Vorlesung 10

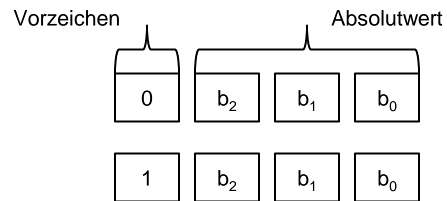
Themen

- Addierer
- Schieberegister
- Zähler
- Zusätzliche Folien
- Scrambler
- Pipelining

Addierer

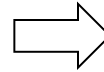
Dezimalsystem: 5, -5

Binärsystem



Addition von Zahlen mit Vorzeichen

- Repräsentation von Zahlen - einfach
- Implementierung der arithmetischen Operationen - kompliziert



Dezimal	b_3	b_2	b_1	b_0
7	0	1	1	1
6	0	1	1	0
5	0	1	0	1
4	0	1	0	0
3	0	0	1	1
2	0	0	1	0
1	0	0	0	1
0	0	0	0	0
0	1	0	0	0
-1	1	0	0	1
-2	1	0	1	0
-3	1	0	1	1
-4	1	1	0	0
-5	1	1	0	1
-6	1	1	1	0
-7	1	1	1	1

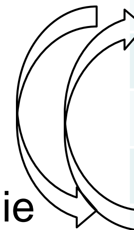
Addition von Zahlen mit Vorzeichen

- Betrachten wir die Subtraktion A - B
- Sie kann als Addition von A und –B verstanden werden.
- B ist positive Zahl. Absolutwert von B ist mit n-1 Bits binär kodiert, das n-te Bit ist 0.
- Wir weisen jeder negativen Zahl -B ein Code B~ zu so dass die Operationen A – B und A + B~ die gleiche Ergebnisse liefern, wenn man Übertrag vernachlässigt und nur n-bits betrachtet
- Das gilt für wenn folgendes erfüllt ist:
- $A - B = A + B\sim - 2^n$ (1)
- Subtraktion von 2^n ändert eine n-bit Zahl nicht.
- Aus (1) folgt
- $B\sim = 2^n - B$
- B~ ist Zweierkomplement von B

Addition von Zahlen mit Vorzeichen

- Wie kann man ~B berechnen?
- B~ = 2^n - 1 – B + 1
- 2^n – 1 = 111…1
- !B = 2^n - 1 – B bekommt man wenn man alle Bits von B negiert
- Deswegen gilt
- B~ = !B + 1
- -B ⇔ !B + 1 (2)
- Wenn die negativen Zahlen als Zweierkomplement dargestellt werden, sieht die Tabelle der 4 bit Zahlen (n=4) wie links aus
- Die Formel (2) gilt auch für negative B
- Die Zahl im Dezimalsystem kann mit folgender Formel gerechnet werden:
- $N = -2^{n-1} b_{n-1} + 2^{n-2} b_{n-2} + b_0$

!B + 1



Dezi mal	b_3	b_2	b_1	b_0
7	0	1	1	1
6	0	1	1	0
5	0	1	0	1
4	0	1	0	0
3	0	0	1	1
2	0	0	1	0
1	0	0	0	1
0	0	0	0	0
-1	1	1	1	1
-2	1	1	1	0
-3	1	1	0	1
-4	1	1	0	0
-5	1	0	1	1
-6	1	0	1	0
-7	1	0	0	1
-8	1	0	0	0

Beachten wir, dass die Menge von negativen Zahlen um 1 größer ist

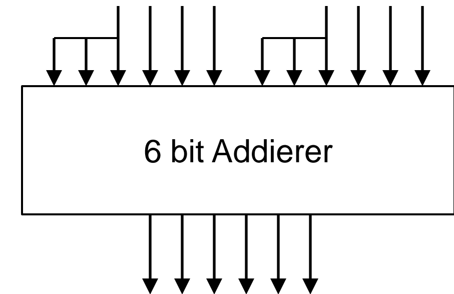
Addition von Zahlen mit Vorzeichen

- Wenn wir -B durch Zweierkomplement ersetzten können wir die Subtraktion A – B als Addition A + ~B = A + !B + 1 durchführen

Dezi mal	b_3	b_2	b_1	b_0
7	0	1	1	1
6	0	1	1	0
5	0	1	0	1
4	0	1	0	0
3	0	0	1	1
2	0	0	1	0
1	0	0	0	1
0	0	0	0	0
-1	1	1	1	1
-2	1	1	1	0
-3	1	1	0	1
-4	1	1	0	0
-5	1	0	1	1
-6	1	0	1	0
-7	1	0	0	1
-8	1	0	0	0

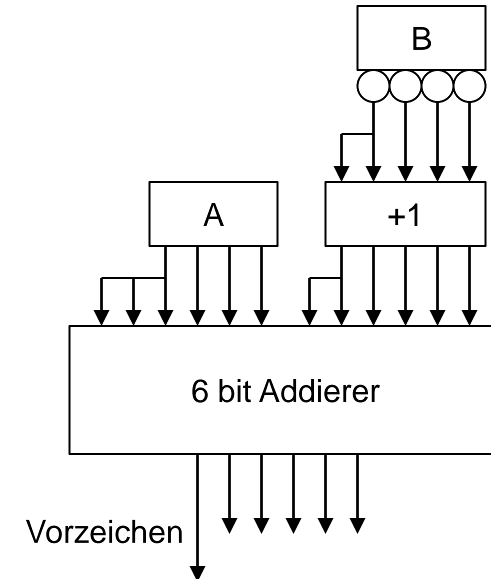
Addition von Zahlen mit Vorzeichen

- Beachten wir dass das Ergebnis von Addition von zwei 4 bit Zahlen nicht immer mit vier Bits dargestellt werden kann. Z.b., im extremsten Fall, für Addition -8 + -8 = -16 brauchen wir 6 Bits. 16dec = 110000
- Wenn wir 4 bit Addieren verwenden, werden nur die Ergebnisse im Bereich -8 bis 7 richtig sein
- Um alle 4 Bit Zahlen mit Vorzeichen richtig zu addieren, brauchen wir 6 Bit Addierer und die Summanden müssen auf 6 Bits erweitert werden, man nennt es Vorzeichenerweiterung.
- Im Fall von positiven Zahlen werden die zu erweiternden Bits mit Nullen und in Fall von negativen Zahlen mit Eisen gefüllt.



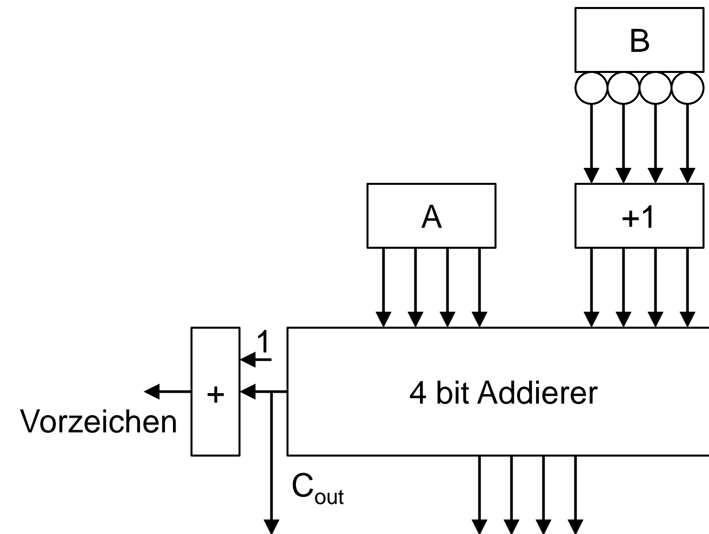
Vergleich von Zahlen mit Vorzeichen

- Auch einen Komparator (>, =, <) von Zahlen A und B kann man mit Addierer realisieren. Die Idee ist B von A abzuziehen und das Vorzeichen zu prüfen.
- Wenn Vorzeichen - ist, bzw. das hochwertigste Bit 1 ist, ist B > A. Ansonsten wenn Ergebnis 000000 ist, gilt A = B ansonsten A > B.



Vergleich von positiven Zahlen

- Wenn wir 4 Bit positive Zahlen vergleichen ist es ausreichend 4 Bit Addierer zu verwenden. In dem Fall fehlen die Hochwertigsten Bits
- Das Vorzeichen vom Ergebnis ist negativ wenn der Addierer keinen Übertrag C_{out} gibt



- Wir können die Addition, Subtraktion und Vergleich von positiven oder Zahlen mit Vorzeichen mit Addierer und mit EXOR realisieren können.
- Die Multiplikation und Division können als Wiederholung von Additionen oder Subtraktionen durchgeführt werden
- Die grundlegende analytische Funktionen wie Exp, Log, sin, cos können mit Multiplikationen, Divisionen, Additionen und Subtraktionen realisiert werden (z.B. Taylorreihe, CORDIC) und alle anderen Funktionen, mithilfe von diesen grundlegenden Funktionen.
- Ein Addierer ist ausreichend, um alle Berechnungen durchzuführen.
- Einige Links
- https://www.allaboutcircuits.com/technical-articles/an-introduction-to-the-cordic-algorithm/
- Calculate exp() and log() Without Multiplications (quinapalus.com)

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

		3	9	
+		6	9	
	1	1		← Übertrag
	1	0	8	

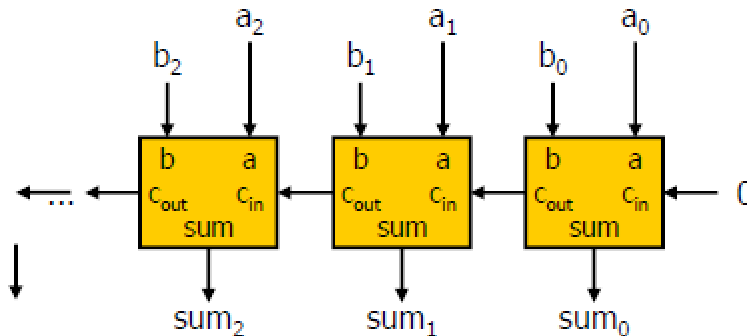
		1	0	0	1	1	1	
+	1	0	0	0	1	0	1	
	0	0	0	1	1	1		← Übertrag
	1	1	0	1	1	0	0	
	x64	x32		x8	x4			

Abbildungen aus den Vorlesungen VLSI Design, Peter Fischer, LS SuS

SuS - Lehre (uni-heidelberg.de)

Subtraktion – Wikipedia

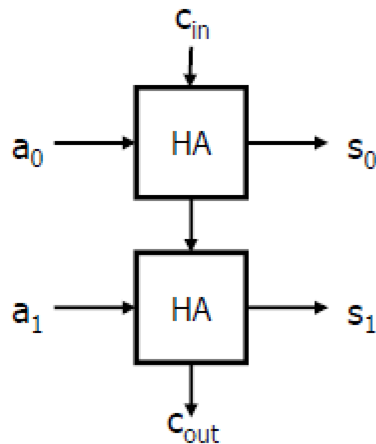
- Die Addition erfolgt stellenweise wie bei Dezimalzahlen mit einem **Übertrag (carry)**:
- In jeder Stufe werden also aus den **3 Eingängen** a, b, c_{in} die **Ausgänge** sum und c_{out} erzeugt.
- Man nennt diesen Schaltungsblock den full adder (FA) (Volladdierer)



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

SuS - Lehre (uni-heidelberg.de)

- 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)



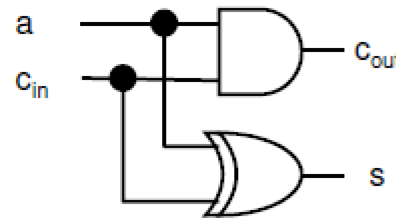
c_{in}	a	s	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

"." ist UND — "+" ist ODER

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

$s = a \oplus c_{in}$

Addition

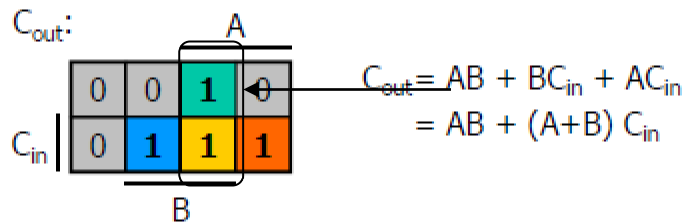


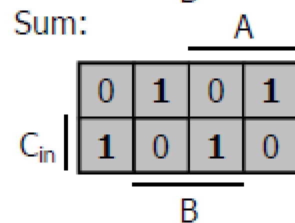
SuS - Lehre (uni-heidelberg.de)

Addierer

- …

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

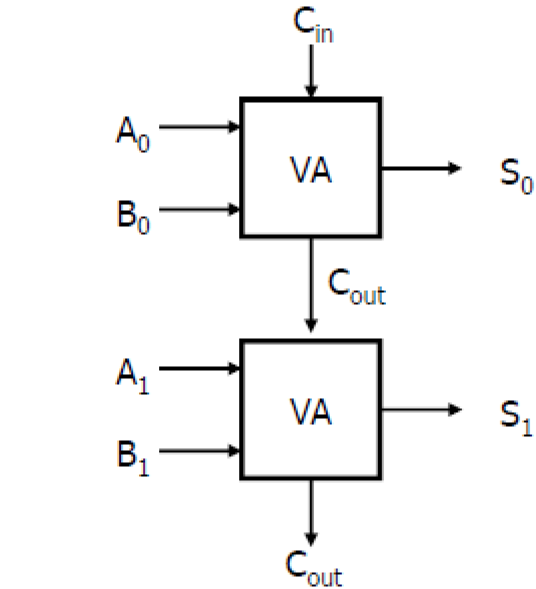




$S = A \oplus B \oplus C_{in}$

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

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

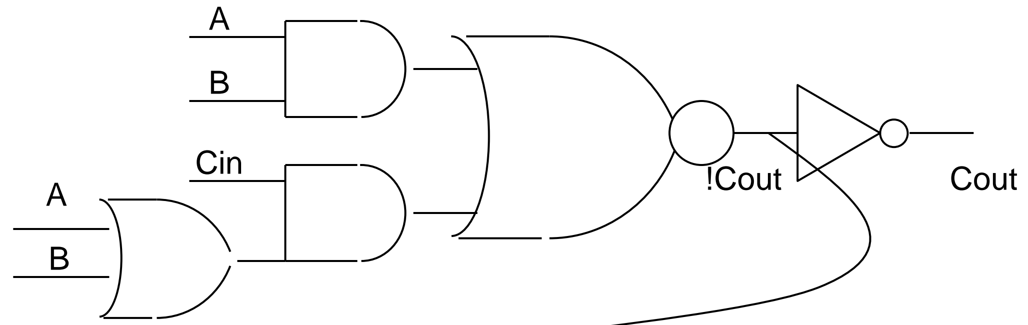


Der Carry-Pfad muß optimiert werden, da das Carry durch alle N Bit 'rippeln' muß

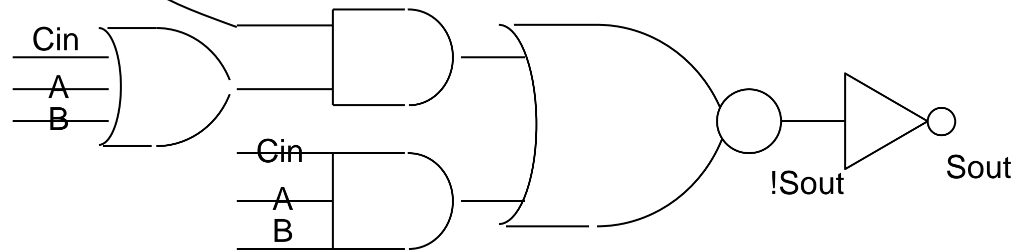
Karnaugh Karten

- …

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

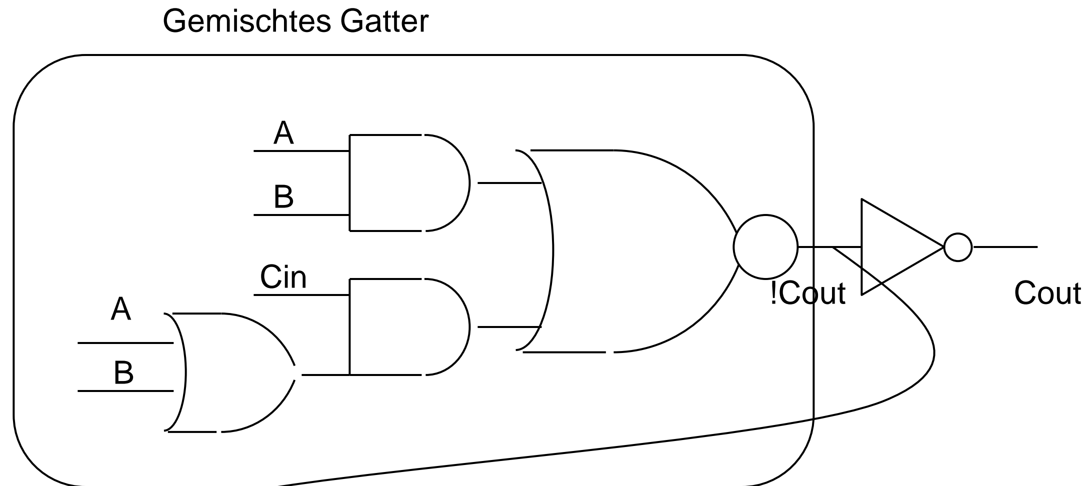


$S = ABC_{in} + (A + B + C_{in}) \cdot\, !C_{out}$:

