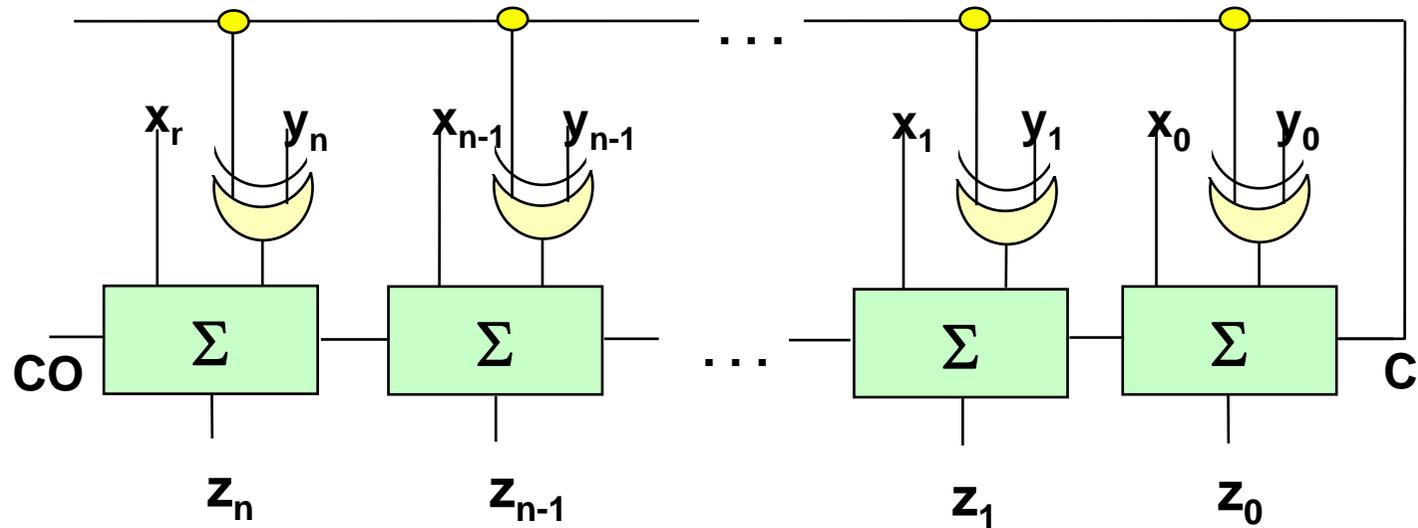


6. Zahlendarstellungen und Rechnerarithmetik



Negative Zahlen, Zweierkomplement
Rationale Zahlen, Gleitkommazahlen
Halbaddierer, Volladdierer
Paralleladdierwerk, von-Neumann-Addierwerk
Multiplikation: Barrel-Shifter, Iterative Realisierung
Algorithmus von Booth
Aufbau einer ALU

Darstellung ganzer Zahlen

Die einfachste Möglichkeit zur Unterscheidung positiver und negativer Zahlen ist die **Interpretation des ersten Bits als Vorzeichen**:

0 000	=	0
0 001	=	1
0 010	=	2
0 011	=	3
0 100	=	4
0 101	=	5
0 110	=	6
0 111	=	7

1 000	=	- 0
1 001	=	- 1
1 010	=	- 2
1 011	=	- 3
1 100	=	- 4
1 101	=	- 5
1 110	=	- 6
1 111	=	- 7

Es gibt zwei verschiedene Darstellungen für 0.

Darstellung ganzer Zahlen 2

Entsprechend können mit 8 Bits, also mit einem Byte, ganze Zahlen von -127 bis 127 dargestellt werden:

0000 0000	=	0
0000 0001	=	1
0000 0010	=	2
0000 0011	=	3
0000 0100	=	4
0000 0101	=	5
0000 0110	=	6
0000 0111	=	7
0000 1000	=	8
0000 1001	=	9
0000 1010	=	10
0000 1011	=	11
0000 1100	=	12
0000 1101	=	13
0000 1110	=	14
0000 1111	=	15

0001 0000	=	16
0001 0001	=	17
..
0111 1111	=	127
1000 0000	=	- 0
..
1111 1010	=	- 122
1111 1011	=	- 123
1111 1100	=	- 124
1111 1101	=	- 125
1111 1110	=	- 126
1111 1111	=	- 127

Nachteile der Vorzeichendarstellung

1. Die Darstellung negativer Zahlen verändert sich bei Bereichserweiterungen:

- 5 als 4-Bit Zahl = 1101
- 5 als 1-Byte Zahl = 1000 0101
- 5 als 2-Byte Zahl = 1000 0000 0000 0101

2. Die Addition einer positiven und einer negativen Zahl funktioniert anders als üblich:

- 5
+ 12

7

1000 0101
+ 0000 1100

+ 0 000 0111

Wie kommt das zustande ?

3. Null hat zwei verschiedene Darstellungen:

1000 0000 = 0
0000 0000 = 0

Einerkomplementdarstellung

In der Einerkomplementdarstellung negiert man Zahlen durch bitweises Komplementieren.

0000	=	0
0001	=	1
0010	=	2
0011	=	3
0100	=	4
0101	=	5
0110	=	6
0111	=	7

1000	=	- 7
1001	=	- 6
1010	=	- 5
1011	=	- 4
1100	=	- 3
1101	=	- 2
1110	=	- 1
1111	=	- 0

**Das erste Bit gibt das Vorzeichen an.
Es gibt zwei Darstellungen für Null.**

Einerkomplement (Definition)

Die Bitfolge

$z_n \quad z_{n-1} \quad z_{n-2} \quad \dots \quad z_1 \quad z_0$

repräsentiert die Binärzahl

$$- z_n * (2^n - 1) + z_{n-1} * 2^{n-1} + z_{n-2} * 2^{n-2} + \dots + z_1 * 2^1 + z_0$$

Eigenschaften:

- Symmetrischer **Zahlenbereich**: $[-(2^n-1) .. (2^n-1)]$
- **Null** hat zwei Darstellungen: $0\dots 0$ und $1\dots 1$
- **Negieren** durch bitweises Komplementieren
- **Bereichserweiterung** durch Auffüllen mit dem Vorzeichenbit
- **Addition** mit Standardaddierwerk möglich, aber Vorsicht:
 - **Überlauf** liegt nur dann vor, wenn $a_n + b_n - \text{carry}_n \notin \{0,1\}$
 - **End-around-carry**: carry_n zum Ergebnis dazuaddieren!

Zweierkomplementdarstellung

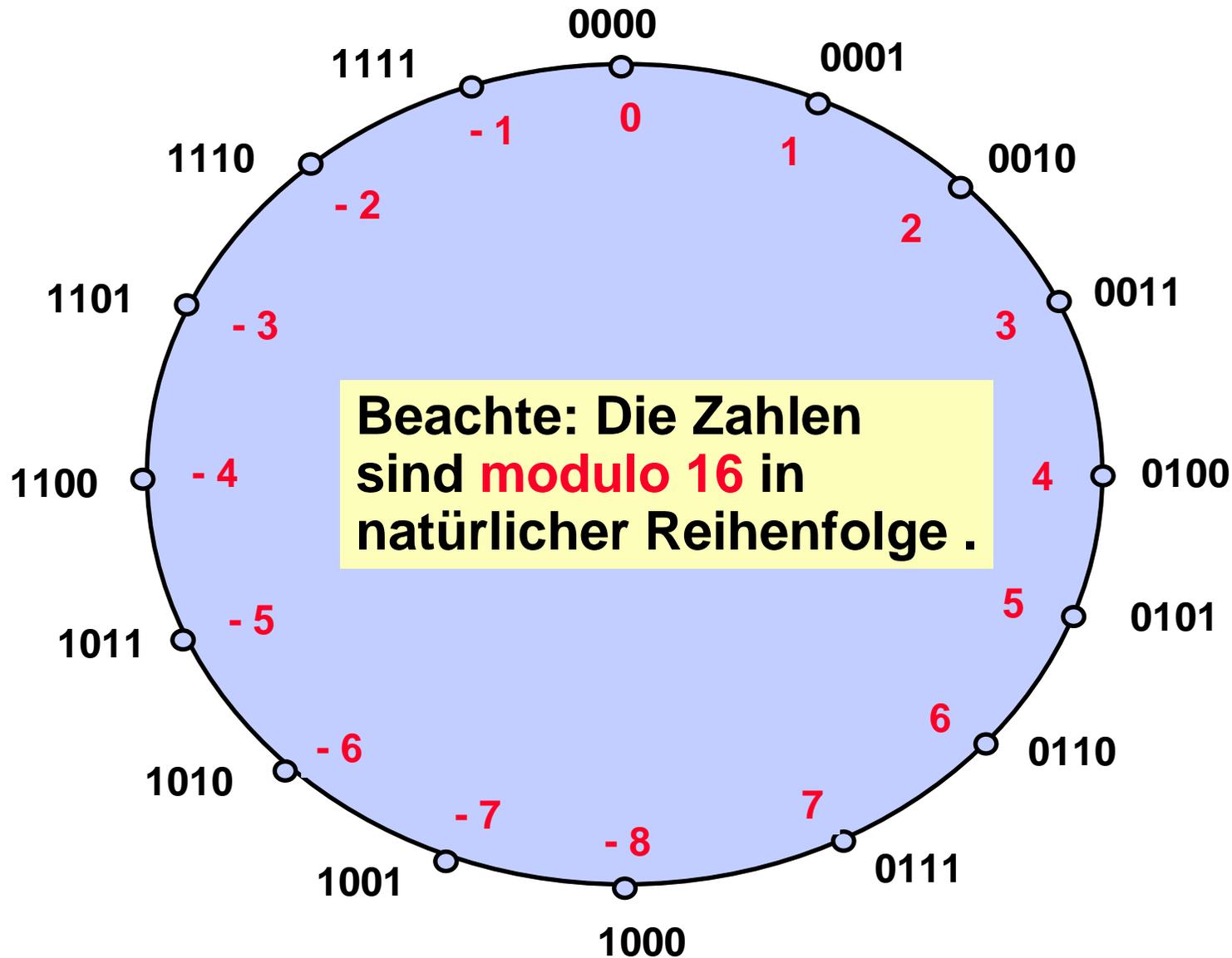
In der Zweierkomplementdarstellung durchläuft man zunächst die positiven Zahlen, dann die negativen Zahlen in umgekehrter Reihenfolge

0000	=	0
0001	=	1
0010	=	2
0011	=	3
0100	=	4
0101	=	5
0110	=	6
0111	=	7

1000	=	- 8
1001	=	- 7
1010	=	- 6
1011	=	- 5
1100	=	- 4
1101	=	- 3
1110	=	- 2
1111	=	- 1

Auch in der Zweierkomplementdarstellung gibt die erste Ziffer das Vorzeichen an.

