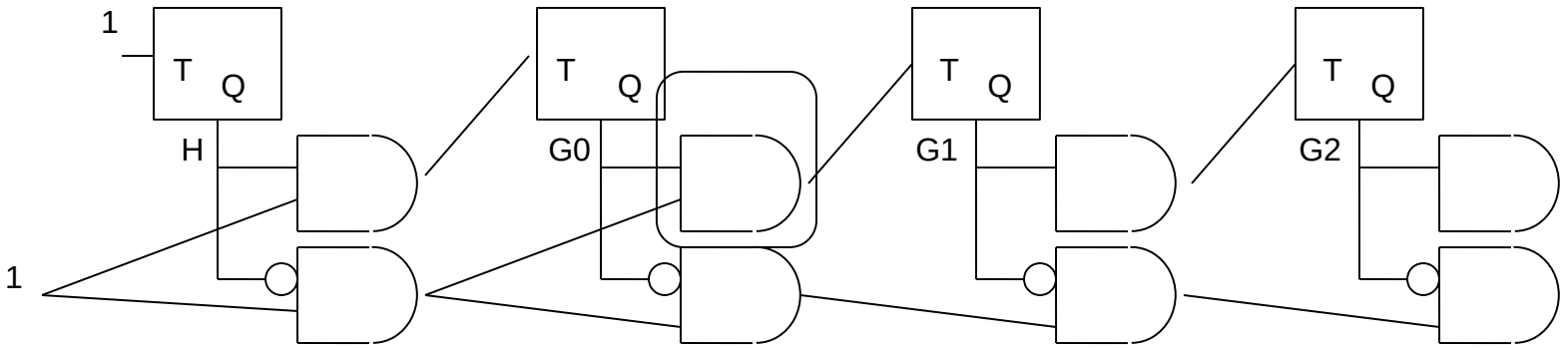
	G2	G1	G0	H
0	0	0	0	1
1	0	0	**1**	0
2	0	1	1	1
3	0	1	**0**	0
4	1	1	0	1
5	1	1	**1**	0
6	1	0	1	1
7	1	0	**0**	0



Grey-Zähler

- Grey

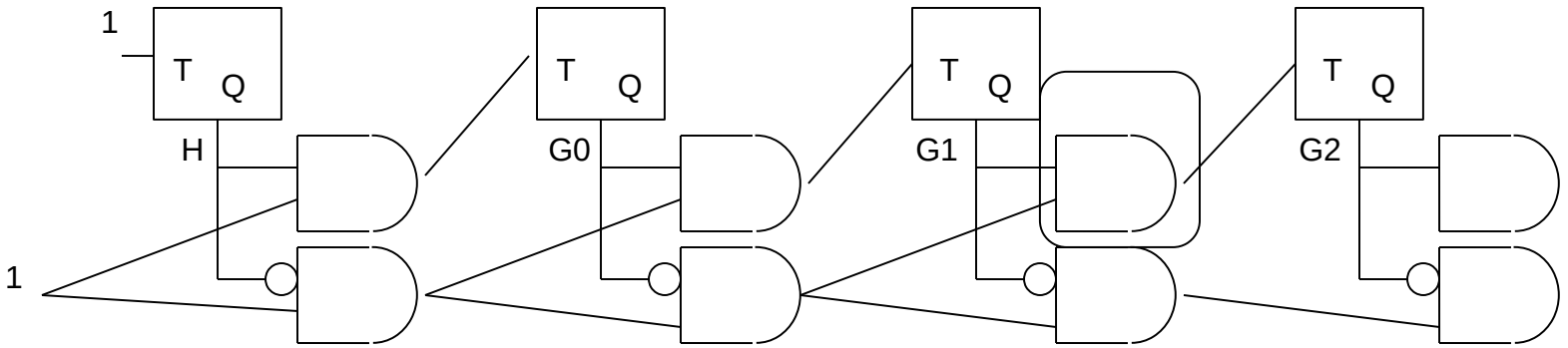
	G2	G1	G0	H
0	0	0	0	1
1	0	0	1	0
2	0	**1**	1	1
3	0	1	0	0
4	1	1	0	1
5	1	1	1	0
6	1	**0**	1	1
7	1	0	0	0



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