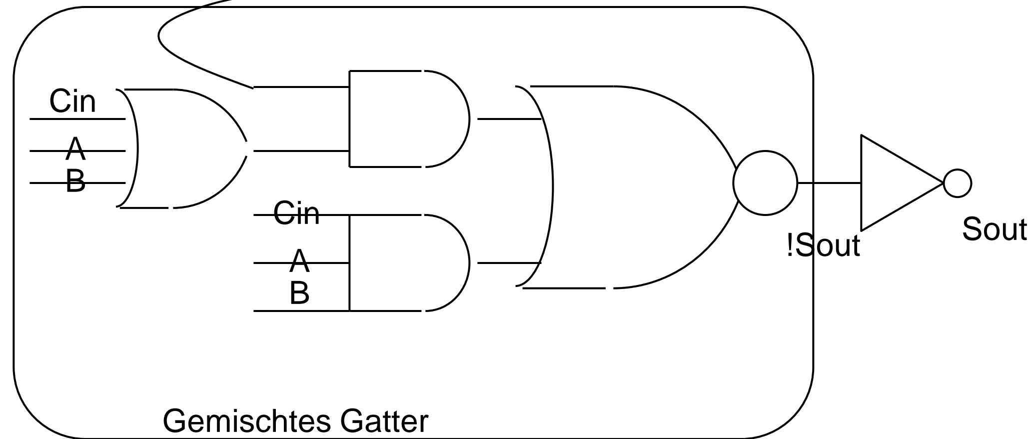


- …

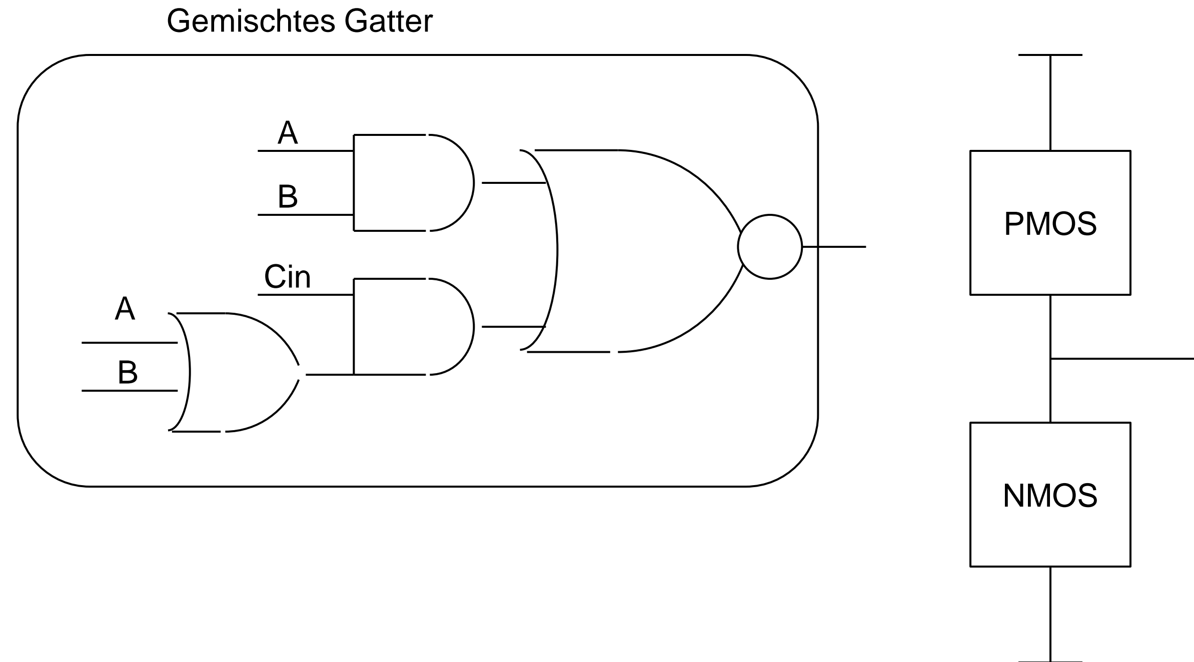
$C_{out} = AB + (A+B)\,C_{in}$:



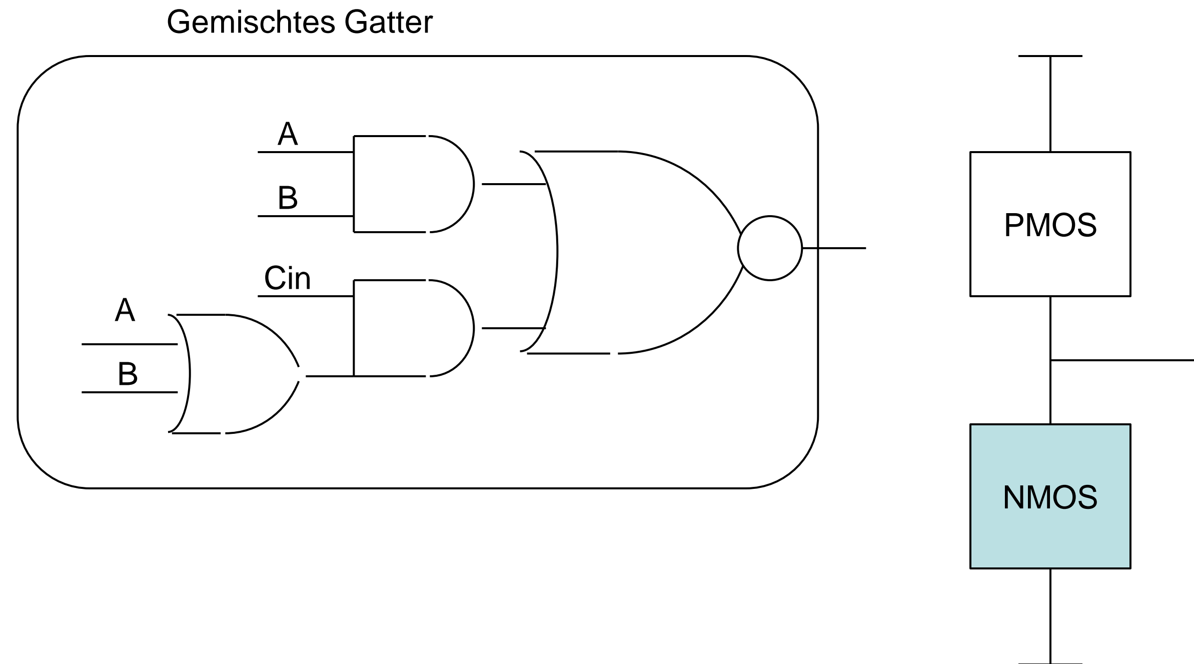
$S = ABC_{in} + (A + B + C_{in}) \cdot !C_{out}$:



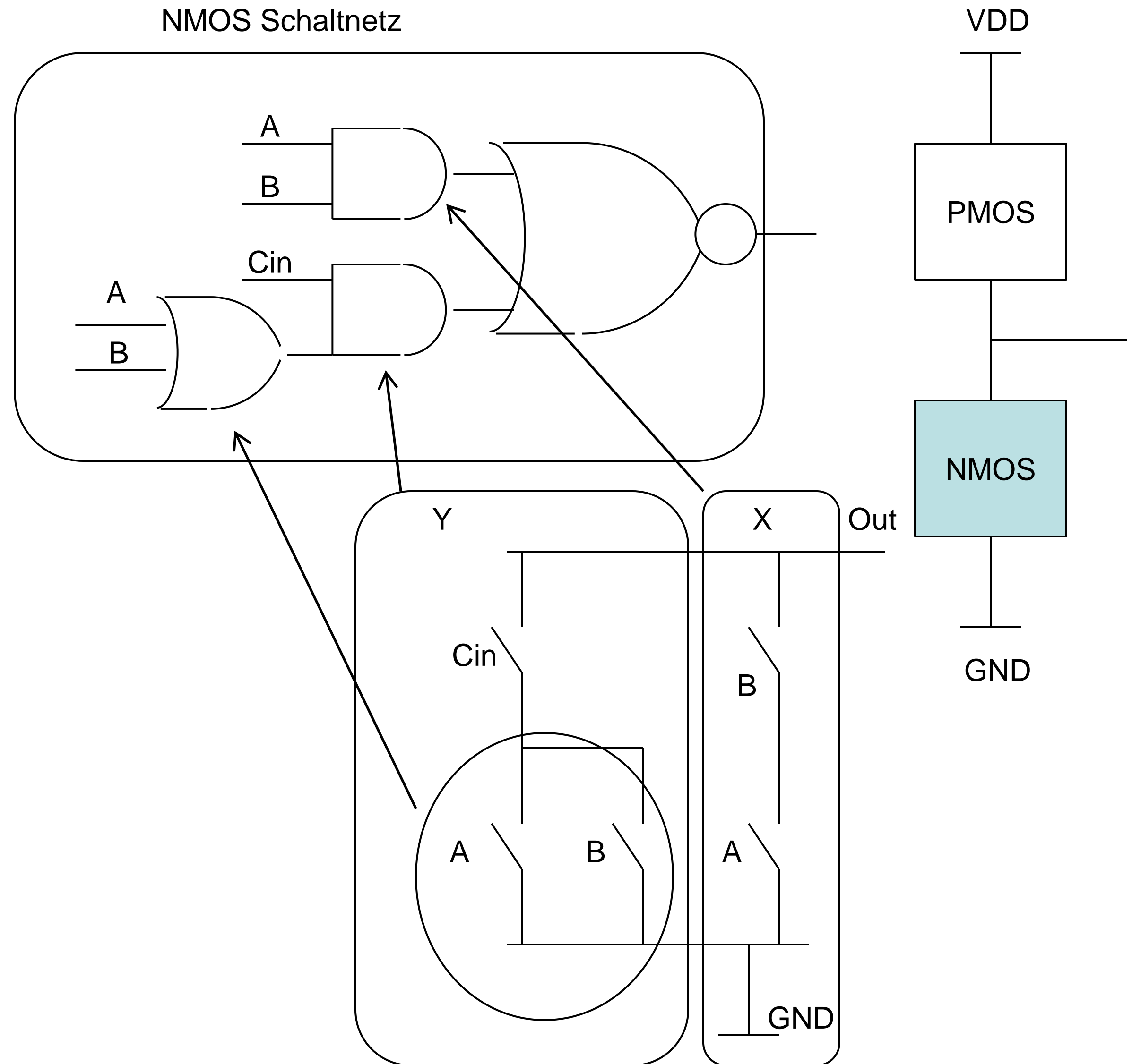
- Versuchen wir ein gemischtes Gatter für $!C_{out}$ mit Transistoren zu implementieren