Zweierkomplement am Zahlenkreis



Zweierkomplement (Definition)

Die Bitfolge

$z_n \quad z_{n-1} \quad z_{n-2} \quad \dots \quad z_1 \quad z_0$

repräsentiert die Binärzahl

$$- z_n * 2^n + z_{n-1} * 2^{n-1} + z_{n-2} * 2^{n-2} + \dots + z_1 * 2^1 + z_0$$

Eigenschaften:

- Asymmetrischer **Zahlenbereich**: $[-2^n .. (2^n-1)]$
- **Null** hat eindeutige Darstellung $0\dots 0$.
- **Negieren** durch bitweises Komplementieren und Addition von Eins
- **Bereichserweiterung** durch Auffüllen mit dem Vorzeichenbit
- **Addition** mit Standardaddierwerk möglich, aber Vorsicht:
 - **Überlauf** liegt nur dann vor, wenn $a_n + b_n - \text{carry}_n \notin \{0,1\}$

Zweierkomplement für $n=3$

Für $n = 3$ ergibt sich:

$$1000 = -8$$

$$1001 = -7$$

$$1010 = -6$$

$$1011 = -5$$

$$1100 = -4$$

$$1101 = -3$$

$$1110 = -2$$

$$1111 = -1$$

$$0000 = 0$$

$$0001 = 1$$

$$0010 = 2$$

$$0011 = 3$$

$$0100 = 4$$

$$0101 = 5$$

$$0110 = 6$$

$$0111 = 7$$

Zweierkomplement für $n = 31$

Für $n = 31$ ergibt sich:

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 = -2^{31} = -2.147.483.648$$

$$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001 = -2.147.483.647$$

...

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110 = -2$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 = -1$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000 = 0$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001 = 1$$

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010 = 2$$

...

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110 = 2.147.483.646$$

$$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111 = 2.147.483.647$$

Negation in Zweierkomplementdarstellung

Addiert man zu einer Zahl ihre Komplementärzahl, so erhält man die Zahl 111 ... 1111

$$\begin{aligned}111 \dots 1111 &= -2^n + 2^{n-1} + \dots + 2^1 + 1 \\ &= -2^n + (2^n - 1) \\ &= -1\end{aligned}$$

Eine Zahl plus ihr Komplement ergibt also die Darstellung von -1.

Daher negiert man eine Zahl, indem man zu ihrer Komplementärzahl 1 addiert.

Grundrechenarten bei Zweierkomplementzahlen

Addition

$$\begin{array}{r} -5 \\ + 6 \\ \hline 1 \end{array}$$

in Zweierkomplementdarstellung :

$$\begin{array}{r} 1011 \\ + 0110 \\ \hline 0001 \end{array}$$

Wie gewohnt, aber **Vorsicht** bei Bereichsüberschreitung

Negation

$$5$$

in Zweierkomplementdarstellung :

$$0101$$

1010

Komplementiere alle Bits, dann addiere 1

$$-5$$

in Zweierkomplementdarstellung :

$$1011$$

Subtraktion

negiere zweiten Operanden, dann addiere

Addition von Zweierkomplementzahlen

Zifferndarstellung

$$\begin{array}{r}
 u_n \ u_{n-1} \ \dots \ u_1 \ u_0 \\
 + \ v_n \ v_{n-1} \ \dots \ v_1 \ v_0 \\
 \hline
 = \ w_n \ w_{n-1} \ \dots \ w_1 \ w_0 \ ?
 \end{array}$$

Zahlenwert

$$\begin{aligned}
 & -u_n \cdot 2^n + u_{n-1} \cdot 2^{n-1} + \dots + u_1 \cdot 2^1 + u_0 \\
 & + (-v_n \cdot 2^n + v_{n-1} \cdot 2^{n-1} + \dots + v_1 \cdot 2^1 + v_0)
 \end{aligned}$$

normale
Addition

$$= (-u_n - v_n) \cdot 2^n + u_{n-1} \cdot 2^{n-1} + \dots + u_1 \cdot 2^1 + u_0 + v_{n-1} \cdot 2^{n-1} + \dots + v_1 \cdot 2^1 + v_0$$

$$= -u_n \cdot 2^n - v_n \cdot 2^n + (c_i \cdot 2^n + w_{n-1} \cdot 2^{n-1} + \dots + w_1 \cdot 2^1 + w_0)$$

$$= -(u_n + v_n - c_i) \cdot 2^n + w_{n-1} \cdot 2^{n-1} + \dots + w_1 \cdot 2^1 + w_0$$

$$= w_n$$

Die Addition verläuft also wie bei natürlichen Zahlen.
Solange keine Bereichsüberschreitung auftritt,
also solange $0 \leq (u_n + v_n - c_i) \leq 1$ gilt,
ist das Ergebnis exakt.

Beispiele

90	0101 1010	
- 67	1011 1101	
-----	0001 0111	
29		

0101 1010	90	
0011 0101	53	
-----	143 = -113 mod 128	
1000 1111		

$$u_n + v_n - c_i_n = -1$$

Bereichsüberschreitung !

- 40	1101 1000	
- 83	1010 1101	
-----	1000 0101	
-123		

1101 1000	- 40	
1000 1101	- 115	
-----	- 155 = 101 mod 128	
0110 0101		

$$u_n + v_n - c_i_n = 2$$

Bereichsüberschreitung !

Rationale Zahlen als Festkommazahlen

In einem allgemeinen Ziffersystem mit einer Basiszahl b kann man auch **rationale Zahlen** definieren:

Mit den Ziffern $X_n, X_{n-1}, \dots, X_1, X_0$ und Y_1, Y_2, \dots, Y_{m-1} mit $X_i, Y_j \in \{0, 1, \dots, b-1\}$ kann man die gebrochene Zahl

$$X = (X_n X_{n-1} \dots X_1 X_0 . Y_1 Y_2 \dots Y_{m-1})_b$$

mit dem Zahlenwert $X = X_n \cdot b^n + \dots + X_1 \cdot b^1 + X_0 + Y_1 \cdot b^{-1} + Y_2 \cdot b^{-2} + \dots + Y_m \cdot b^{-m}$ bilden.

Beispiele:	<u>gebrochene Binärzahl</u>	<u>gebrochene Dezimalzahl</u>
	0 . 1	0 . 5
	0 . 01	0 . 25
	0 . 11	0 . 75
	1 . 11	1 . 75
	111 . 111	7 . 875
	11011101011 . 100111011	1771 . 6162109375
	0 . 00011001100110011....	0 . 1

Das letzte Beispiel zeigt, dass die Dezimalzahl 0.1 nur als unendlich lange periodische, rationale Binärzahl dargestellt werden kann.

Gleitkommazahlen

Analog zu der Schreibweise $4711 = 0.4711 * 10^4$

werden binäre Gleitkommazahlen gebildet.

Gleitkommazahlen (engl.: floating point numbers)

bestehen aus:

- dem Vorzeichen V
- dem Exponenten E
- der Mantisse M

V, E und M repräsentieren dann die Zahl

$$(-1)^V * M * 2^E$$

Vorzeichen, Exponent und Mantisse

Die **Mantisse** besteht aus Binärziffern $m_1 \dots m_n$ und wird als

$$m_1 \times 2^{-1} + m_2 \times 2^{-2} + \dots + m_n \times 2^{-n}$$

interpretiert.



Das **Vorzeichenbit** gibt an, ob die Zahl positiv oder negativ ist.

Der **Exponent** ist eine Binärzahl, z. B. im Bereich -128 bis +127.

Normierte Gleitkommazahlen

Durch Verschieben des Kommas und gleichzeitiger Anpassung des Exponenten kann man erreichen, dass die erste Stelle der Mantisse 1 ist.

Diese Darstellung heißt **normierte Gleitkommadarstellung**.

$$\begin{aligned} & \boxed{0.01010111 * 2^{14}} \\ & = \boxed{0.1010111 * 2^{13}} \\ & = \boxed{1.010111 * 2^{12}} \end{aligned}$$

normiert

Normierte Gleitkommazahlen haben den Vorteil, dass die Mantissenbits optimal ausgenutzt werden, da keine überflüssigen Nullen gespeichert werden müssen.

Die führende 1 braucht auch nicht gespeichert zu werden.

Verschobene Exponenten (biased notation)

- **Problem:** Durch die Normierung kann die Zahl 0 nicht mehr dargestellt werden.
=> Reserviere Bitfolge 00...0 für die Null
- **Neues Problem:** Nun hat die 1 keine Darstellung mehr.
=> **biased notation: Ausrichtung des Zahlenbereichs** des Exponenten, so dass
 - 00...0 den kleinsten Exponenten, also -2^m und
 - 11...1 den größten Exponenten, also 2^m-1repräsentiert.