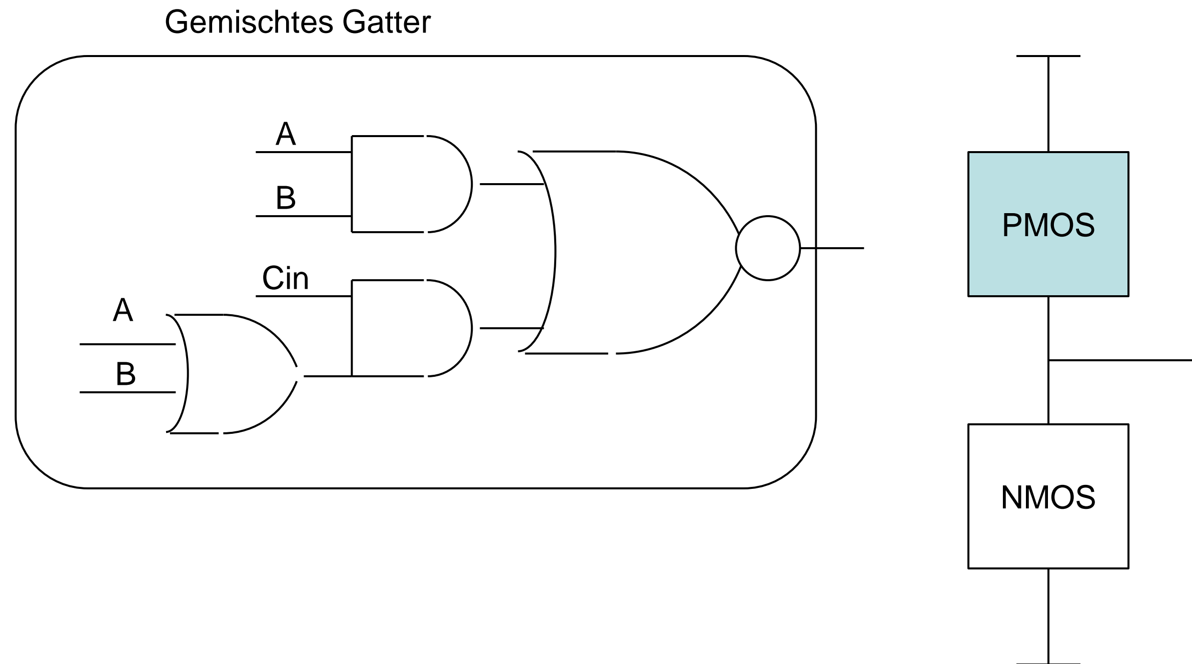
- NMOS Teil



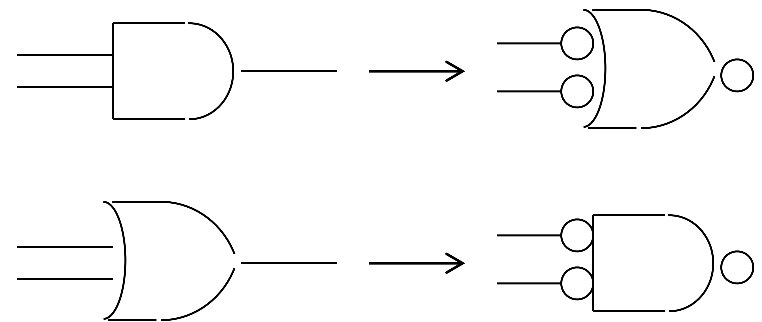
- …


NMOS Schaltnetz
A
B
Cin
A
B
VDD
PMOS
NMOS
GND
Y
X
Out
Cin
B
A
B
A
GND

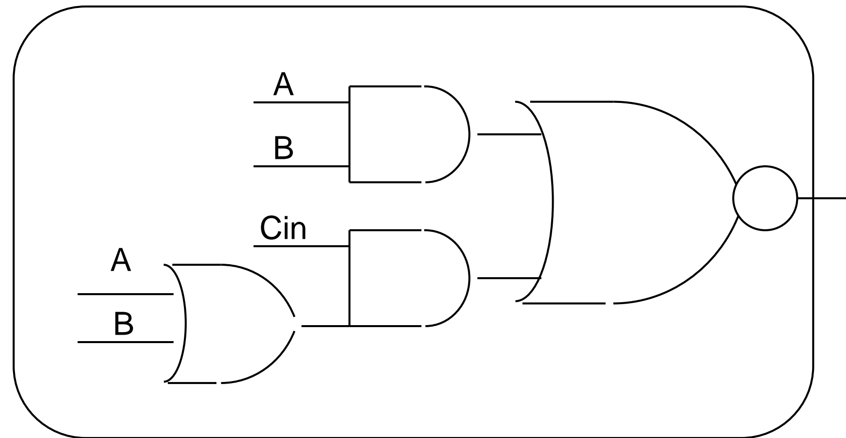
- PMOS Teil



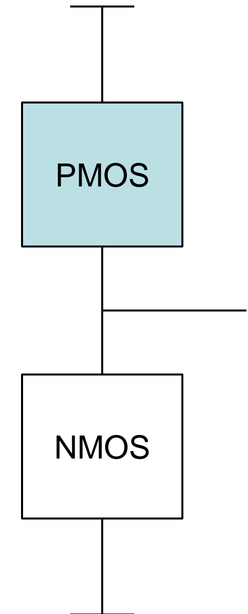
- Wir verwenden die De-Morgansche Regel



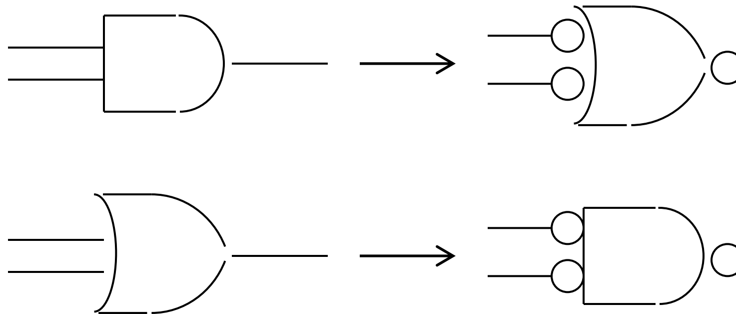
- …



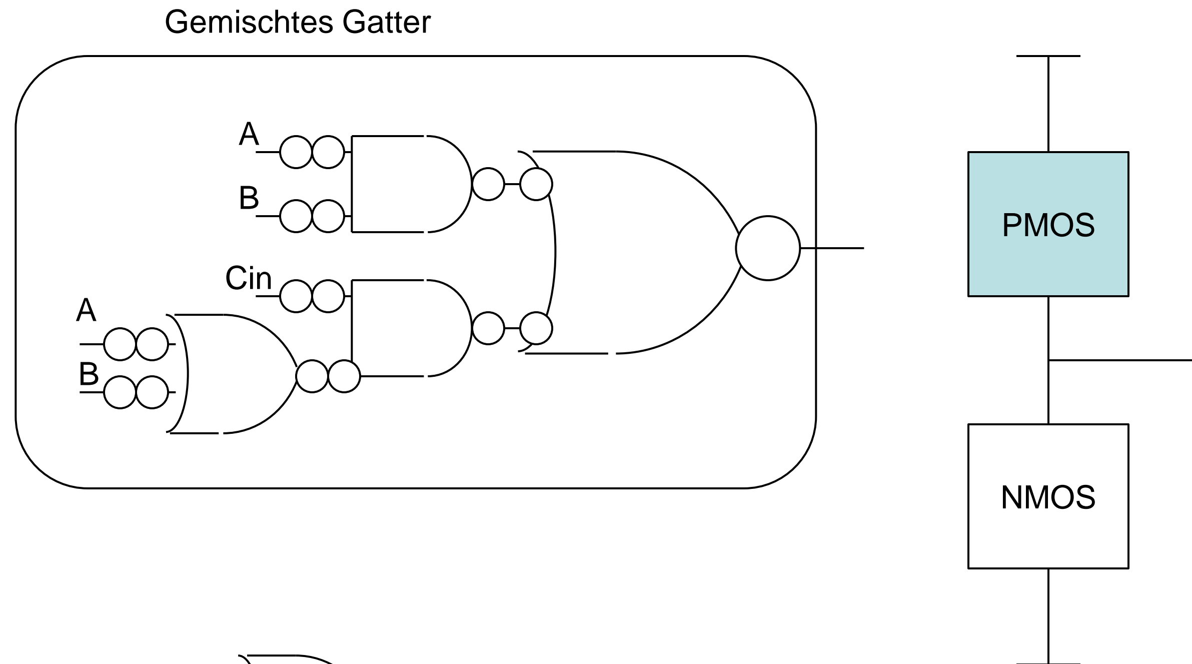
Gemischtes Gatter
A
B
Cin
A
B

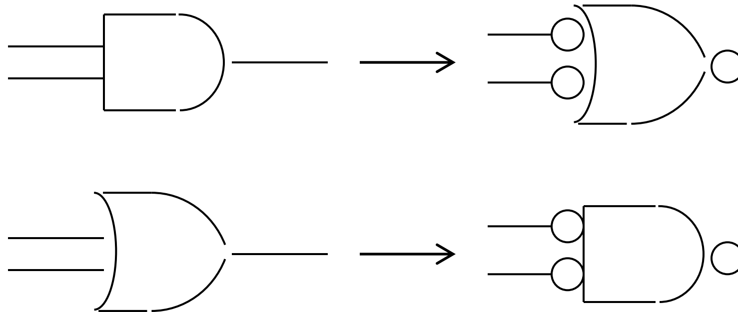


PMOS
NMOS

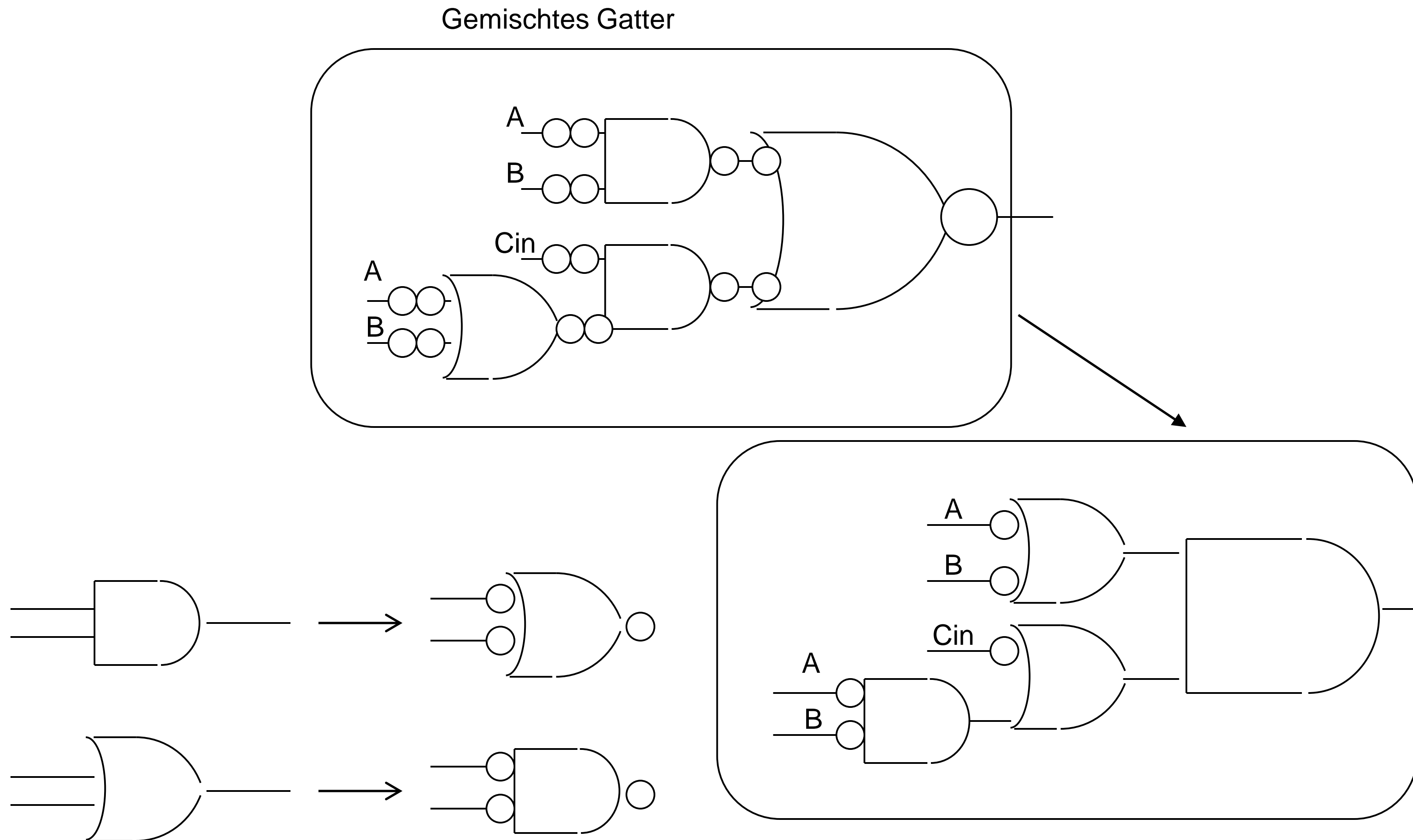


- Erster Schritt: doppelte Negation

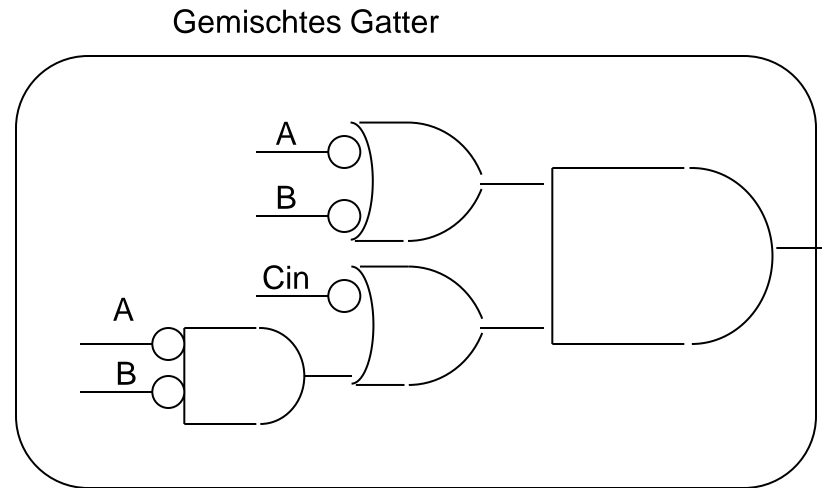


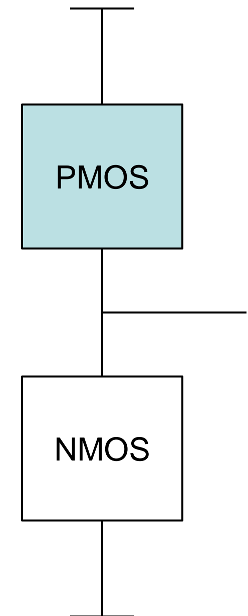


- Zweiter Schritt: De-Morgansche Regel

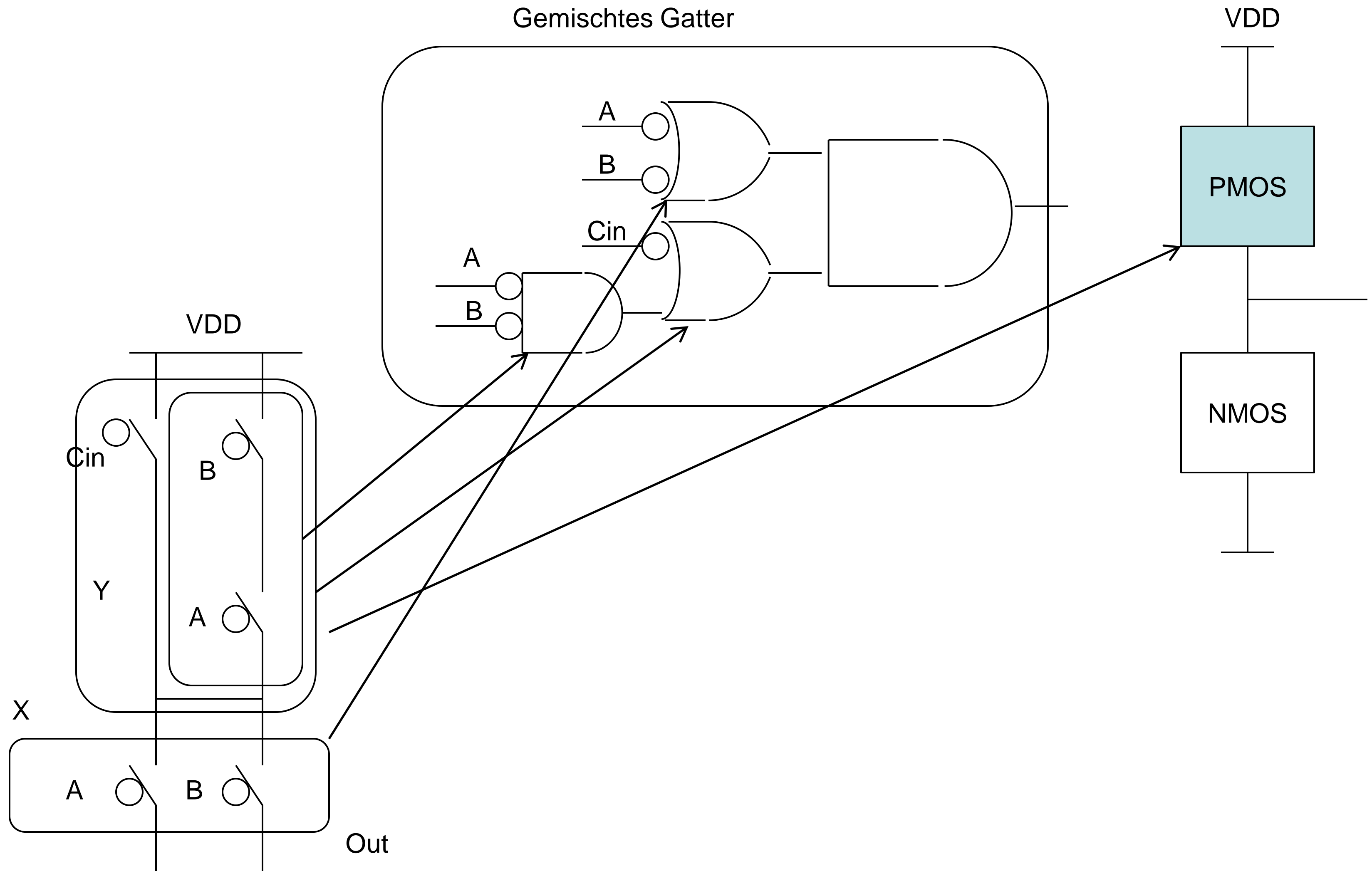


- …

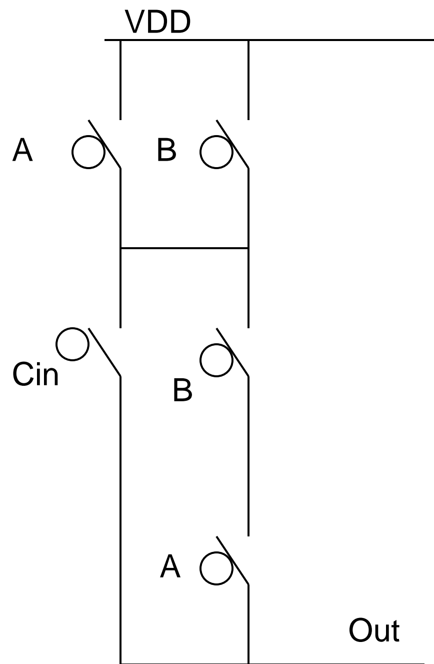

Gemischtes Gatter
A
B
Cin
A
B

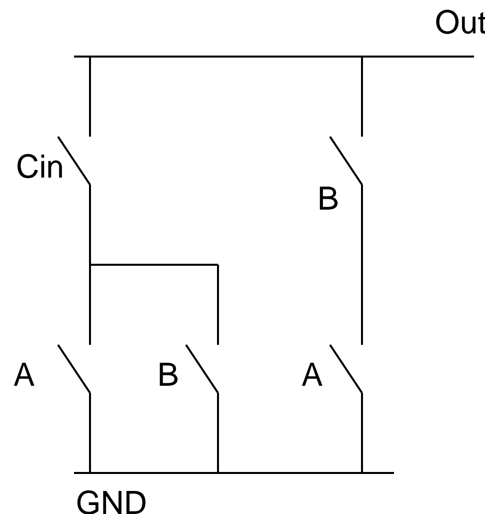

PMOS
NMOS

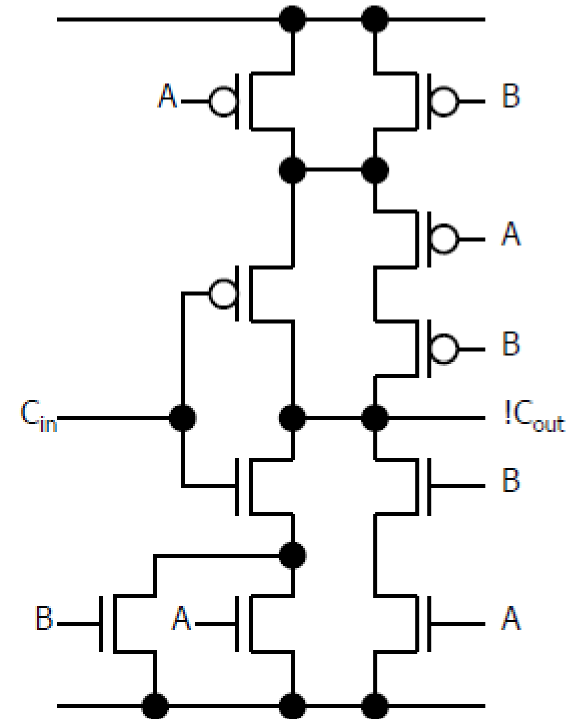
- …


Gemischtes Gatter
A
B
Cin
A
B
VDD
PMOS
NMOS
VDD
Cin
B
Y
A
X
A
B
Out

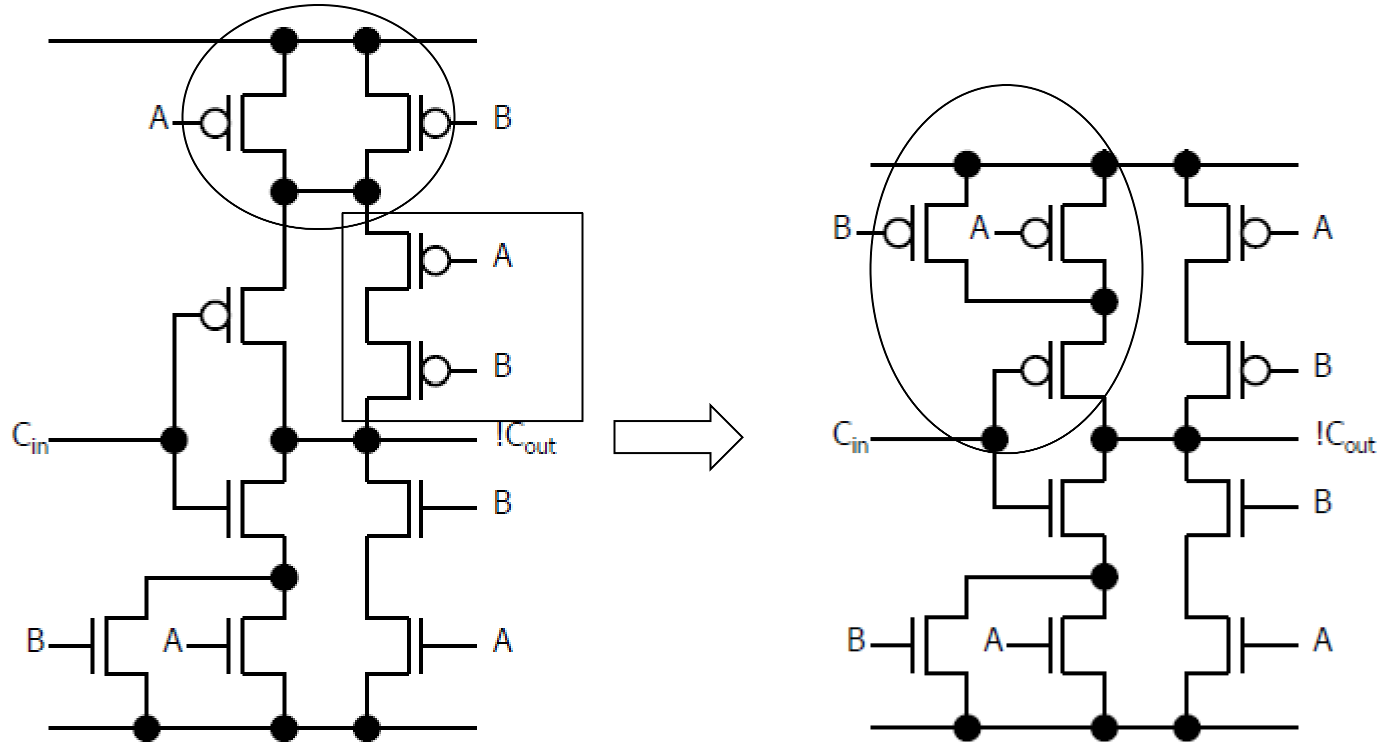
- Das Carry wird durch C_{out} = AB + (A+B)C_{in} gegeben.
- Diese Funktion kann mit dem gemischten Gatter für !C_{out} und einem Inverter implementiert werden
- Problem: 3 PMOS übereinander ('Stack height' = 3)



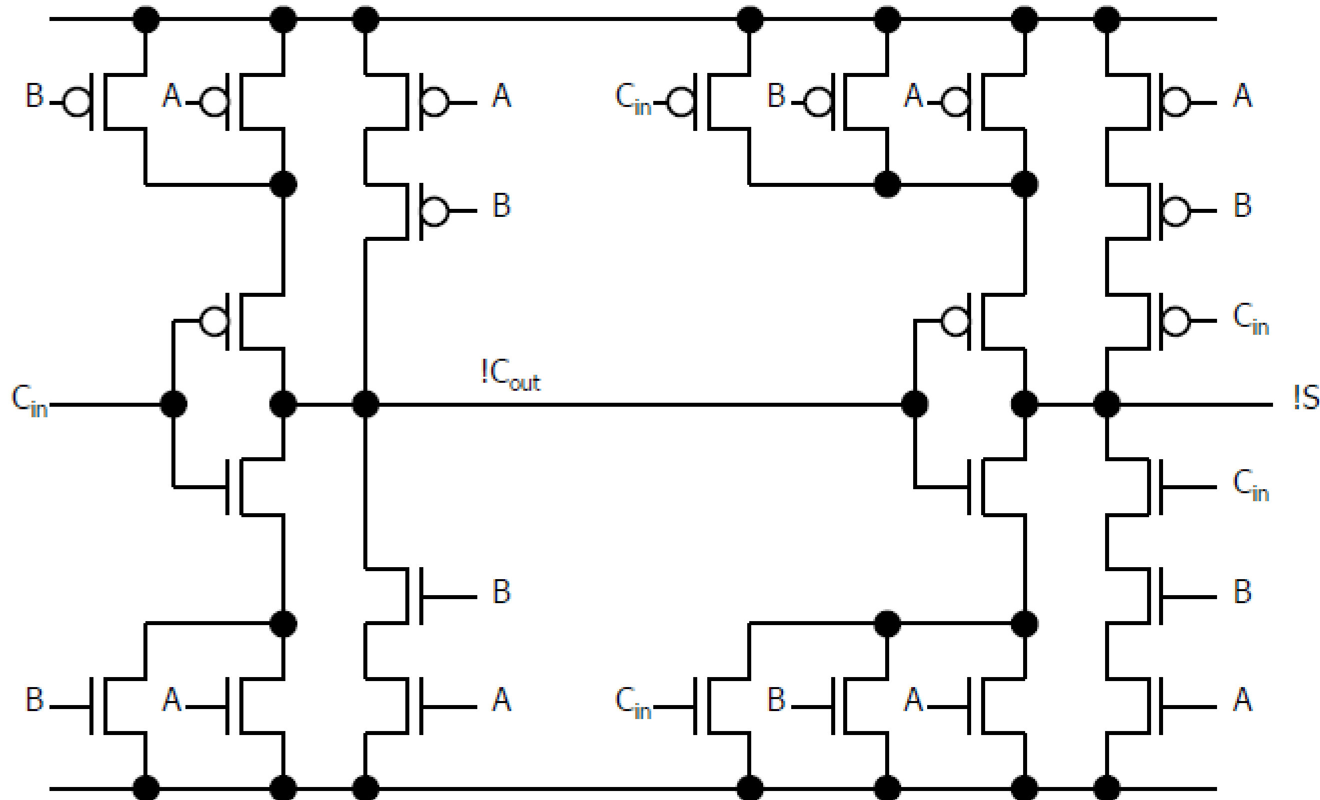




- Der PMOS Zweig kann umgeformt werden
- Idee: Wenn !A UND !B leitet (Schaltung im Rechteck) leitet !A ODER !B auch (Schaltung im Kreis)
- => Man kann !A UND !B an VDD anschliessen

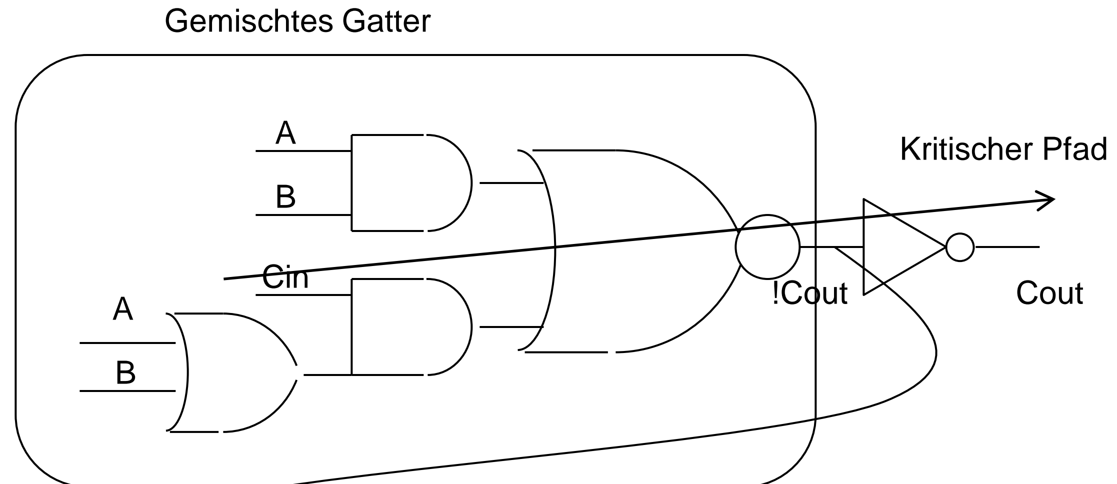


- -> Optimierter Volladierer

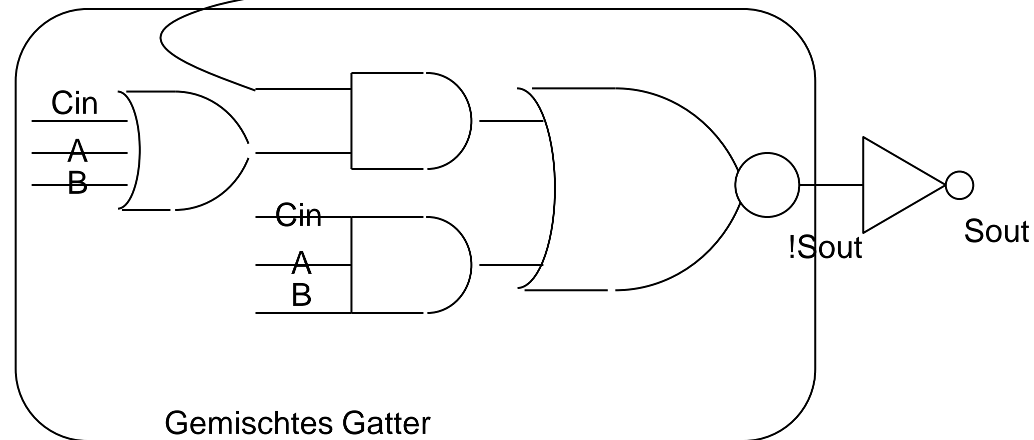


- Carry ist ein kritischer Pfad, da das Signal durch alle Bits läuft

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



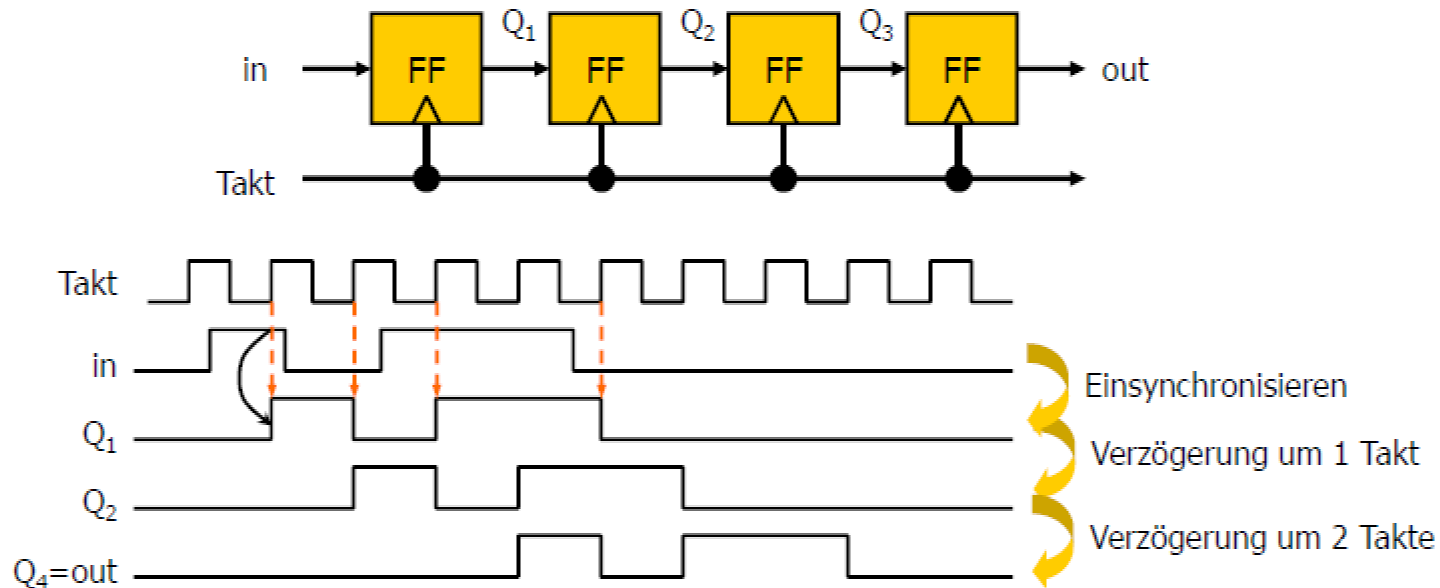
$S = ABC_{in} + (A + B + C_{in}) \cdot !C_{out}$:



Schieberegister

Schieberegister

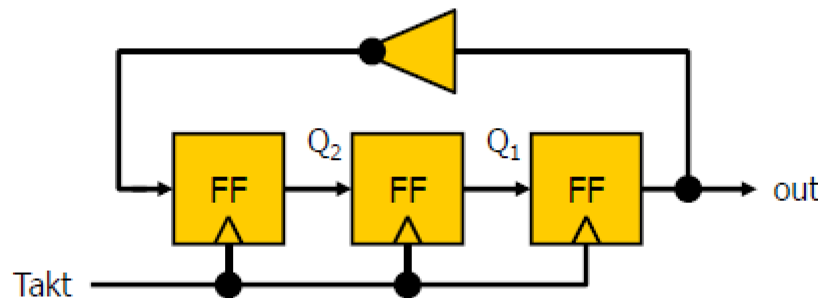
- Schieberegister entstehen durch Hintereinanderschalten von Flipflops (FF).
- Zwischen den Stufen ist keine (wenig) Logik
- **Vorsicht**: Die Hold-Zeit kann leicht verletzt sein. Daher fügt man manchmal Verzögerungen (Buffer) in den Datenpfad ein.
- Anwendungen:
  - Verzögerung von Signalen (z.B. bei Pipelining)
  - Einfache Zustandscodierung
  - Spezielle Zähler (linear feedback shift register - LFSR)

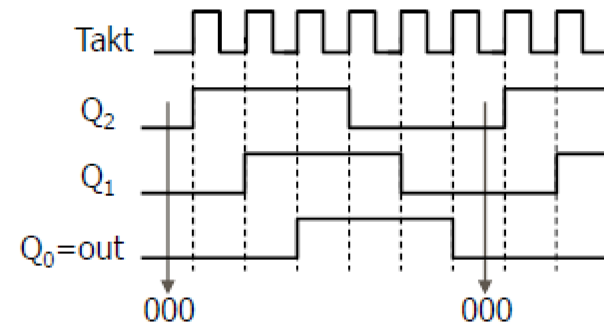


Zähler

Johnson Zähler

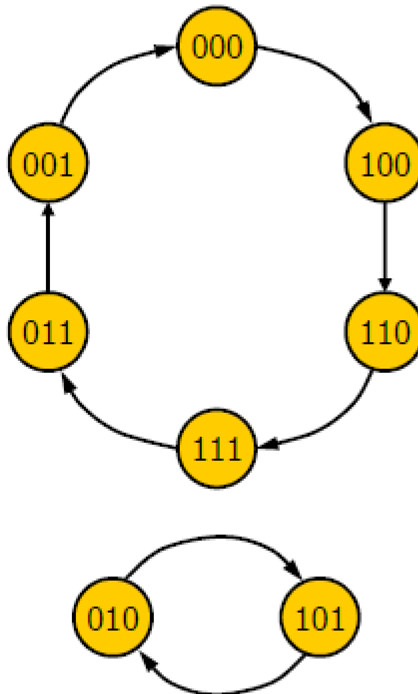
- Beim ‚Johnson Zähler' wird der Ausgang über einen Inverter zum Eingang rückgekoppelt
- Der Zähler hat dadurch 2N Zustände…
- Johnson Zähler ist ein einfaches Beispiel vom linear feedback shift register, (linear rückgekoppeltes Schieberegister)





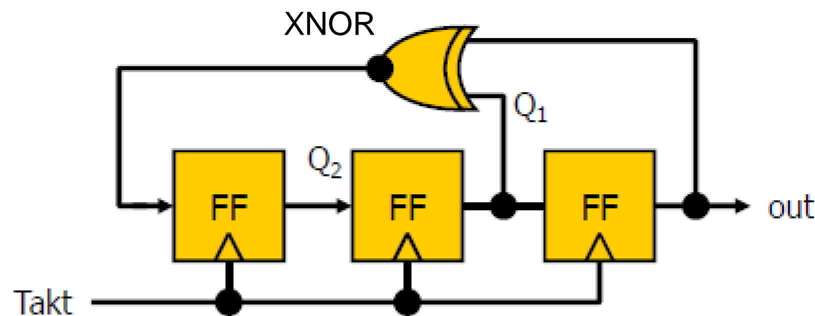
Johnson Zähler: Sprungdiagramm

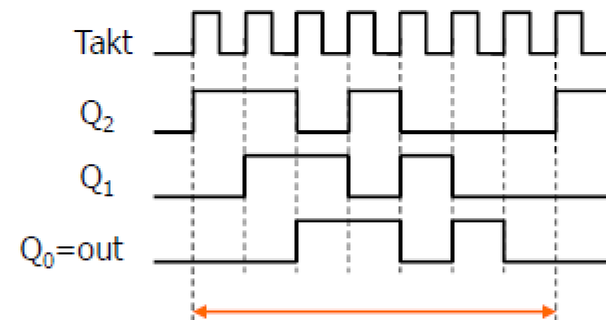
- Bei N=3 gibt es 2^3 mögliche Zustände.
- 6 davon werden vom Johnson Zähler durchlaufen:
- Die verbleibenden beiden Zustände bilden einen eigenen Zyklus.
- Man muss mit einem Reset vermeiden hier zu starten
- Das Zurücksetzen in einen Anfangszustand kann durch sync/async. Reset der FFs erfolgen