Es gilt: **tatsächlicher Exponent = Exponentdarstellung – 2^m**

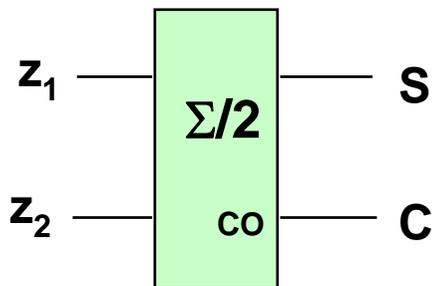
Vorteile:

- eindeutige übliche Darstellung der Null
- normale Kleinerbeziehung auf Gleitkommazahlen

Halbaddierer

Ein Halbaddierer addiert zwei Binärziffern. An den Eingängen liegen die Ziffern z_1 und z_2 . An dem Ausgang S liegt die letzte Stelle der Summe, an dem Ausgang C (für Carry) liegt der Übertrag (0 oder 1).

Schaltzeichen



Schaltfunktion

x	y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Term

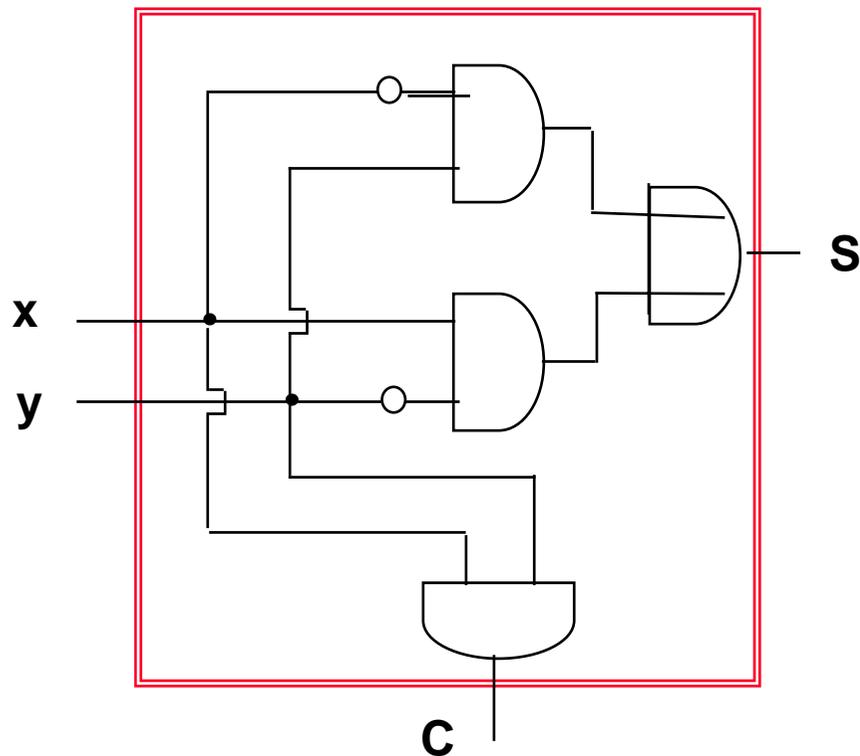
$$\begin{aligned} S &= x' y + x y' \\ &= x \text{ xor } y \\ C &= x y \end{aligned}$$

Realisierung des Halbaddierers

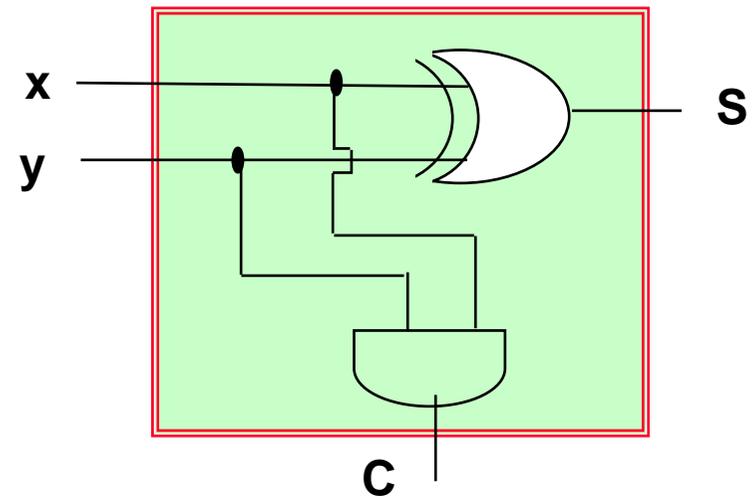
Aus den Booleschen Termen

$$S = x' y + x y' \quad \text{und} \quad C = x y$$

lässt sich sofort eine Realisierung des Halbaddierers gewinnen:



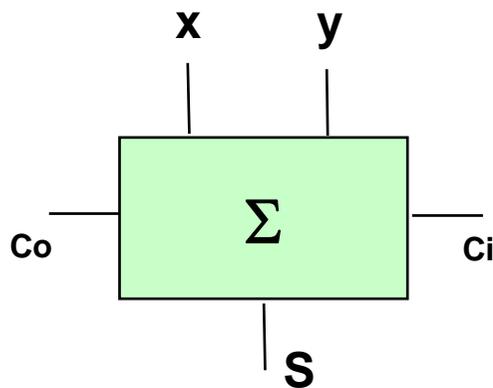
mit einem XOR-Gatter wird die Schaltung einfacher :



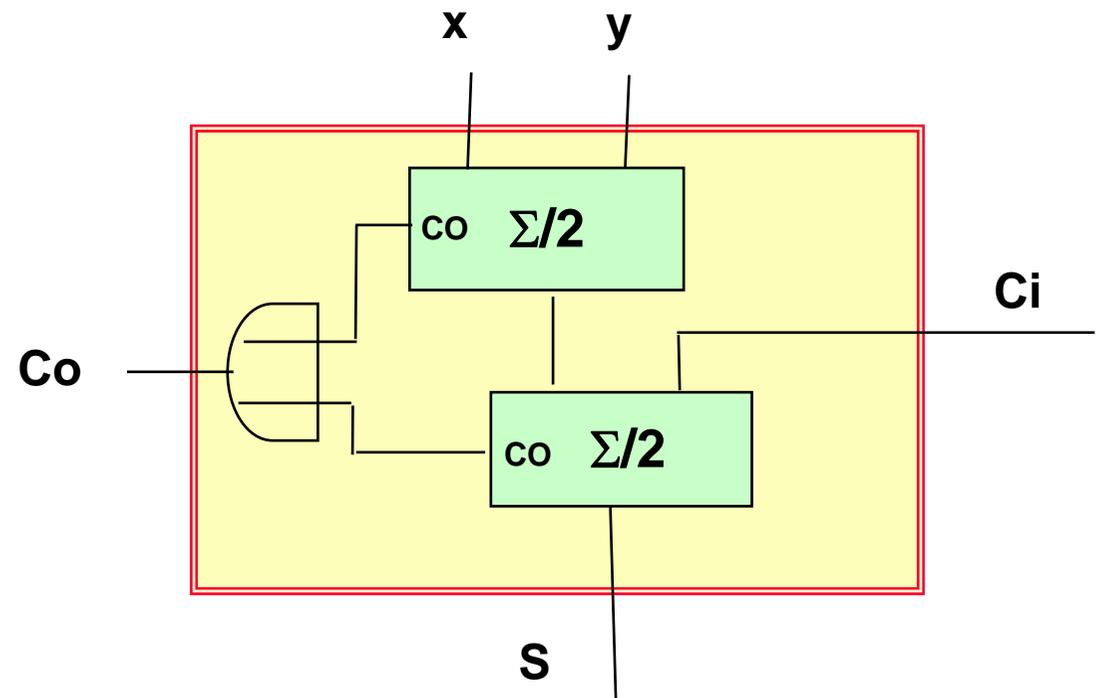
Volladdierer

Ein Volladdierer addiert zwei Ziffern und eine Übertragstelle (C_i = carry in). Am Ausgang S liegt die letzte Ziffer der Summe und am Ausgang C_o (carry out) der Übertrag.

Ein Volladdierer lässt sich mit Hilfe von Halbaddierern konstruieren:



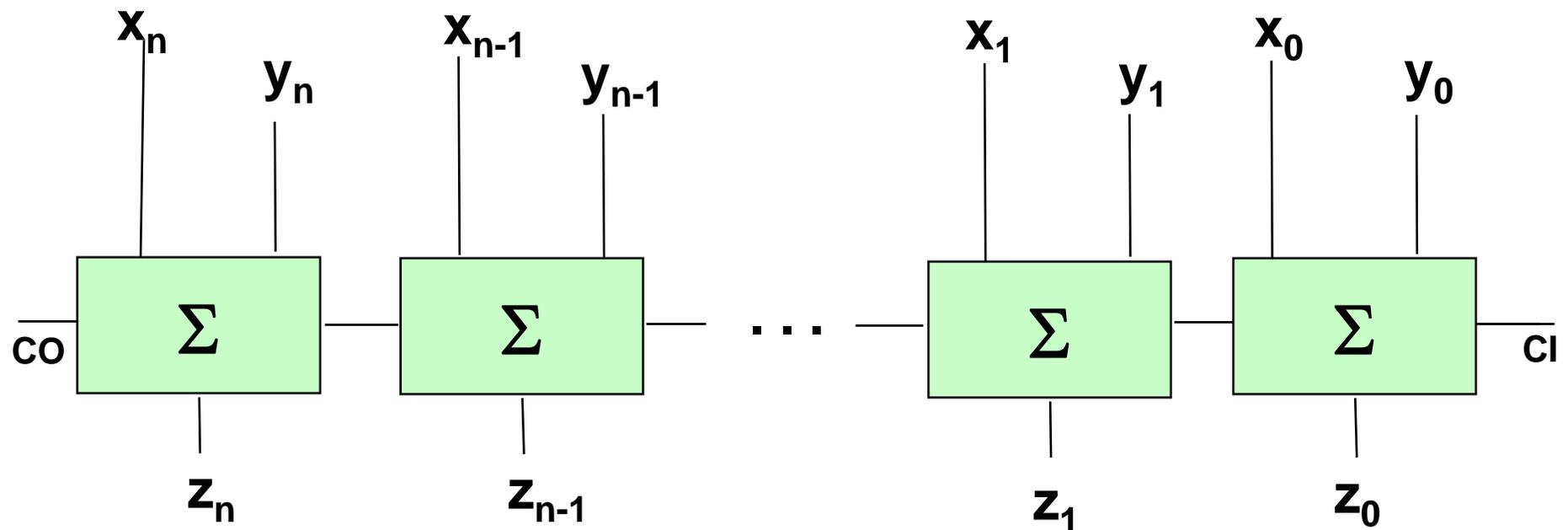
C_i = carry in
 C_o = carry out
 S = Summe



Paralleladdierwerk

Mit einer Kaskade von n Volladdierern läßt sich somit ein Addierwerk für Binärzahlen mit n Stellen realisieren.