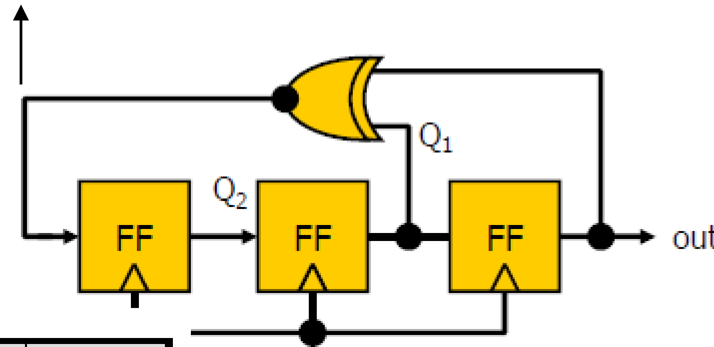
Pseudo-random Zähler

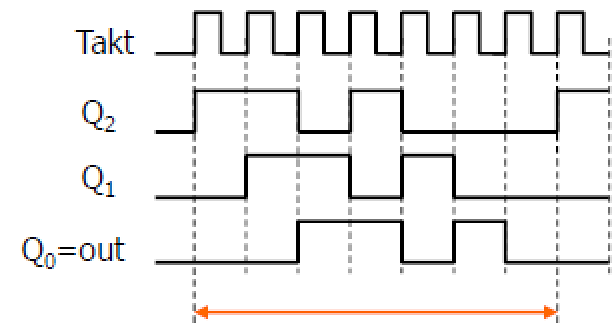
- Ein komplexeres Beispiel vom LFSR:
- Durch Rückkopplung des Ausgangs und eines (oder mehrerer) geeigneten Abgriffs (‚tap‘) kann bei N Flipflops eine Bitsequenz mit der Periode 2N-1 entstehen (‚maximum length‘)
- Die Bitsequenz hat keine erkennbare Struktur und wird daher als Pseudo-Random-Bit-Sequence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





- Einige Eigenschaften:
- In der gesamten Sequenz am XNOR kommt nur genau eine Eins weniger vor als Nullen
- Die Hälfte aller zusammenhängenden Einser/Nullen-Blöcke ist einen Takt lang, ein Viertel ist zwei Takte lang, etc. (bis 1/Sequenzlänge = kein Block länger als Zahl von Bits in SR).
- Beispiel 6 bits, Sequenzlänge 63 – nie mehr als 6 zusammenhängende Nullen/Einsen
- 000000111110111100111010110000101110001101101001000100110010101



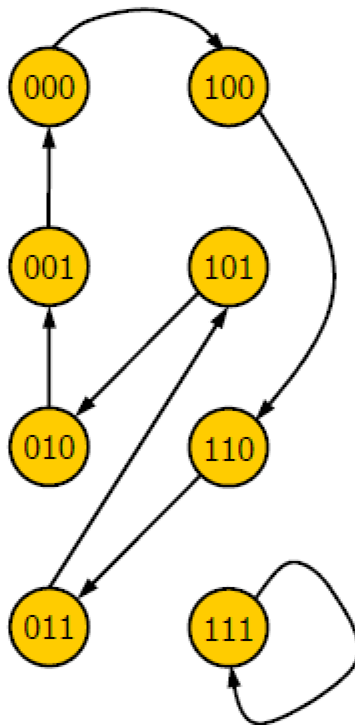


N	Abgriffe	Länge	Maximal ?
3	Q_1	7	ja
4	Q_1	15	ja
5	Q_2	31	ja
15	Q_1	32767	ja
16	Q_{12}, Q_3, Q_1	65535	ja
16	Q_7		96.8%
39	Q_4	5×10^{11}	

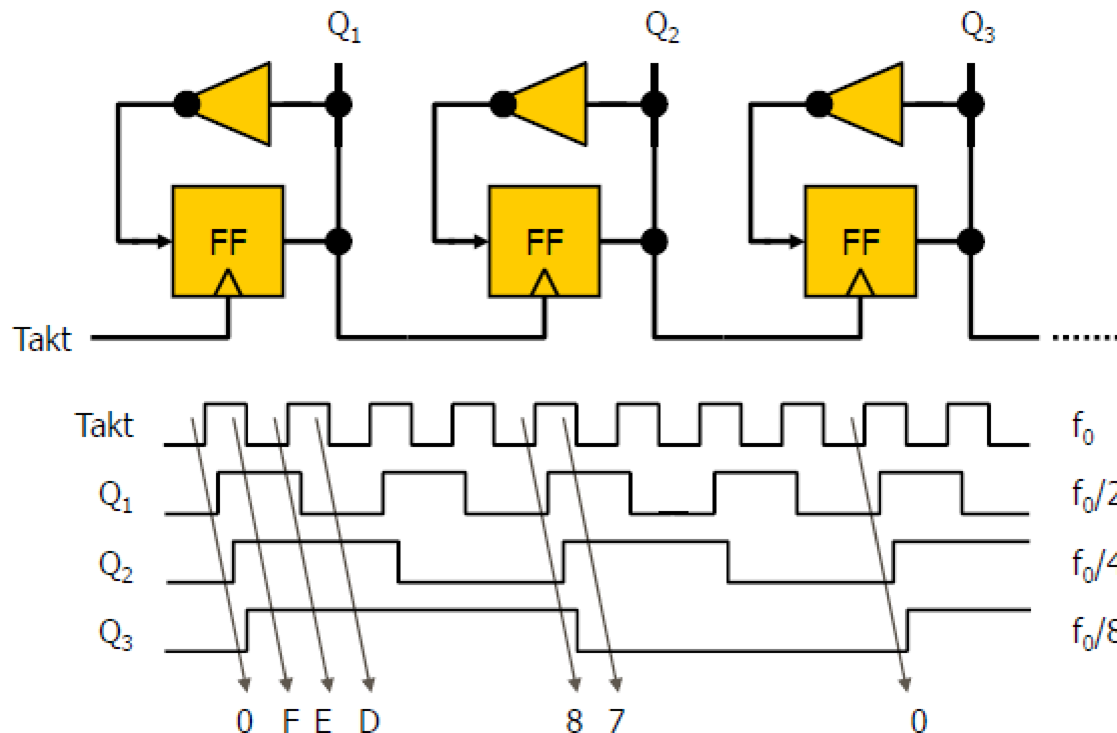
XAPP052 - Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators (7/96) (xilinx.com)

Pseudo-random Zähler

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

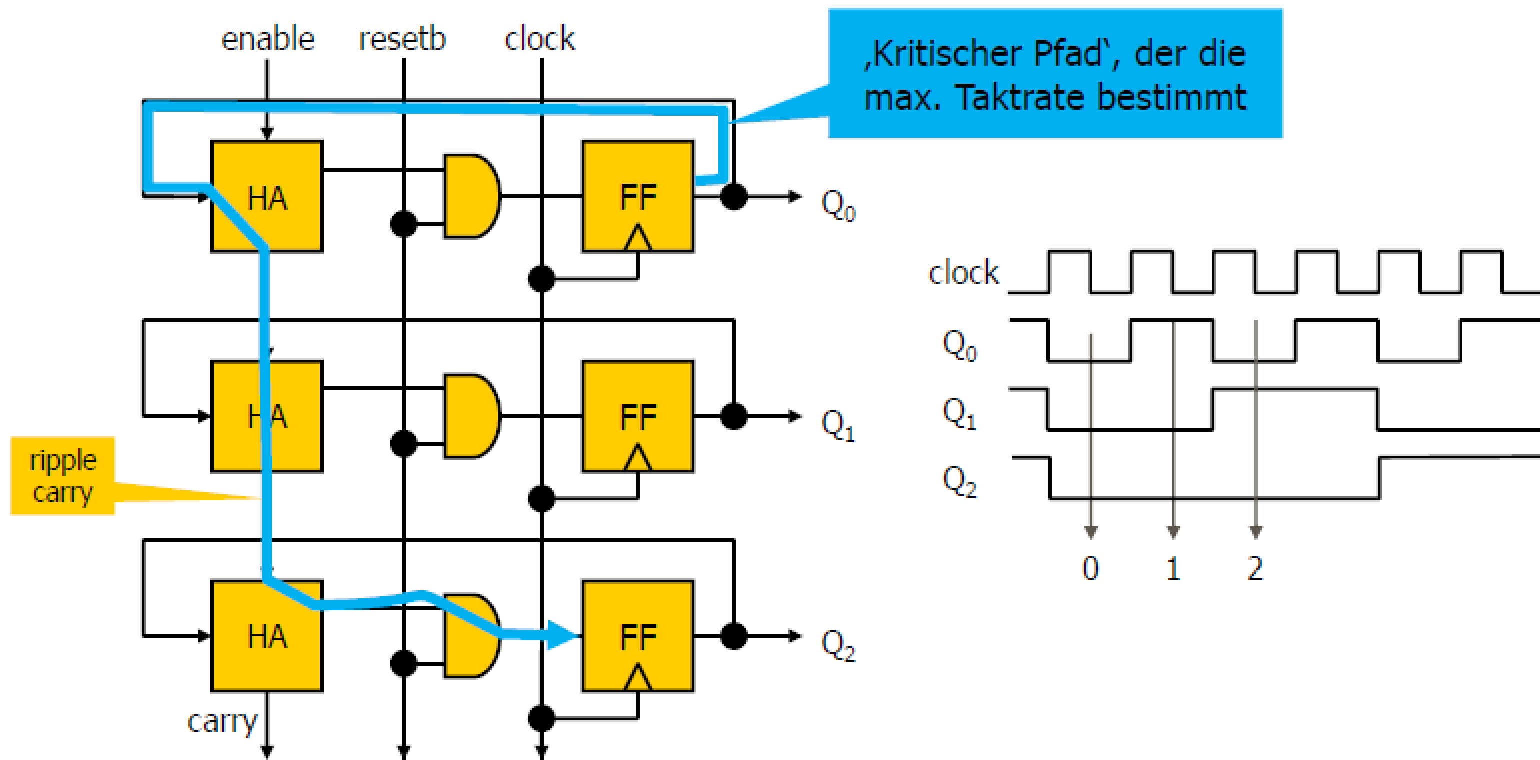


- Rückkopplung von !Q auf D erzeugt Toggle-FFs, die bei jedem Takt den Zustand ändern (0->1->0->...)
- Der Q-Ausgang eines Bits steuert das nächste Bit an (hier Rückwärtszähler):
- Wegen der Verzögerung der einzelnen Stufen sind die Flanken **nicht gleichzeitig** (daher async. Zähler)
- Anwendung: Frequenzteiler



Synchrone Binärzähler

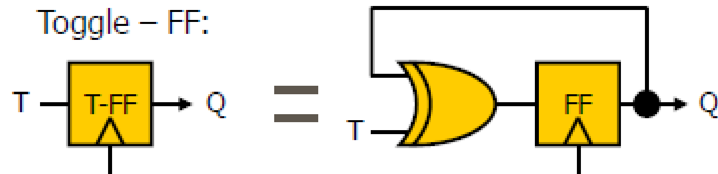
- Alle FFs werden gleichzeitig getaktet
- Die Eingänge werden so beschaltet, daß sich (z.B.) aufsteigend Binärzahlen ergeben
- Implementierung mit Halbaddierern (mit enable und reset)
- Max. Taktfrequenz ist durch die Laufzeit des 'ripple' Carry begrenzt.

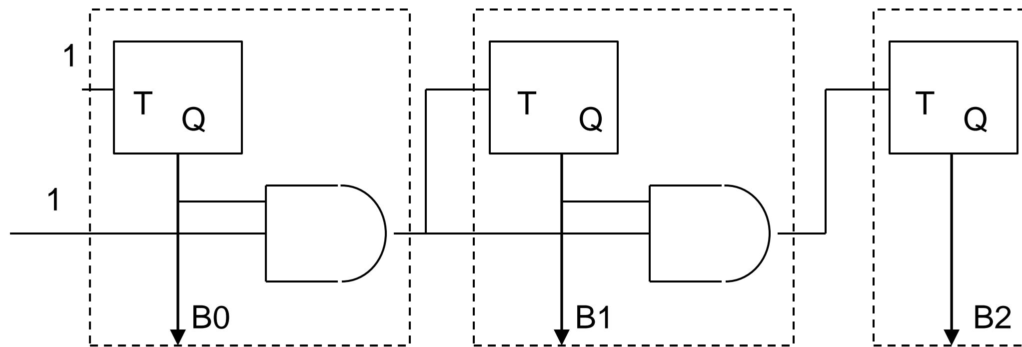


Binärer Zähler mit T-Flipflops

- ...

	B2	B1	B0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

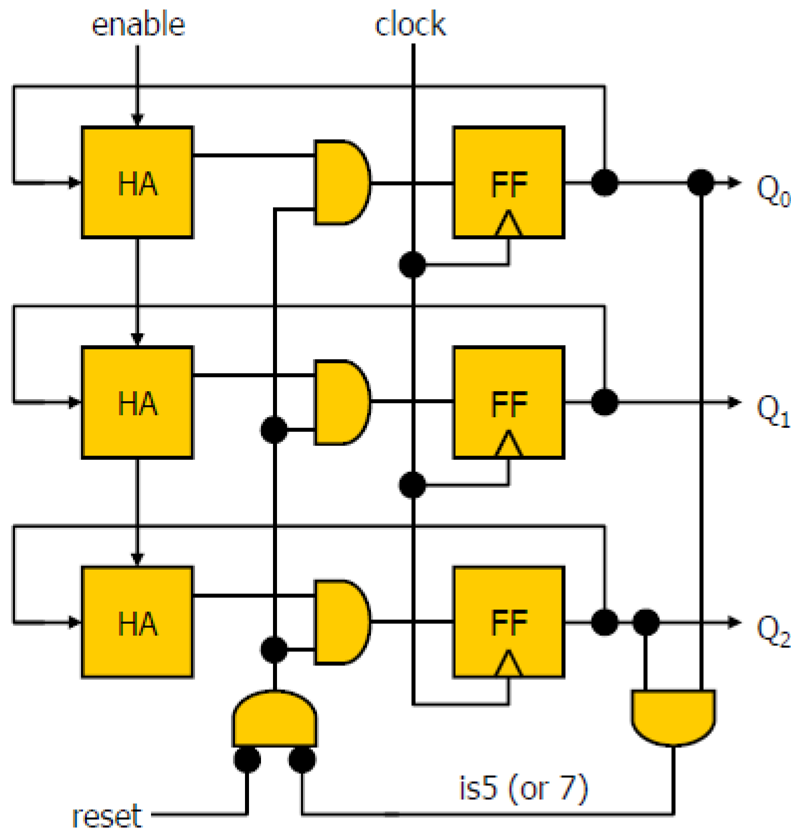


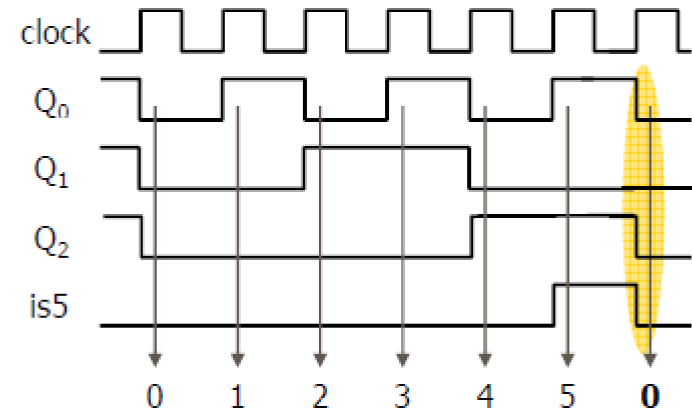


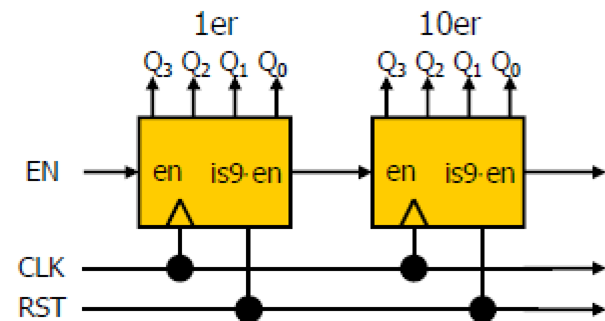
Binärer Zähler mit T-Flipflops

- Dieselbe Struktur wie ein Binärzähler mit Halbaddierer

- 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.