Seien $x = x_{n-1} \dots x_0$ und $y = y_{n-1} \dots y_0$ die zu addierenden Binärzahlen.



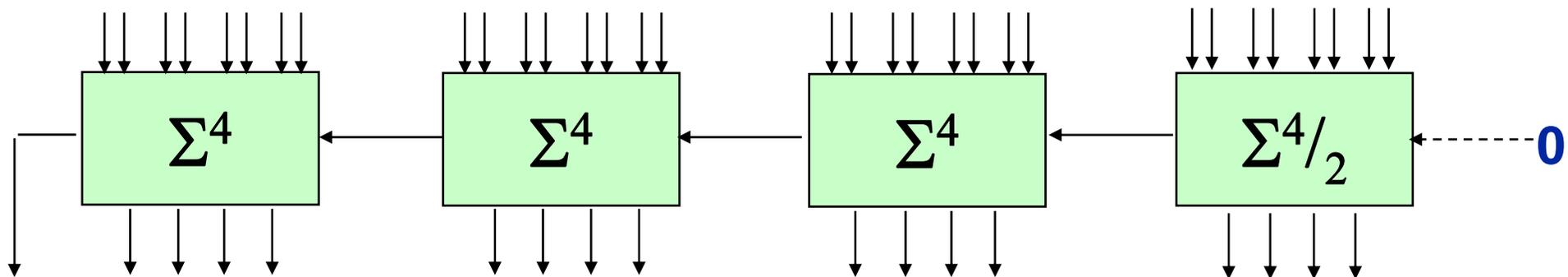
Nachteil: Der i -te Addierer muß auf das Carry des $(i-1)$ -ten Addierers warten.

Übertragungsschnellbestimmung

Beschleunigung des Addiernetzes durch zusätzliche Hardware zur schnellen Bestimmung des Übertrages (**carry-lookahead**):

Idee: Modularisierung, Einführung von weniger Schaltebenen

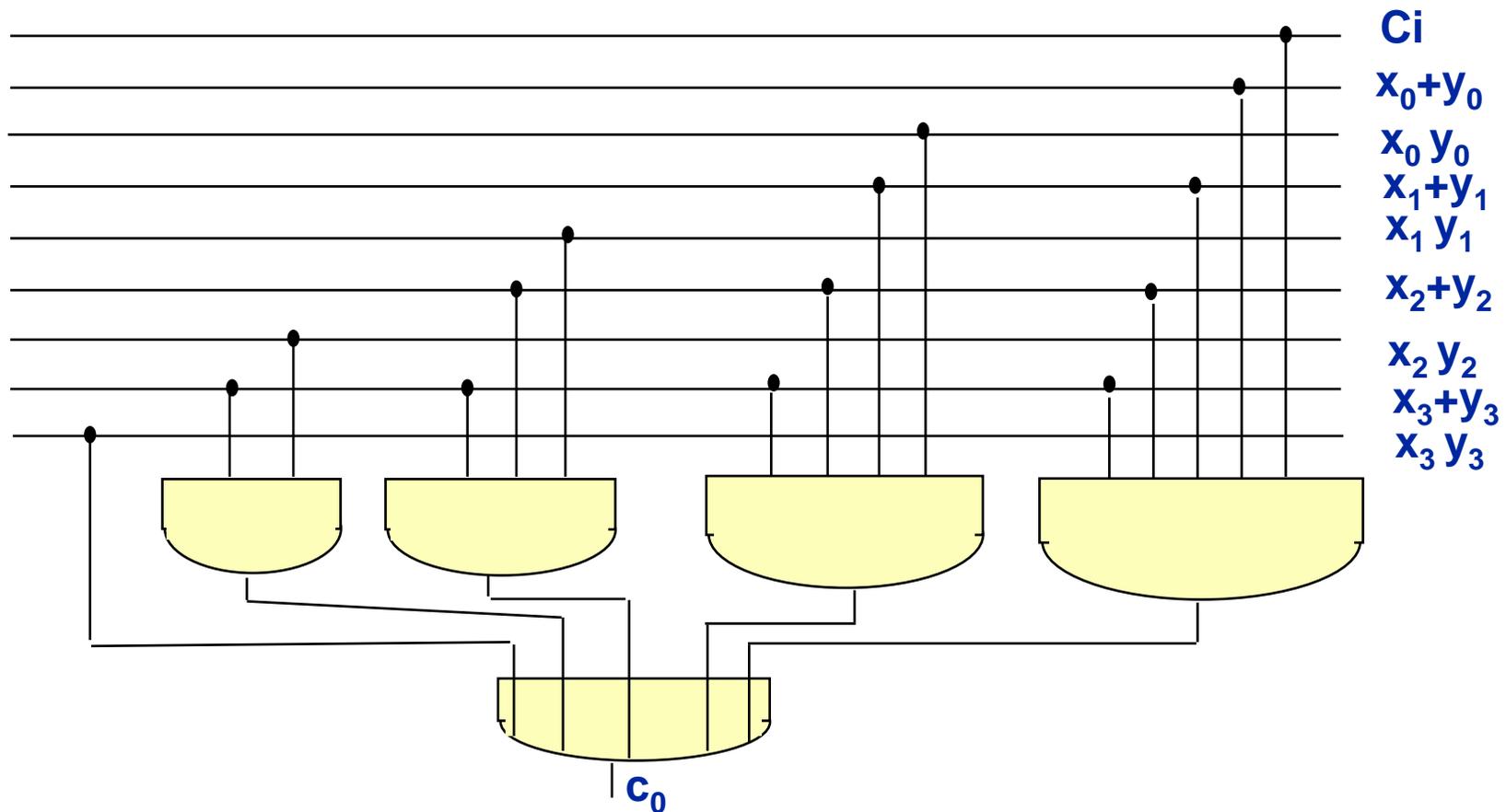
Bilde Bitgruppen der Größe $g = 4 \Rightarrow$ 16 Bit-Addierer hat 4 Schaltebenen



Jedes Modul Σ^4 hat 9 Eingänge und 5 Ausgänge.

Übertragsschnellschaltung (carry bypass, carry lookahead)

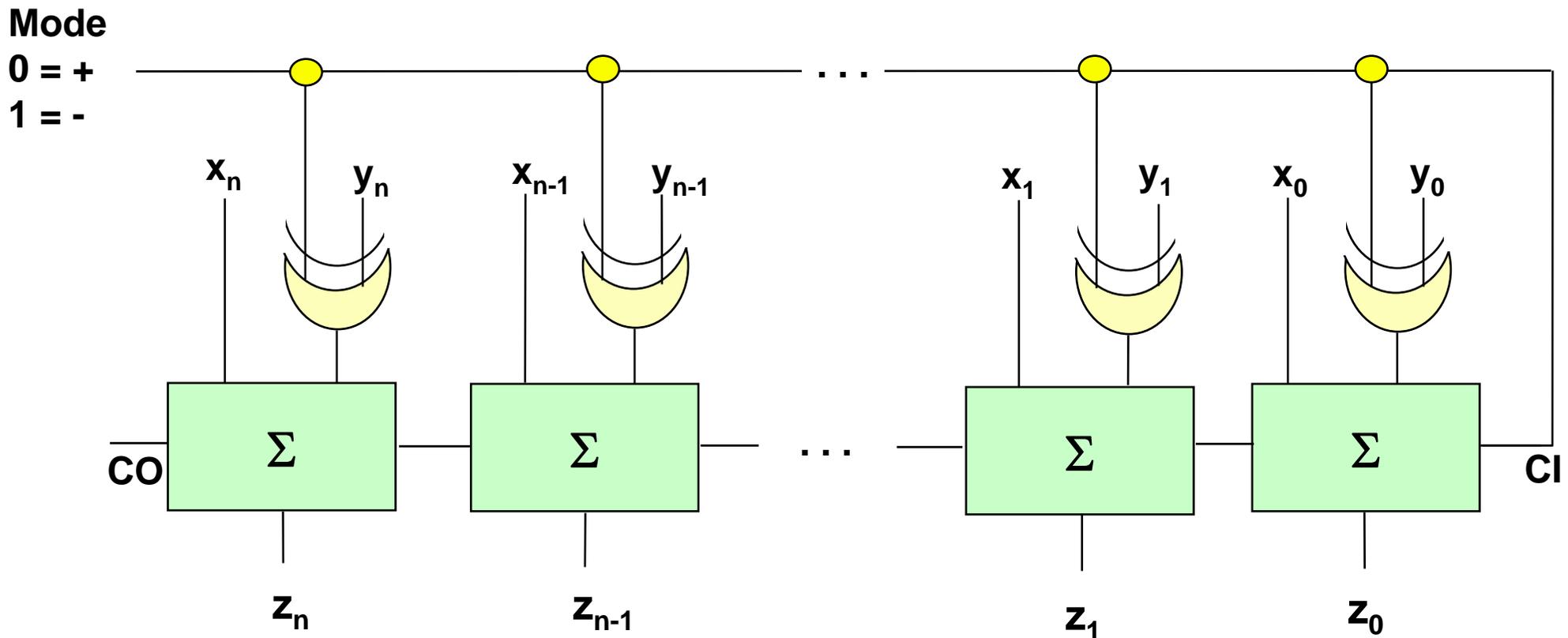
3-stufige Schaltung unter Verwendung mehrstelliger Gatter
(Summe von Produkten aus Summen oder Produkten):