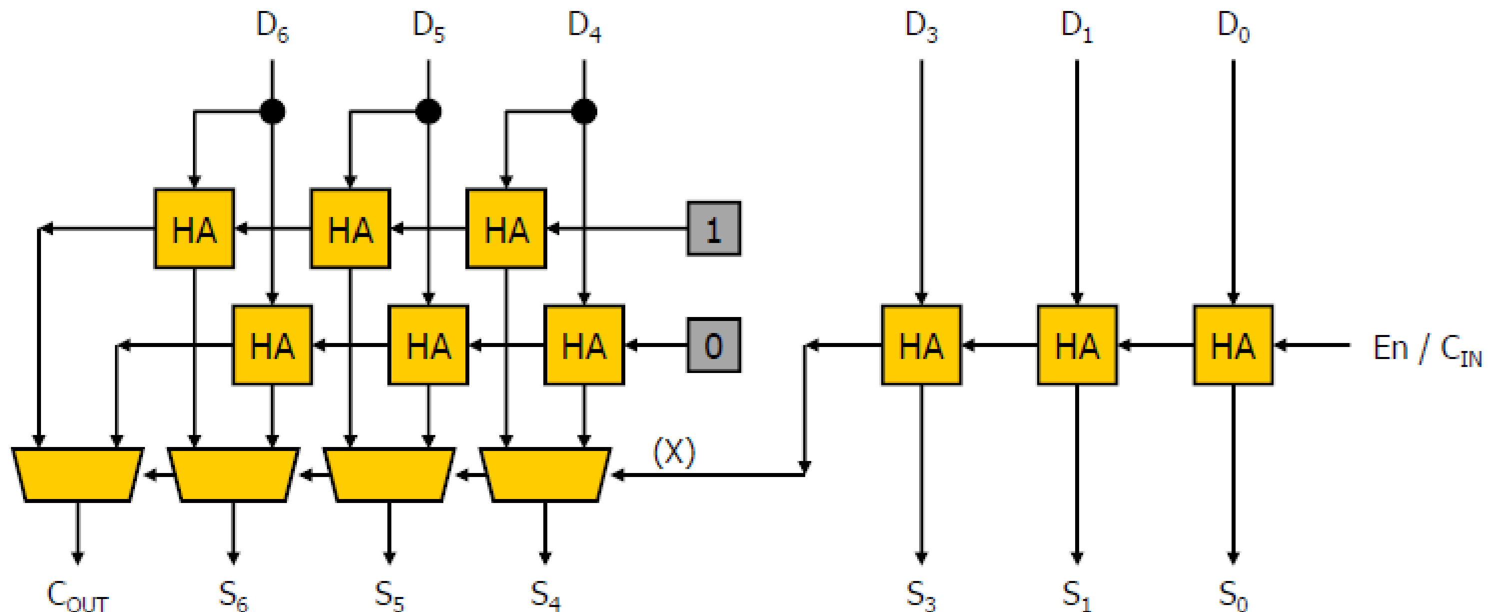




SuS - Lehre (uni-heidelberg.de)

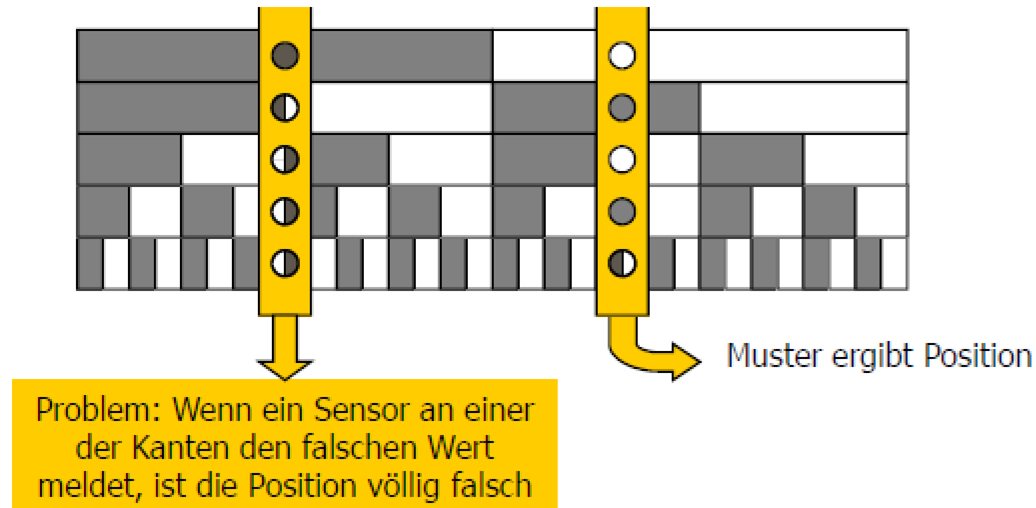
Schnellere Zähler / Addierer

- 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 C_{out} unter den Annahmen $C_{in} = 0$ oder $C_{in} = 1$. Das benötigt ZWEI Addierer.
  - Das C_{out} (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



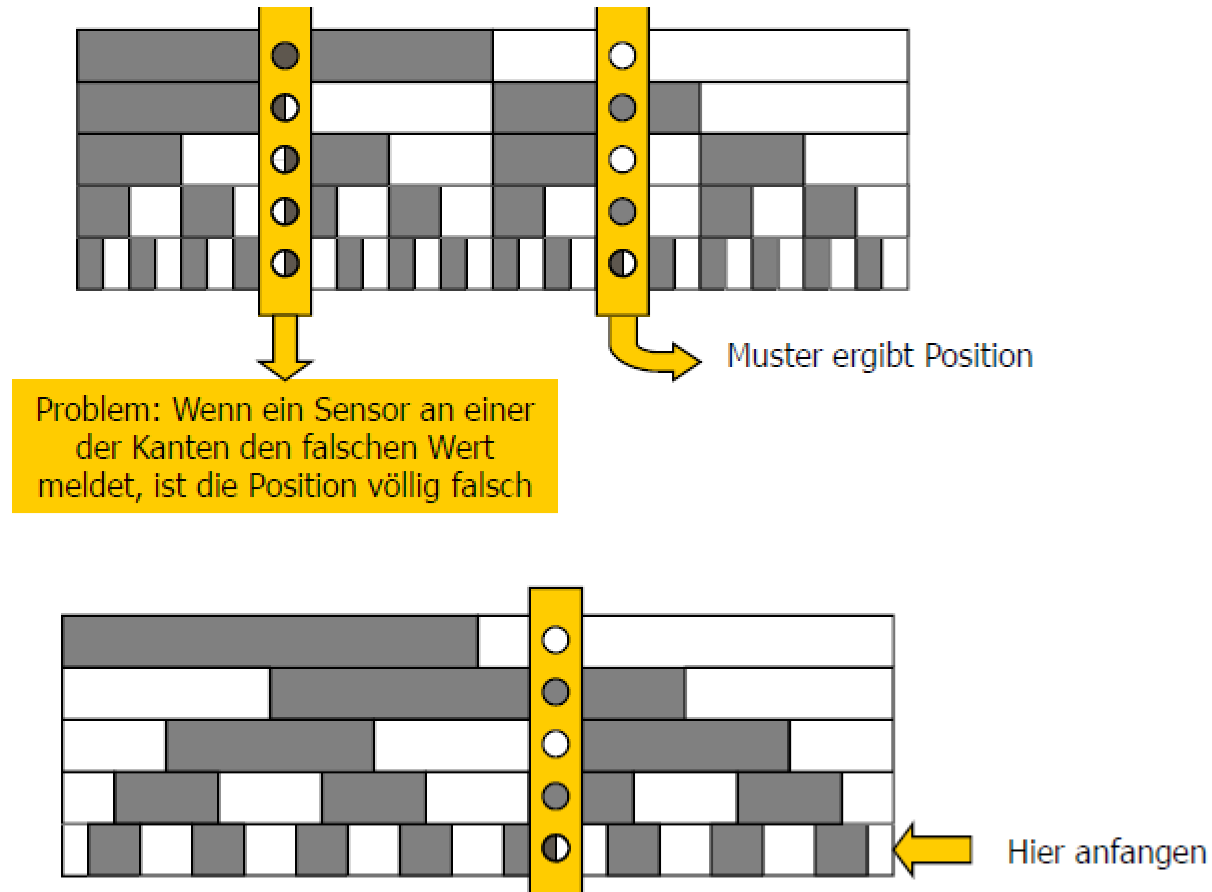
SuS - Lehre (uni-heidelberg.de)

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

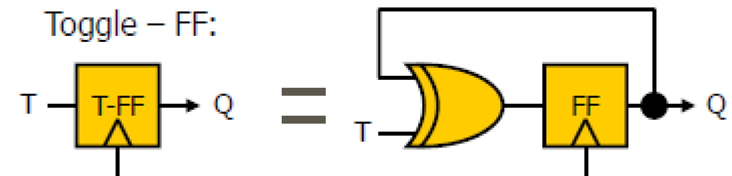


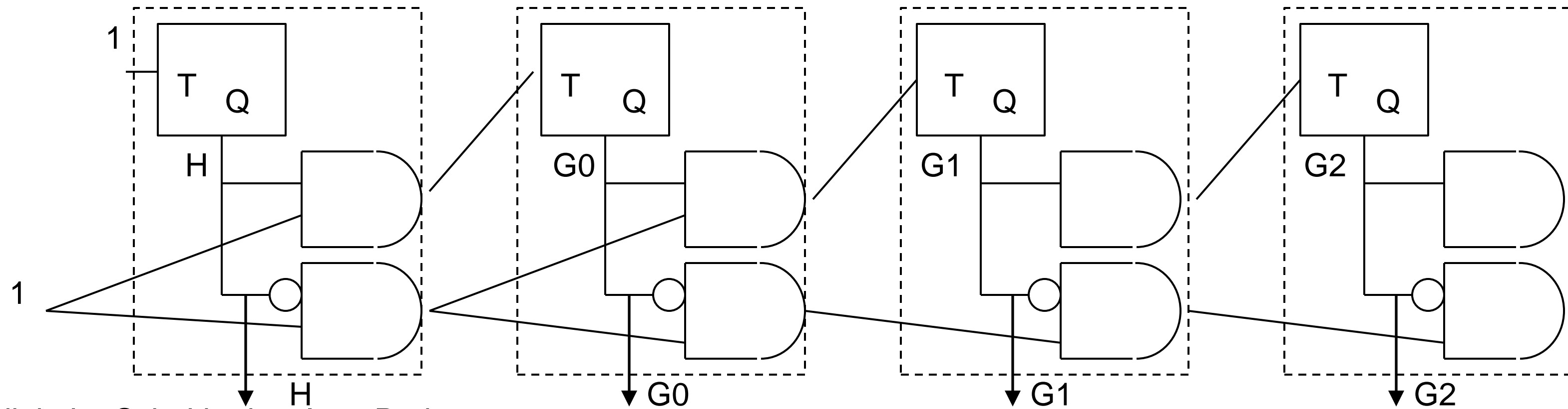
Gray Zähler: Implementierung

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



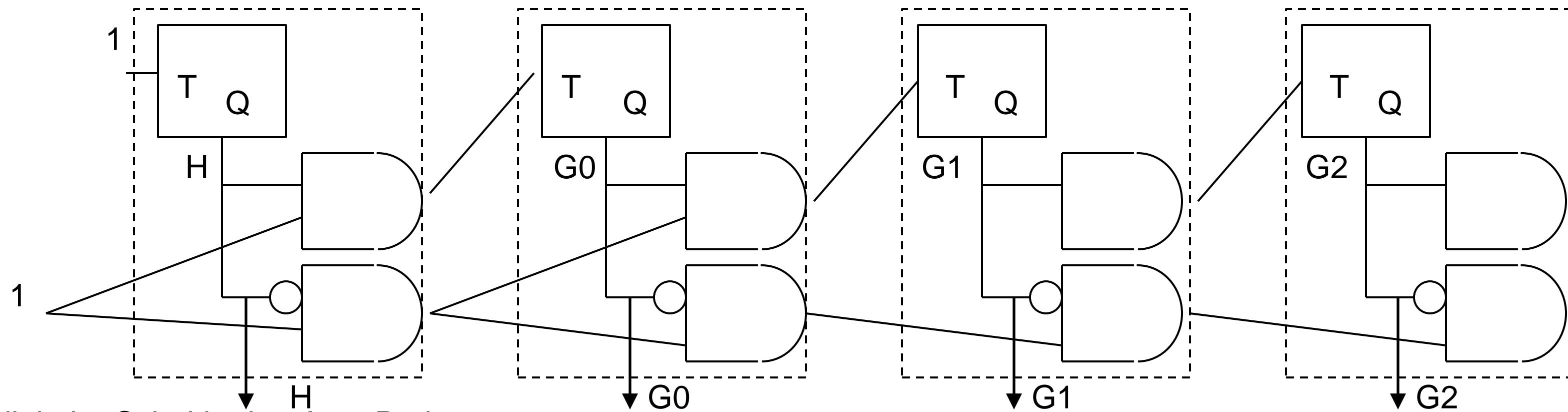
- Implementierung





- Tabelle

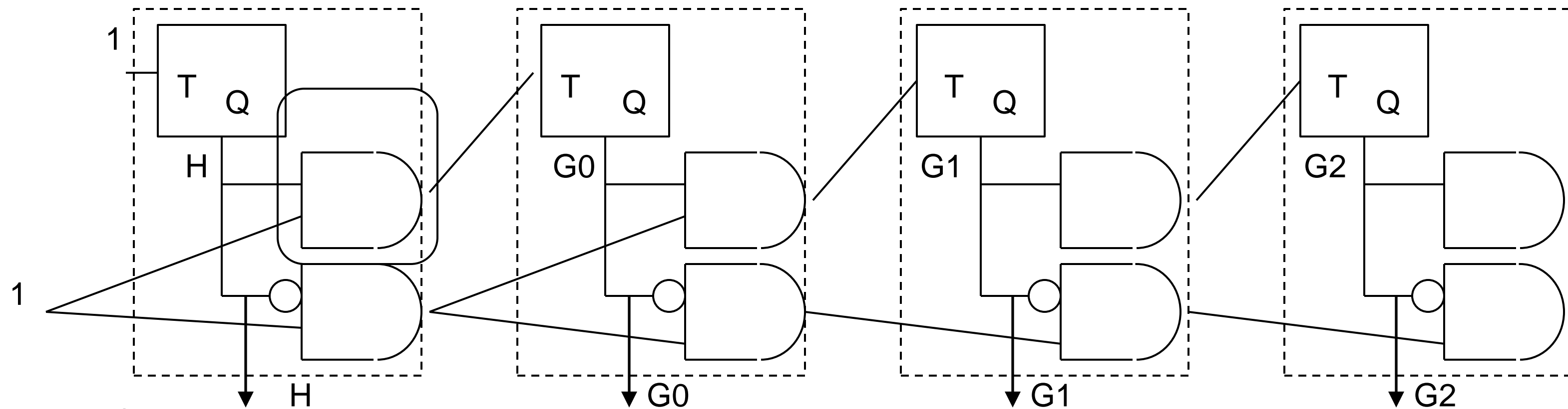
	G2	G1	G0	H
0	0	0	0	1
1	0	0	1	0
2	0	1	1	1
3	0	1	0	0
4	1	1	0	1
5	1	1	1	0
6	1	0	1	1
7	1	0	0	0



Grey-Zähler

- T-Signal vom G0

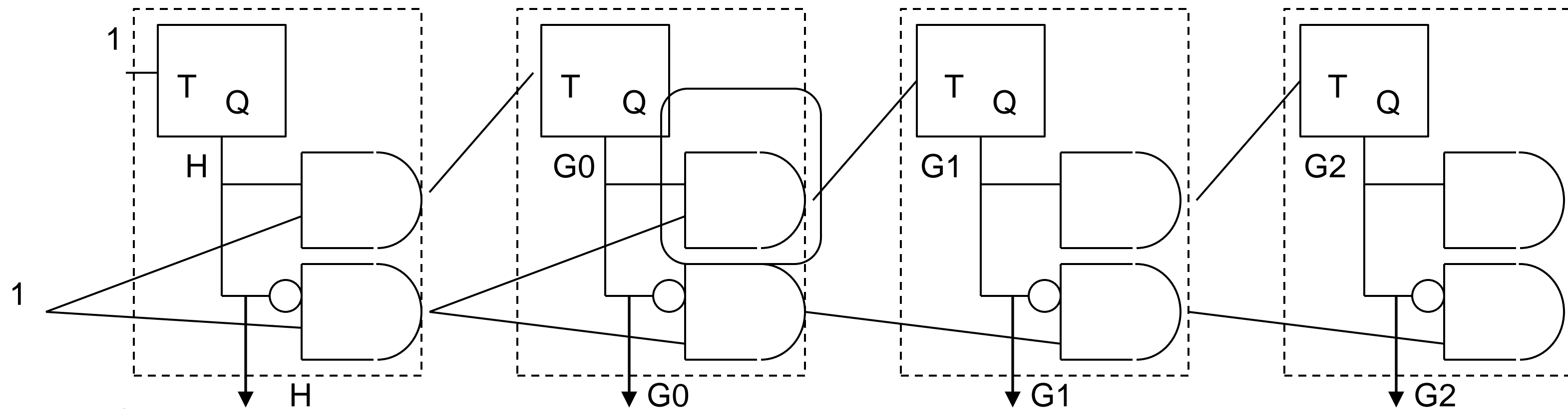
	G2	G1	G0	H
0	0	0	0	1
1	0	0	**1**	0
2	0	1	1	1
3	0	1	**0**	0
4	1	1	0	1
5	1	1	**1**	0
6	1	0	1	1
7	1	0	**0**	0