Schaltkreis mit Volladdierern

Mit einem **zusätzlichen XOR-Glied pro Binärstelle** kann man mit Volladdierern **Zweierkomplementzahlen sowohl addieren als auch subtrahieren.**

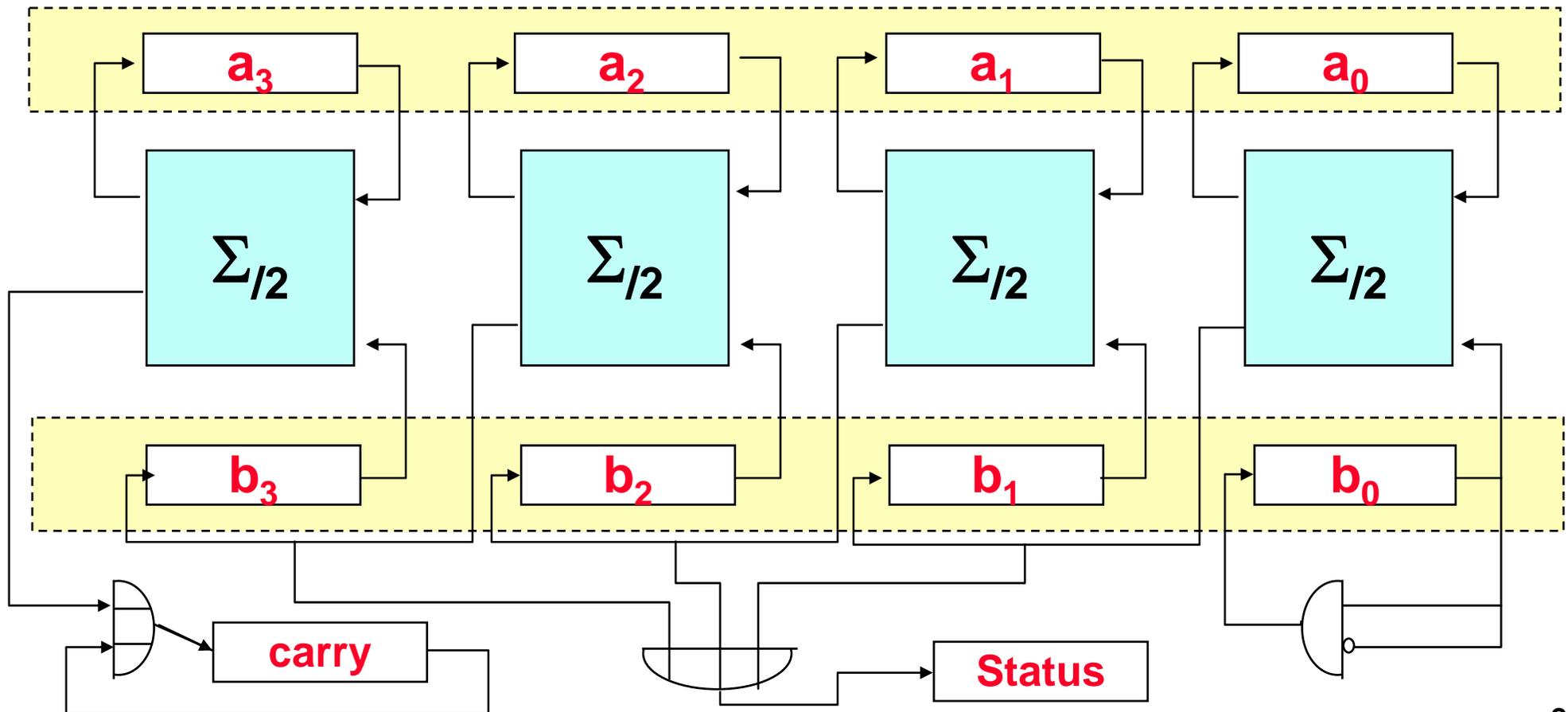
Für **Mode = 1** gelangt das bitweise Komplement von $y_n y_{n-1} \dots y_1 y_0$ an die Addierer und der 0-te Carry-Eingang erhält eine 1.



Das von Neumann-Addierwerk

getaktetes Paralleladdierwerk aus n parallel arbeitenden Halbaddierern und zwei n-Bit Registern a und b

Struktur des Addierwerks für n=4:

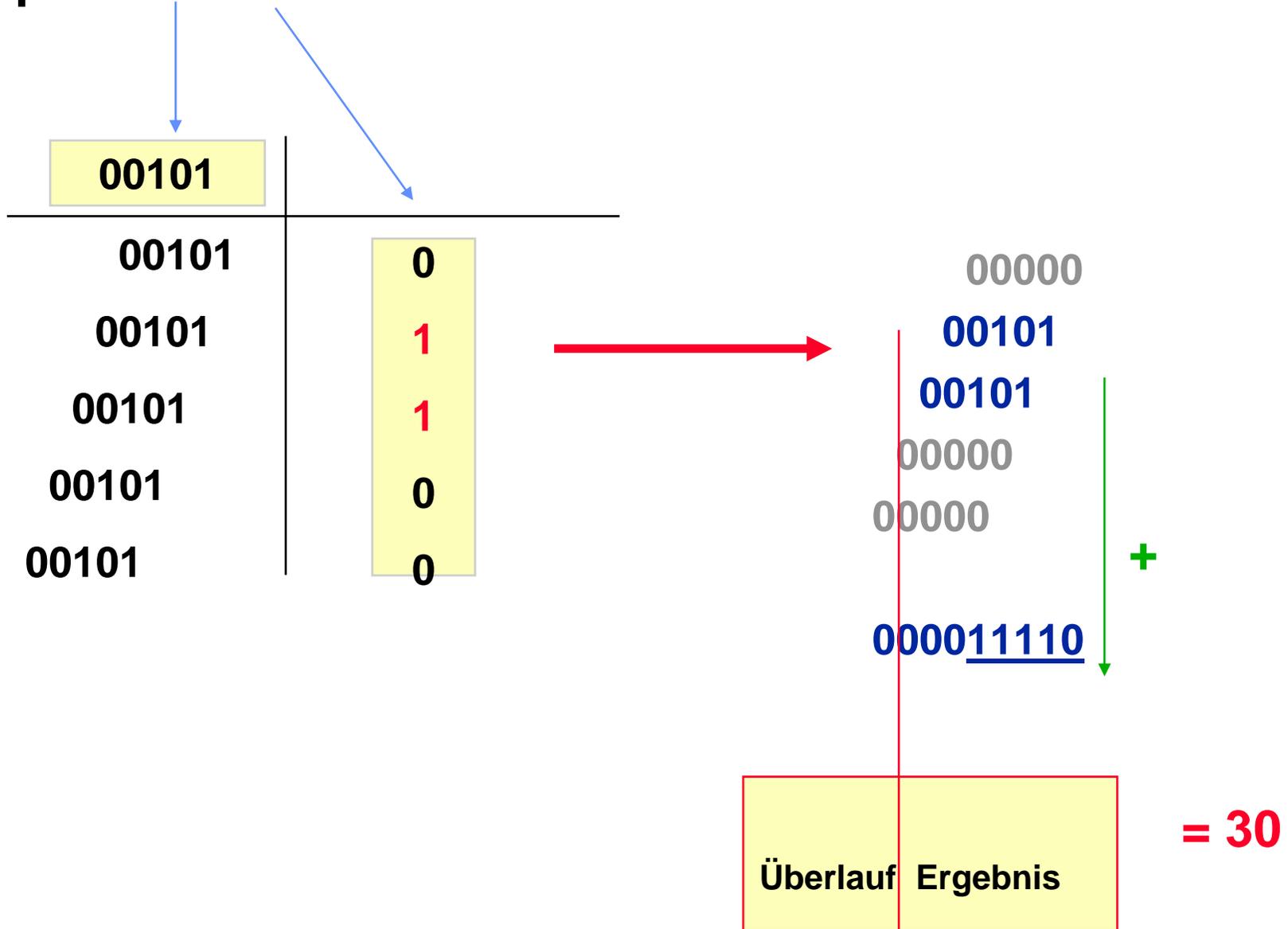


Arbeitsweise des Addierwerks

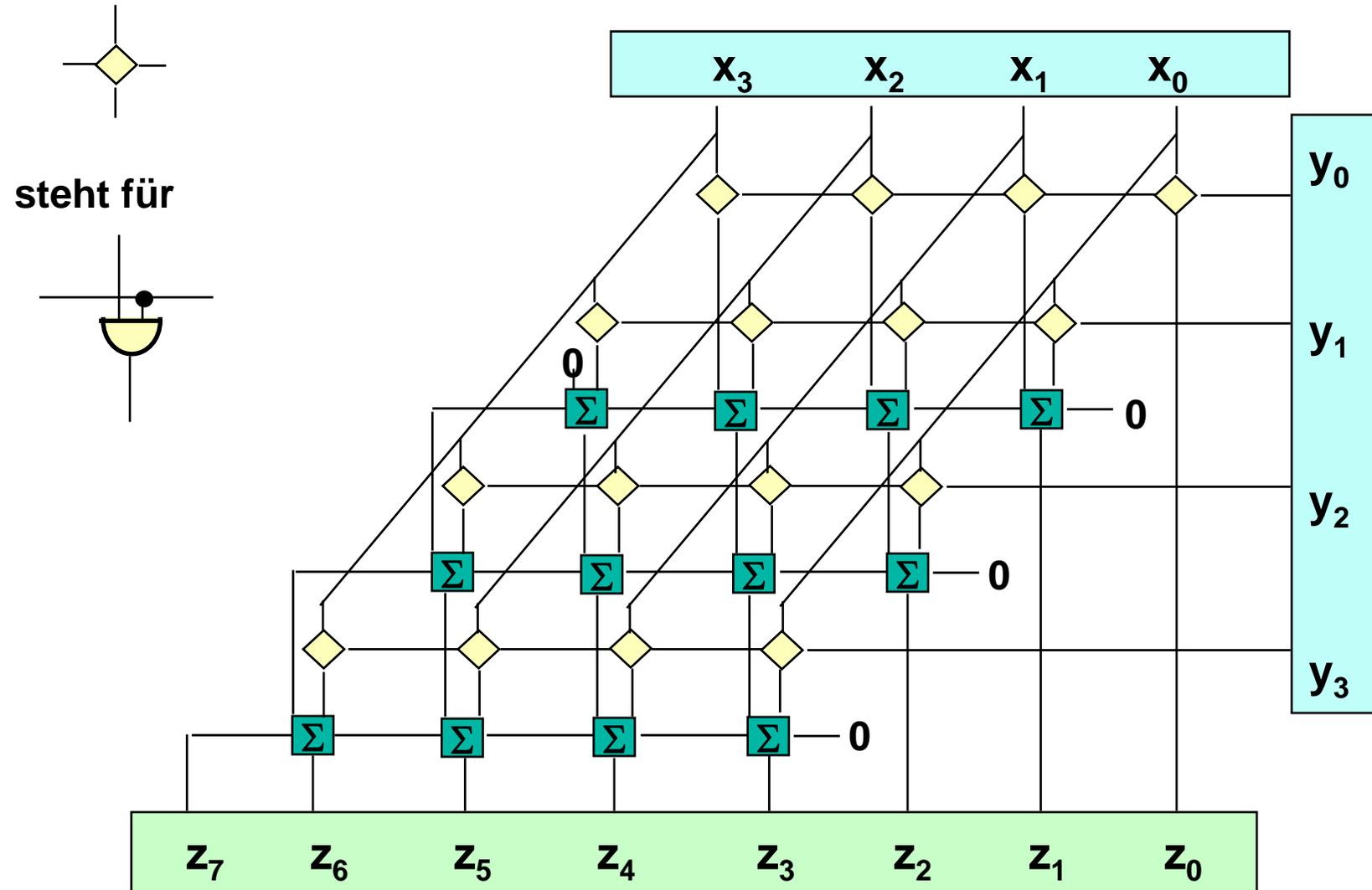
- Die Addition von zwei n-Bit Zahlen erfordert maximal **n Takte**.
- Zu Beginn enthalten die Register a und b die zu addierenden Zahlen.
- Der i-te Halbaddierer erhält die Registerzellen a_i und b_i als Eingänge.
- Der Summenausgang wird in die Registerzelle a_i zurückgeschrieben. Der Übertragsausgang wird in die Registerzelle b_{i+1} geleitet.
- Das **Register a speichert** demnach **Zwischensummen** und schließlich die Endsumme, während das **Register b** die vom Halbaddierer gebildeten **Überträge** speichert.
- Summen und Überträge werden in die Register zurückgeschrieben, bis das Statusregister 0 enthält.