Grey-Zähler

- T-Signal vom G1

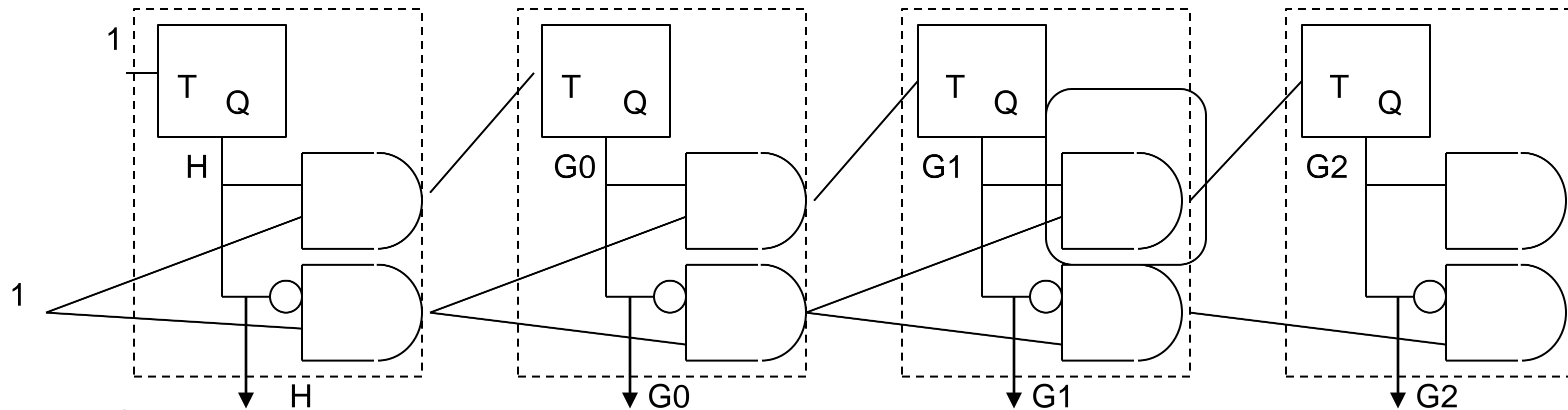
	G2	G1	G0	H
0	0	0	0	1
1	0	0	1	0
2	0	**1**	1	1
3	0	1	0	0
4	1	1	0	1
5	1	1	1	0
6	1	**0**	1	1
7	1	0	0	0



Grey-Zähler

- T-Signal vom G2 (kein MSB)

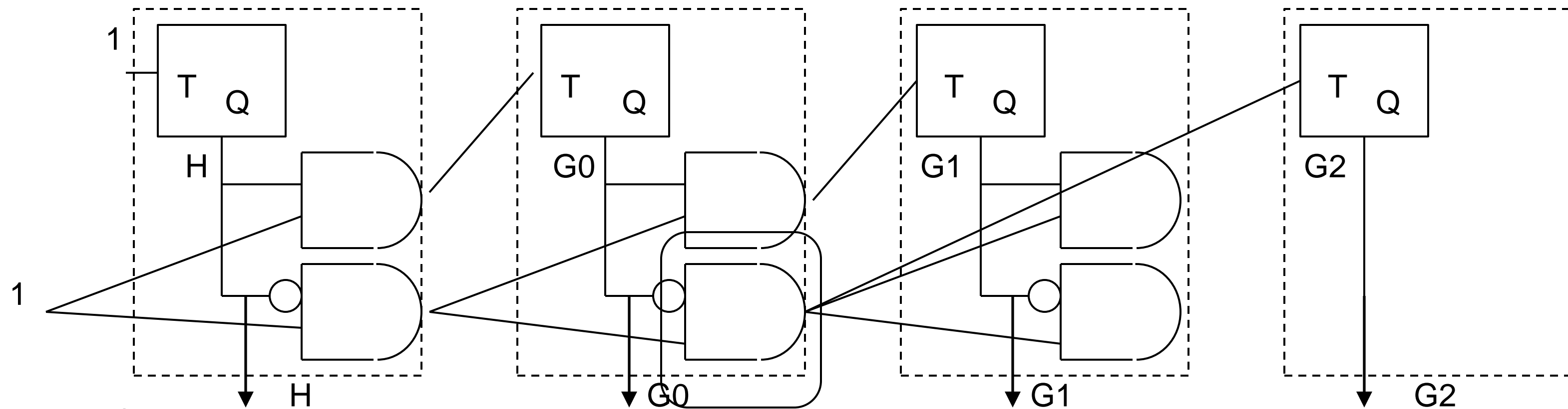
	G2	G1	G0	H
0	0	0	0	1
1	0	0	1	0
2	0	**1**	1	1
3	0	1	0	0
4	**1**	1	0	1
5	1	1	1	0
6	1	**0**	1	1
7	1	0	0	0



Grey-Zähler

- T-Signal vom MSB G2

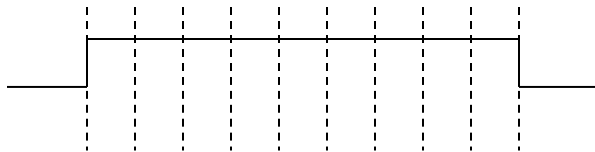
	G2	G1	G0	H
0	0	0	0	1
1	0	0	1	0
2	0	**1**	1	1
3	0	1	0	0
4	**1**	1	0	1
5	1	1	1	0
6	1	**0**	1	1
7	1	0	0	0

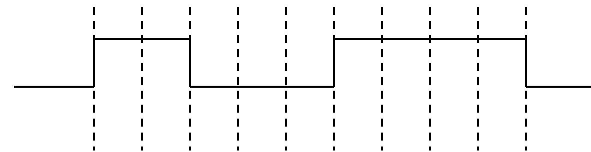


- Zusätzliche Folien

Scrambler

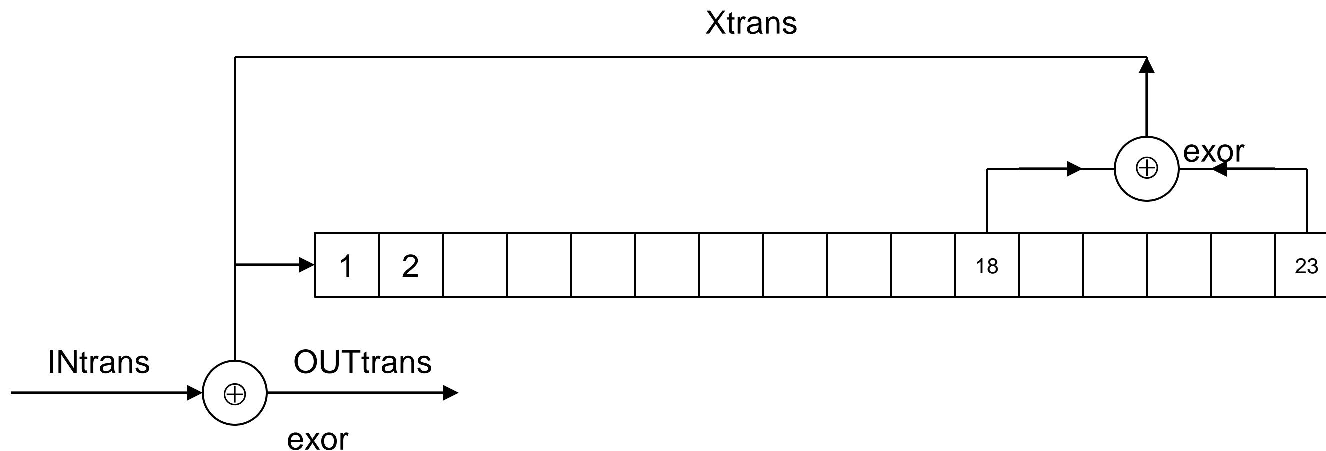
- Ein **Scrambler** (deutsch *Verwürfler*) verwendet linear rückgekoppelte Schieberegister (LFSR), um ein Digitalsignal umkehrbar umzustellen
- Ein Scrambler basierend auf LFSR stellt wegen der einfachen und bekannten Verfahren keine brauchbare Verschlüsselung von Daten dar.
- Ein Scrambler wird durch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


Original


Gescrambeld

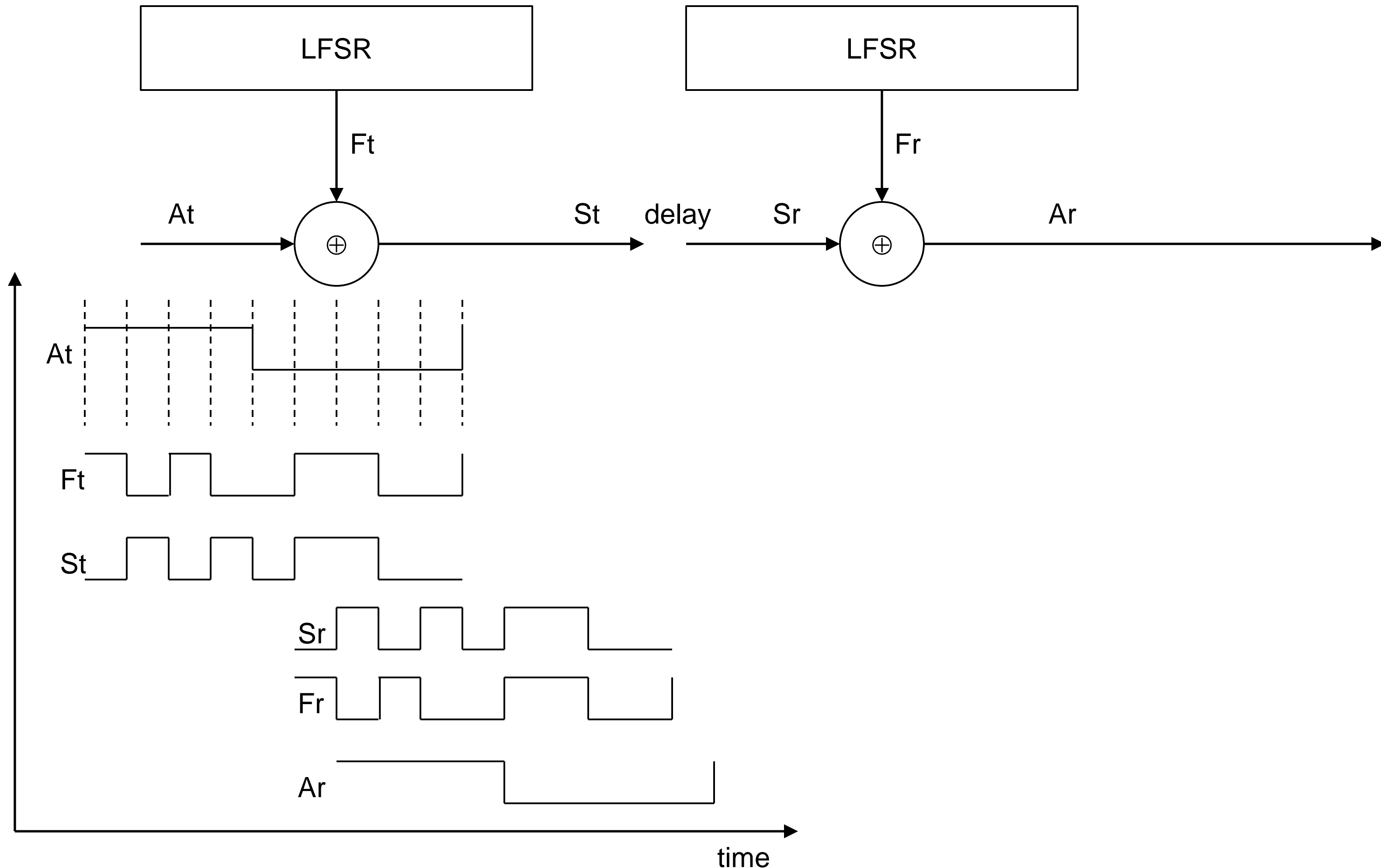
Synchrone (additive) Scrambler

- 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

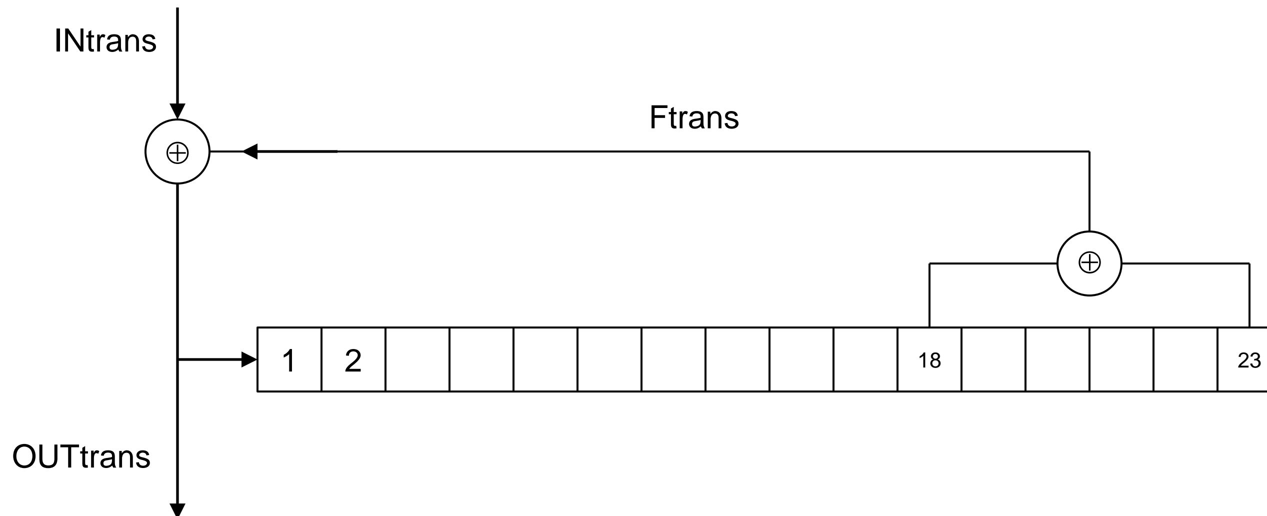
- Descrambling basiert auf der Gleichheit $A = A \oplus F \oplus F$
- Wenn man A zweimal mit gleicher Zahl F „ex-odert“ bekommt man A



Selbstsynchronisierende (multiplikative) Scrambler

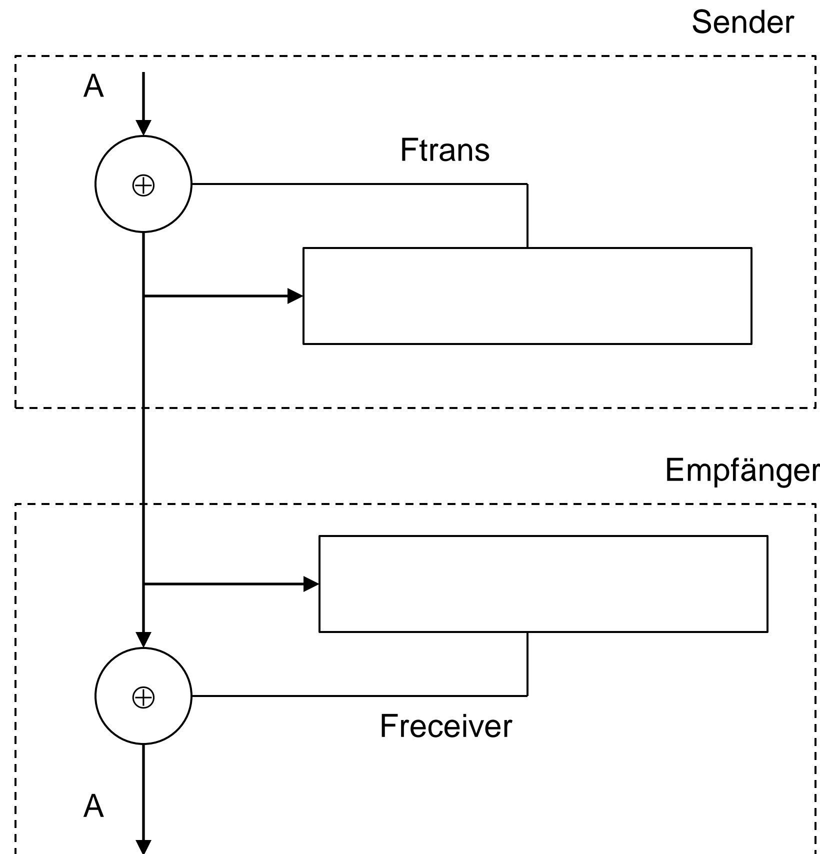
- Selbstsynchronisierende oder auch multiplikative Scrambler benötigen keinen definierten Startwert und auch kein Sync-Wort, um die Codephase des Empfängers mit der Codephase des Senders abzugleichen. Auch kann der Startwert des LFSR beliebig sein.
- Erreicht wird die Funktion der Selbstsynchronität 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

- Tramsmitter (Sender)