Multiplizieren

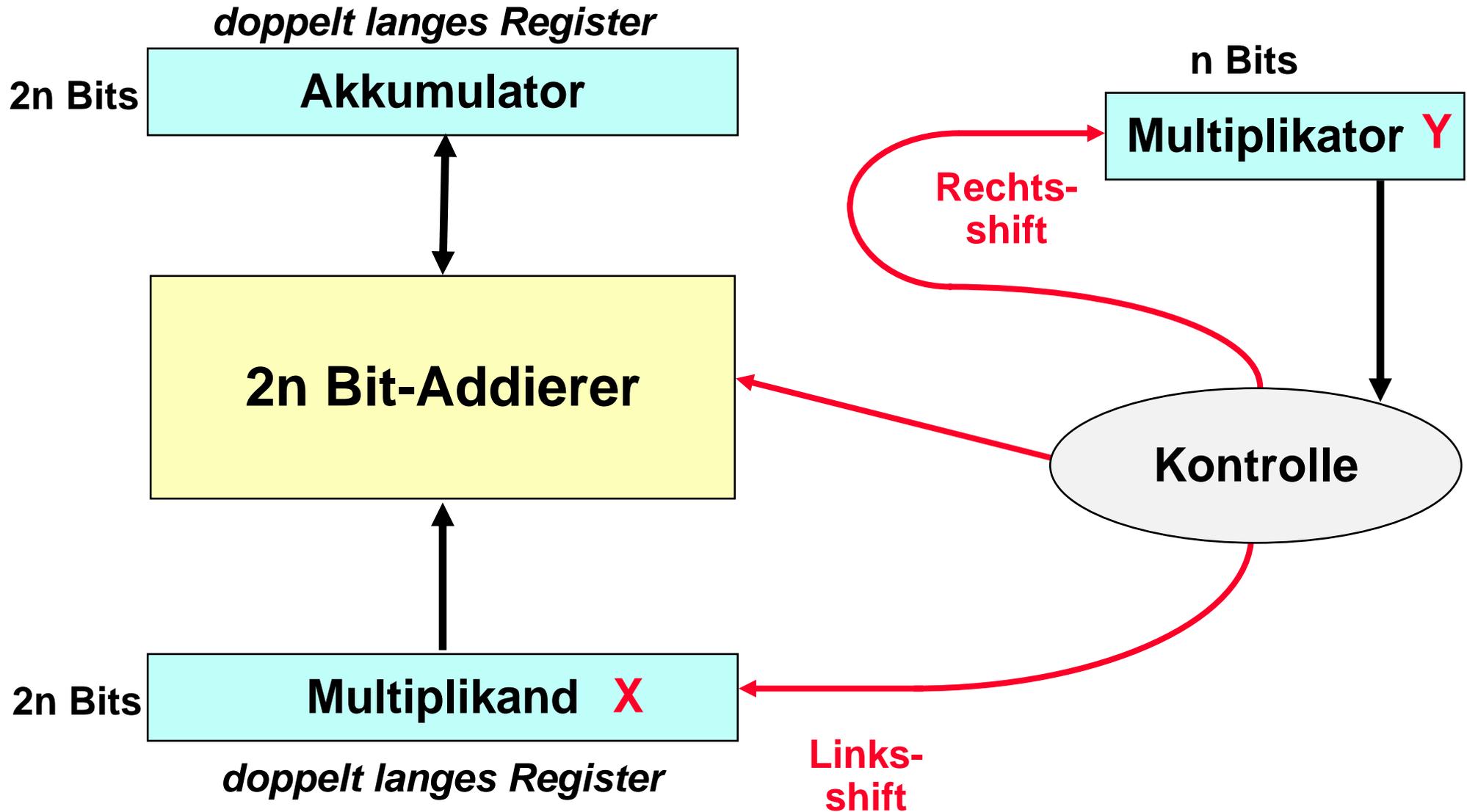
Beispiel **5 x 6:**



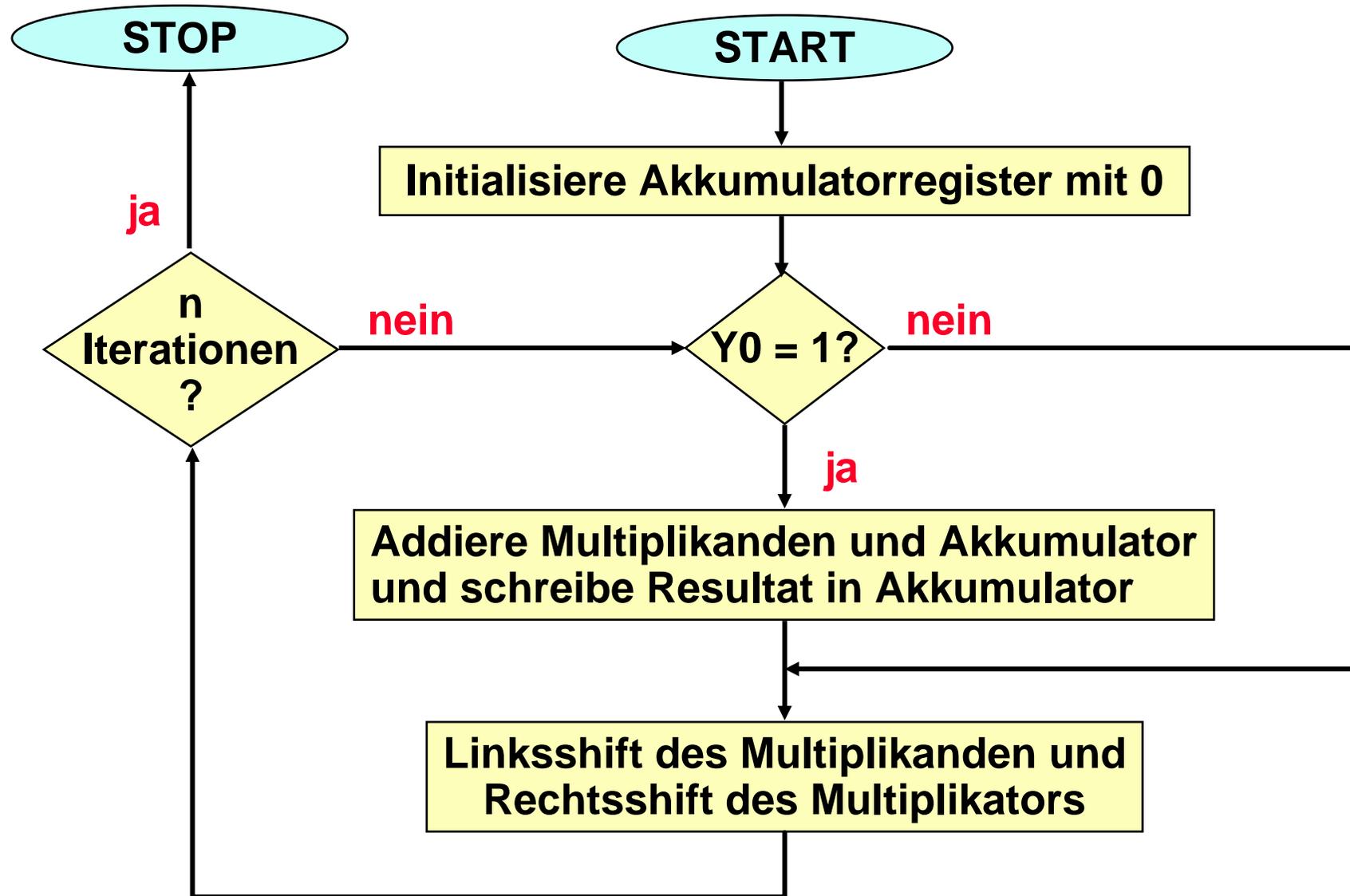
Barrel Shifter - Multiplikationswerk



Iterative Realisierung der Multiplikation



Ablaufdiagramm



1. Optimierung

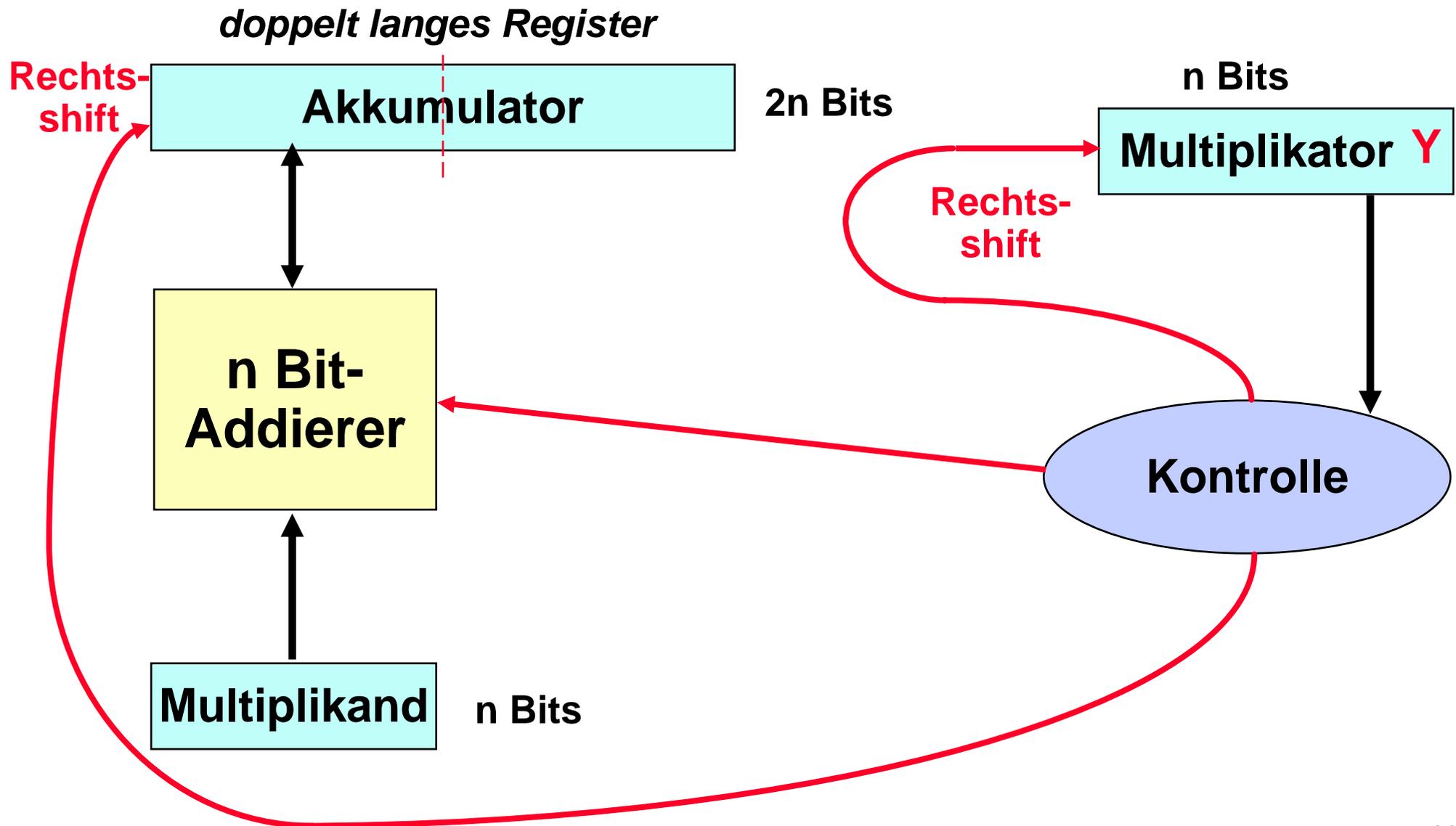
Beobachtungen:

- Die obere Hälfte des Multiplizierenregisters ist zu Beginn mit Nullen belegt.
- Beim Linksshift werden Nullen nachgeschoben, so dass sich die letzten Ziffern des Produktes nicht mehr ändern.

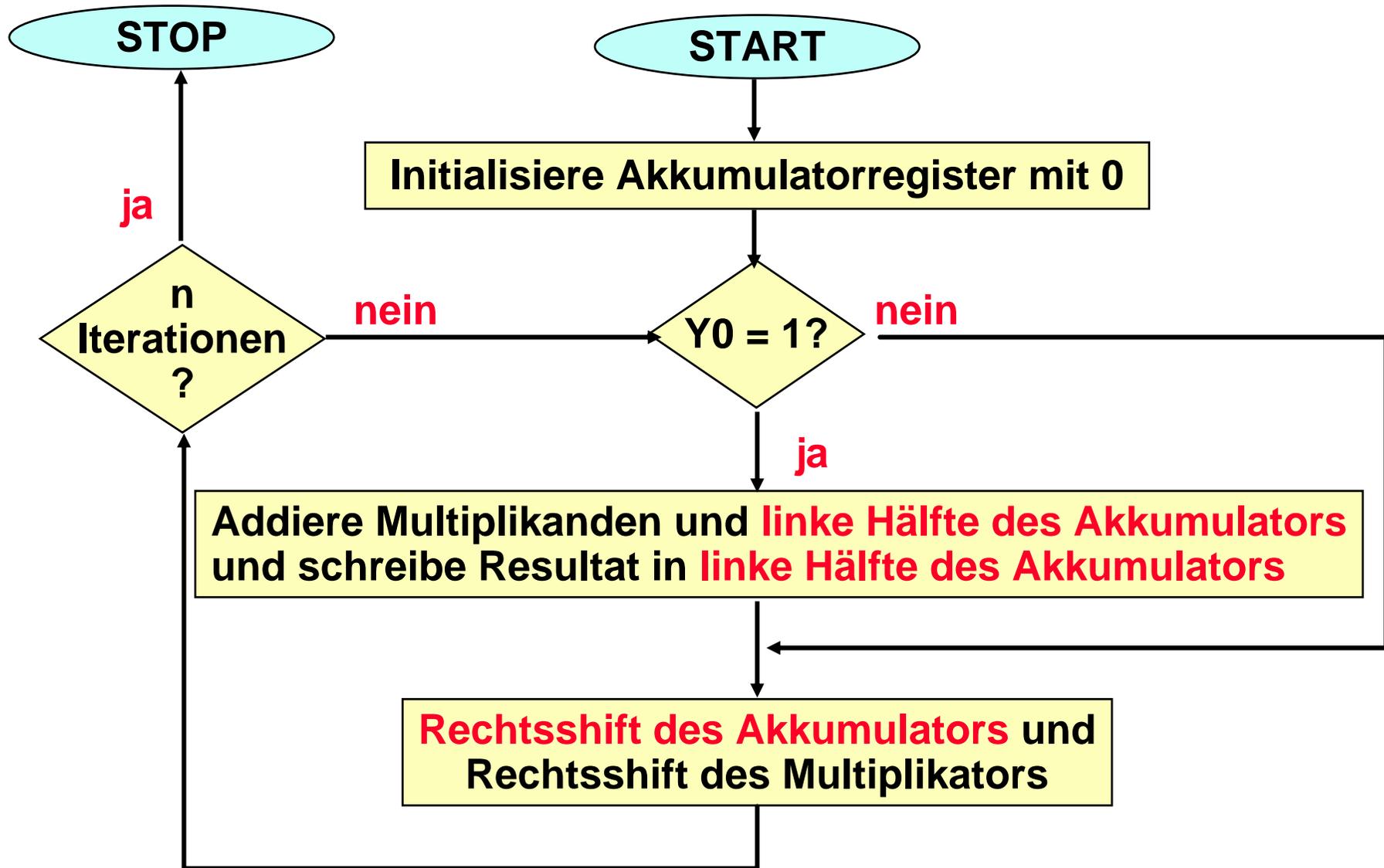
**=> Es genügen
ein n Bit-Addierer sowie
ein n Bit-Register für den Multiplizieren.**