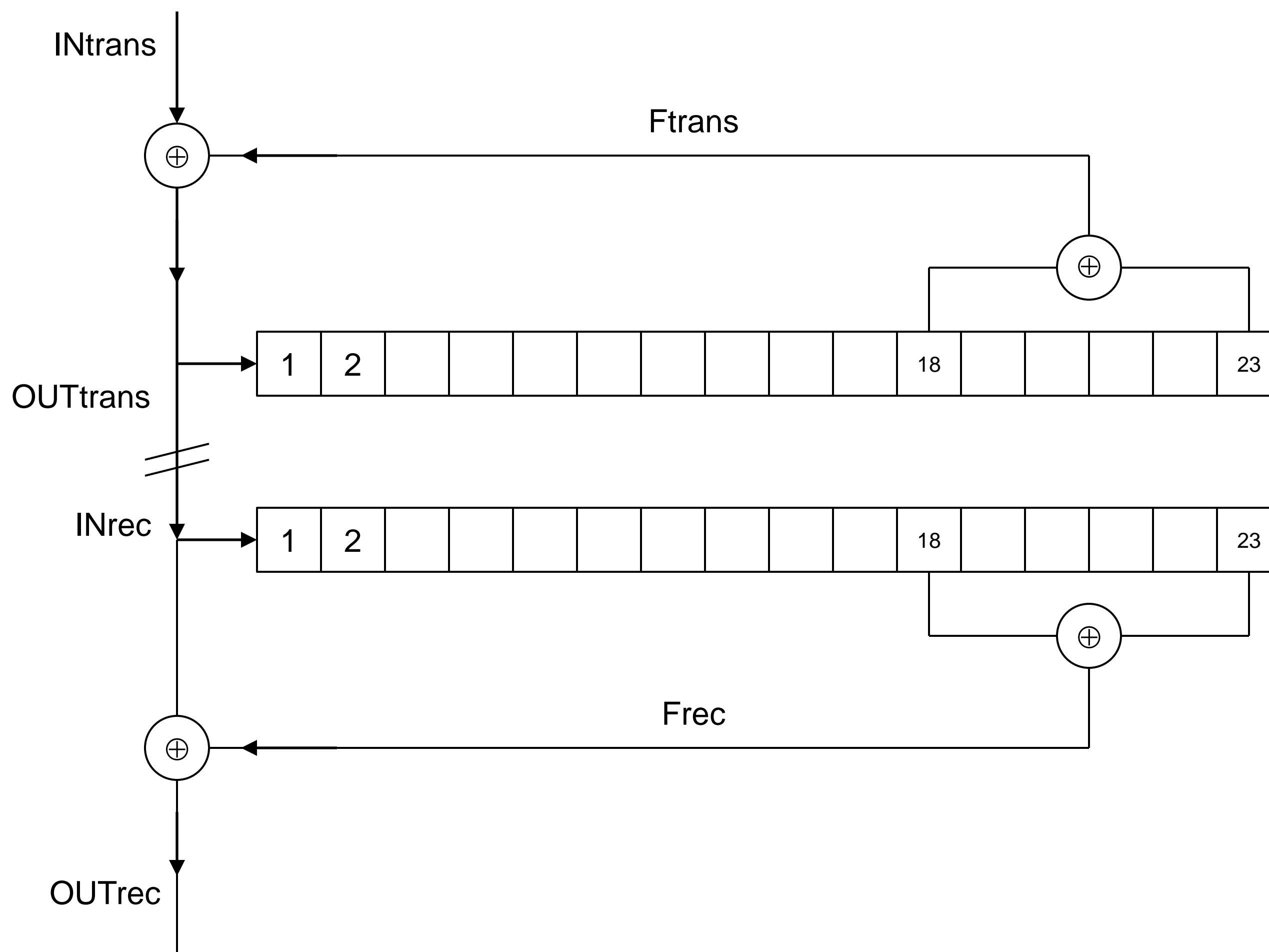


Selbstsynchronisierende (multiplikative) Scrambler

- Descrambling basiert ebenfalls auf der Gleichheit A = A ⊕ F ⊕ F
- Trick: dieselbe Bitequenz wird im Sender und im Empfänger ins Schieberegister reingetaktet, deswegen Ftrans = Freceiver

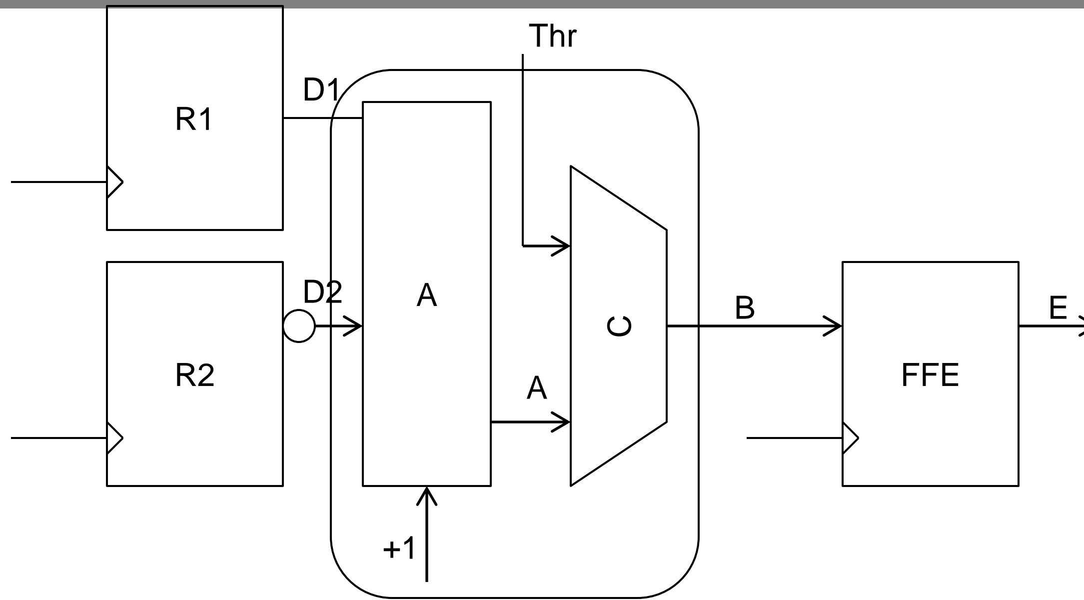


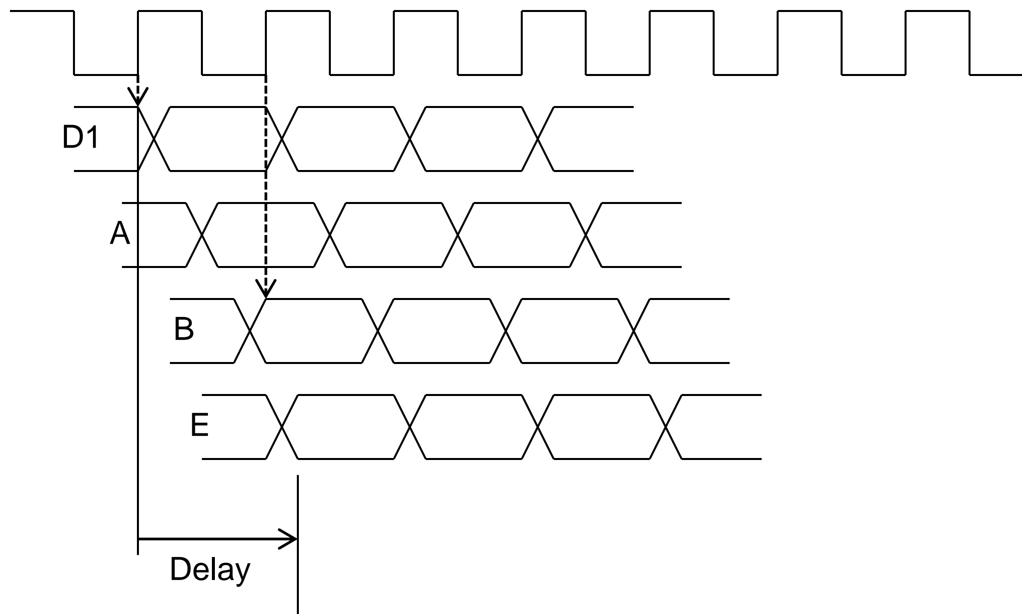
Selbstsynchronisierende (multiplikative) Scrambler



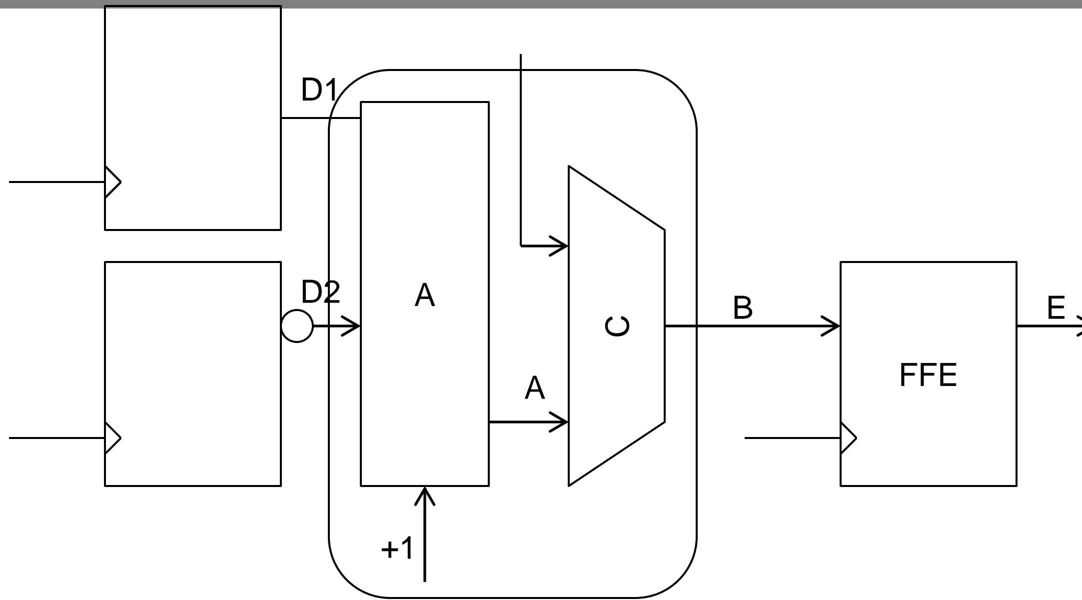
Ftrans = Frec

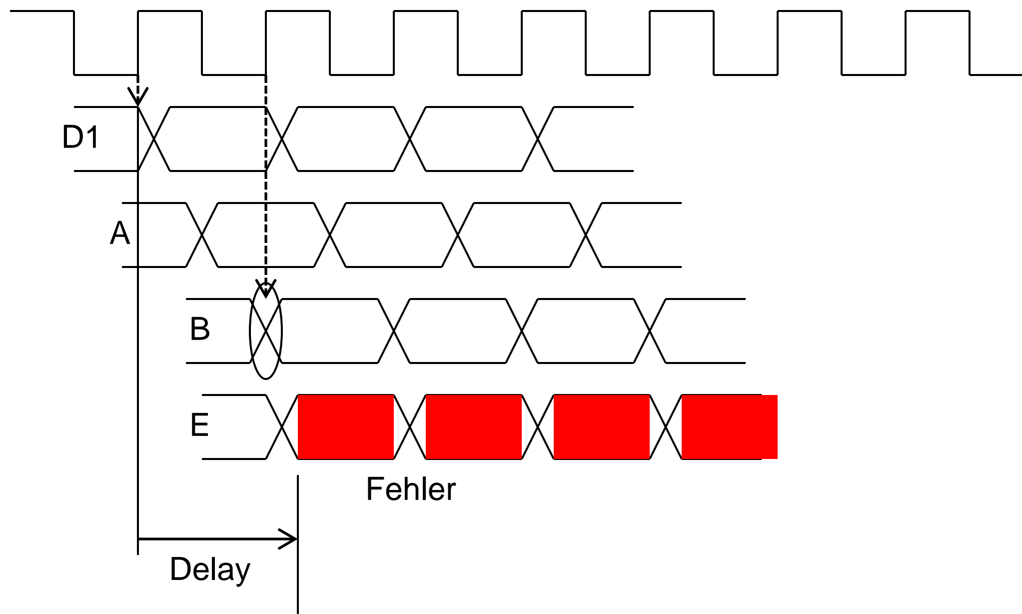
OUTrec = Frec ⊕ INrec = Frec ⊕ (INtrans ⊕ Ftrans) = INtrans ⊕ Ftrans ⊕ Ftrans = INtrans


R1
D1
Thr
D2
A
C
B
E
R2
A
FFE
+1

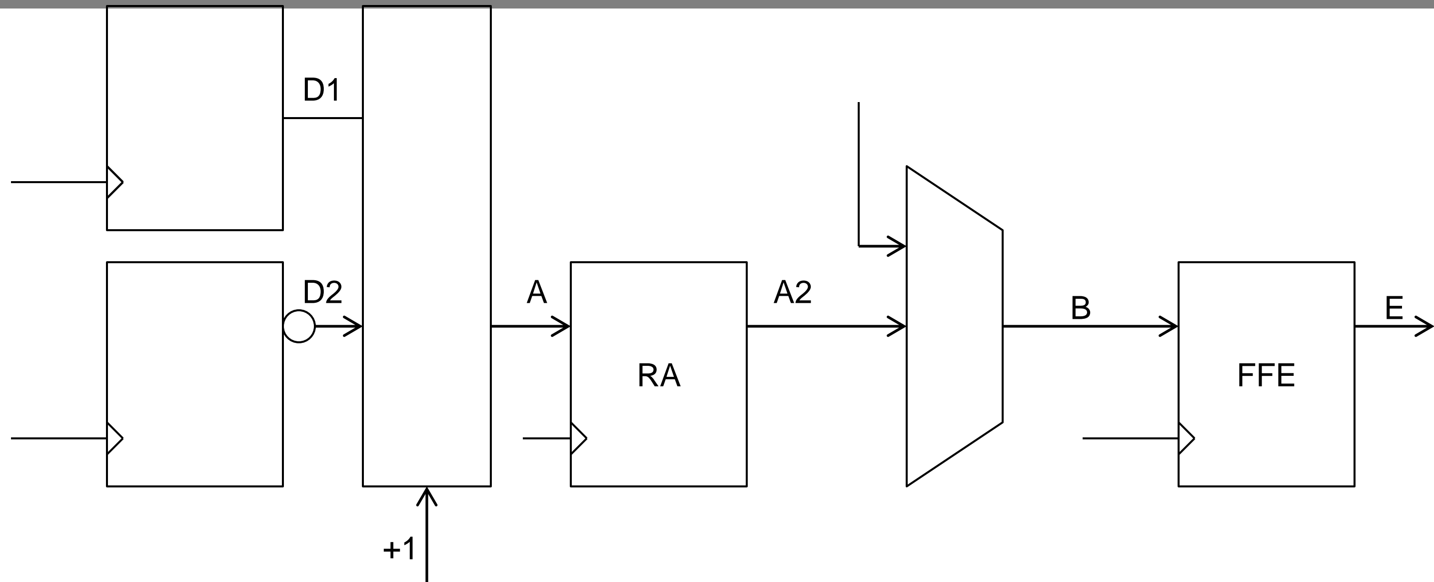

D1
A
B
E
Delay

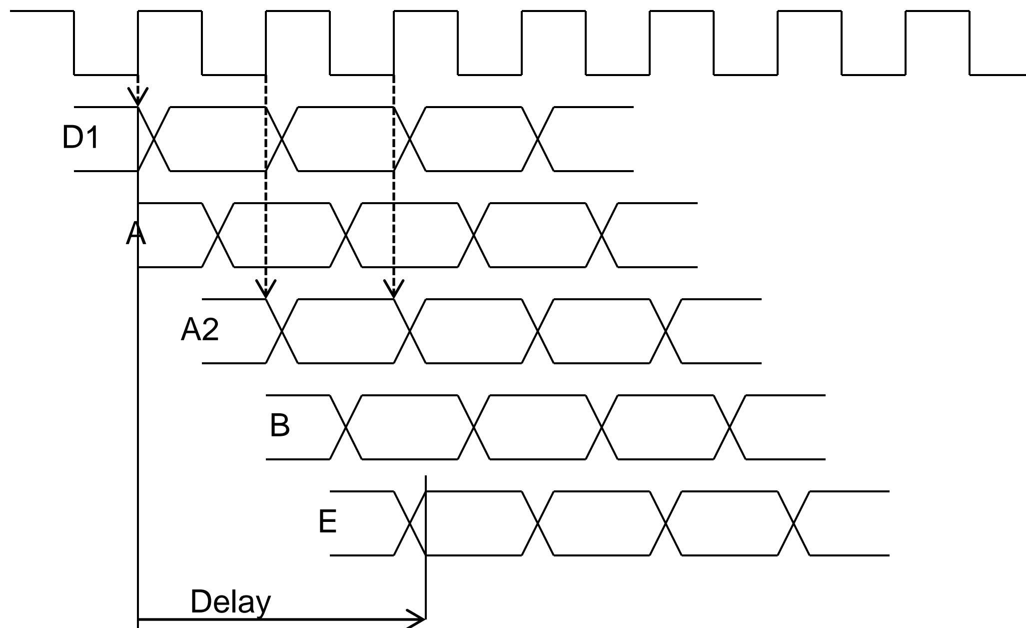
Pipelining





Pipelining





Pipelining