**Statt den Multiplizieren nach links zu schieben,
kann man das Produkt im Akkumulator nach rechts
schieben.**

Modifiziertes Multiplizierwerk



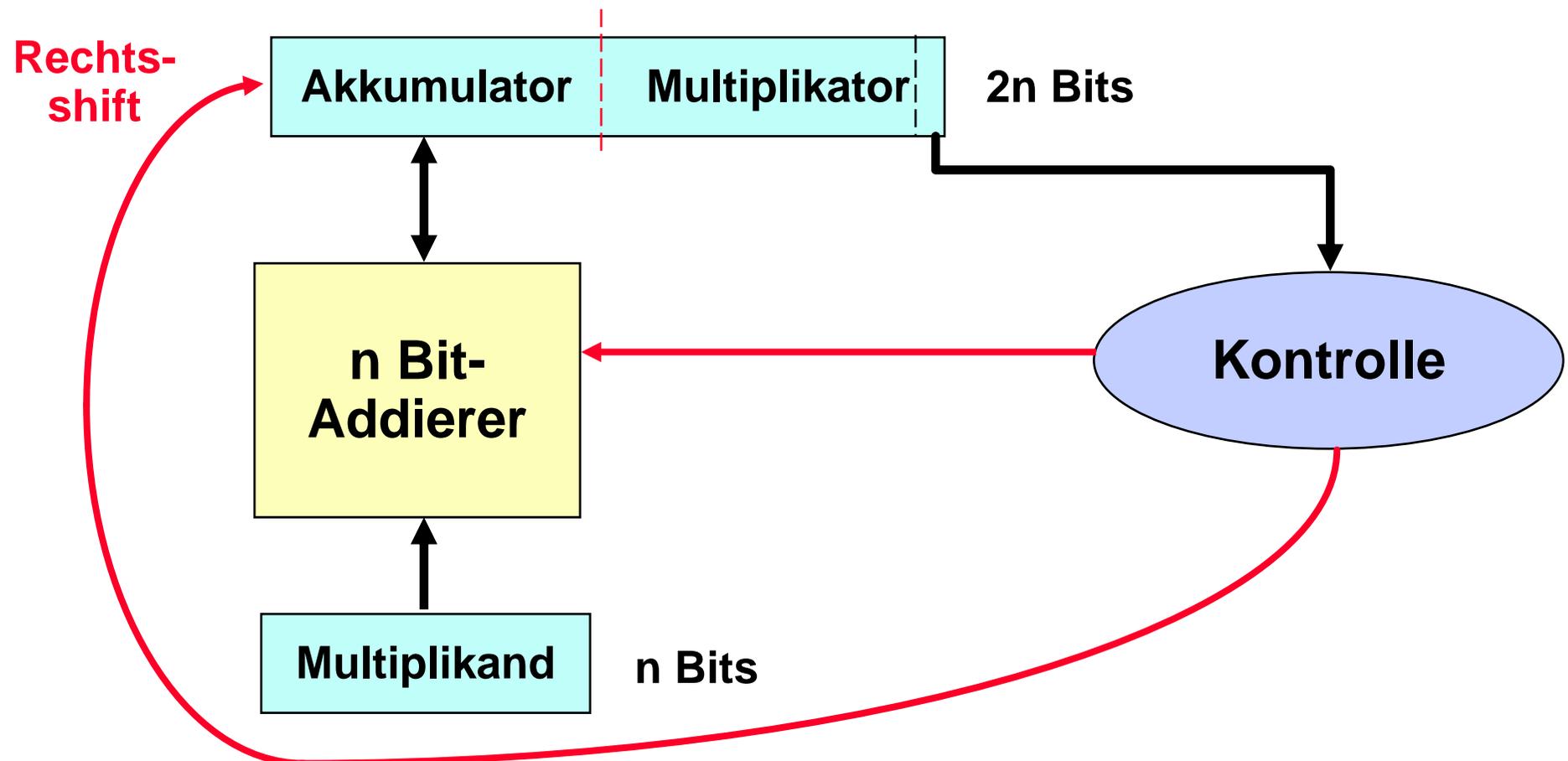
Modifiziertes Ablaufdiagramm



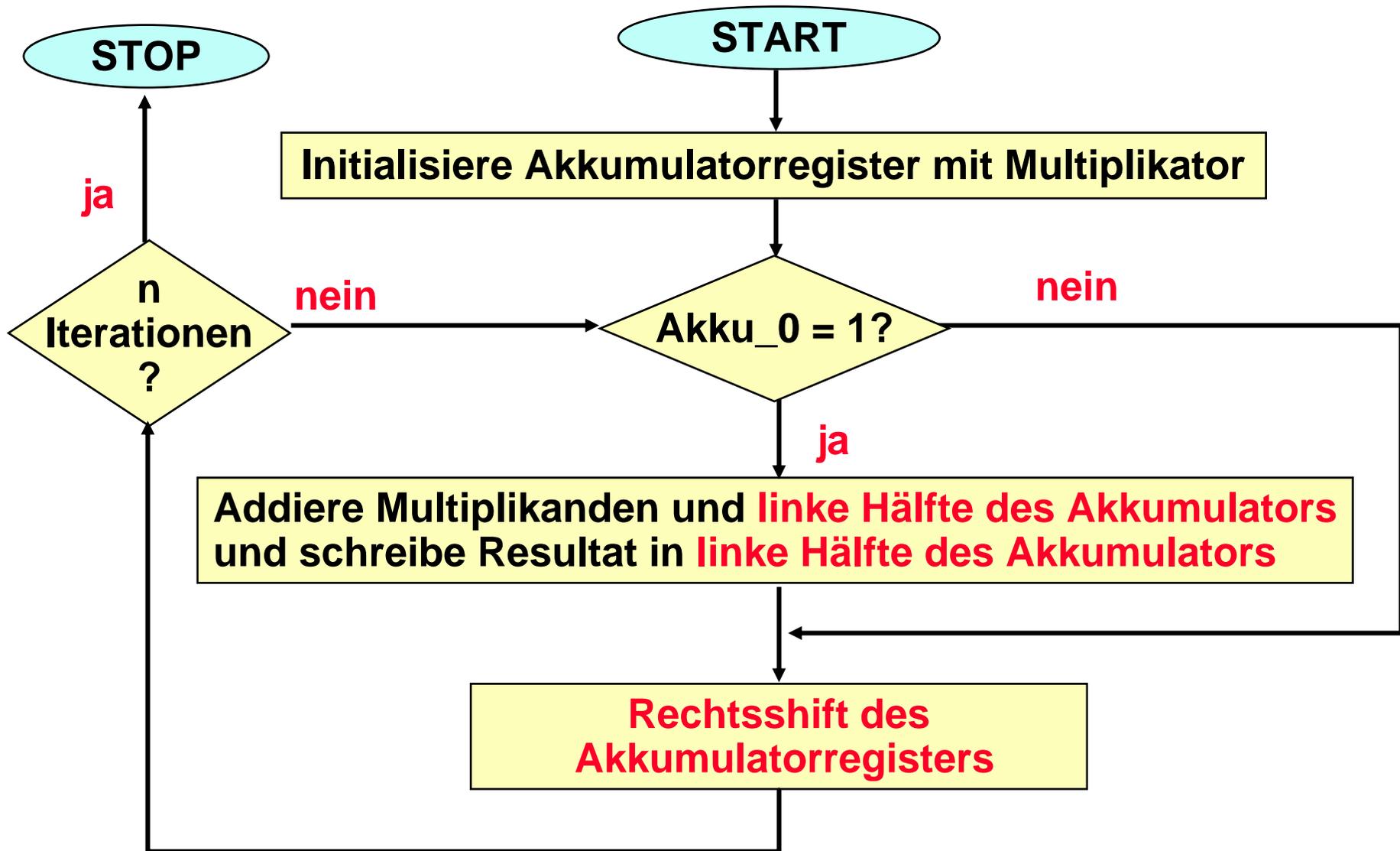
2. Optimierung

Beobachtung:

Die linke Hälfte des Akkumulators kann zur Speicherung des Multiplikators verwendet werden, da dieser in dem Maße nach rechts geschoben wird, wie sich das Produkt im Akkumulator nach rechts ausdehnt.



Endgültiges Ablaufdiagramm



Multiplikationsalgorithmus von Booth

- Verwendung von Addition und Subtraktion
- Einsparung von Additionen/Subtraktionen, wenn möglich
- funktioniert auch für Zweierkomplementzahlen

Idee: Multiplikation mit 2^i

 \cong Linksshift des Multiplikanden um i Bits und Addition

 Multiplikation mit $(\underbrace{11\dots1}_j)_2 = 2^j - 1$
j Bits

\cong Linksshift des Multiplikanden um j Bits, Addition und anschließende Subtraktion

Beispiel:

Booth:

$$\begin{aligned} &10101 * 11111 \\ &= 1010100000 \\ &- \quad \quad 10101 \\ &= 1010001011 \end{aligned}$$

Standard:

$$\begin{aligned} 10101 * 11111 &= 101010000 \\ &+ 10101000 \\ &+ 1010100 \\ &+ 101010 \\ &+ 10101 \\ \hline &= 1010001011 \end{aligned}$$

Multiplikationsalgorithmus von Booth 2

benachbarte
Multiplikatorbits

$Y_i Y_{i-1}$	Situation	Aktion
00	Folge von Nullen	arithm. Shift des Produktreg.
01	Ende einer Folge von Einsen	Addition Multiplikand und Produktreg. + arithm. Shift
10	Beginn einer Folge von Einsen	Subtraktion des Multiplikanden vom Produktreg. + arithm. Shift
11	Folge von Einsen	arithm. Shift des Produktreg.

<u>Akkumulator / Multiplikator</u>	<u>Aktion</u>	<u>Multiplikand</u>
0000 0000 1011 1111 <u>0</u>	Subtraktion	0110 0000
- <u>0110 0000</u>		
1010 0000	Shift (Vorzeichenbit nachschieben)	
1101 0000 0101 1111 <u>1</u>	Shift	
1110 1000 0010 1111 <u>1</u>	Shift	
1111 0100 0001 0111 <u>1</u>	Shift	
1111 1010 0000 1011 <u>1</u>	Shift	
1111 1101 0000 0101 <u>1</u>	Shift	
1111 1110 1000 0010 <u>1</u>	Addition	
+ <u>0110 0000</u>		
0101 1110	Shift	
0010 1111 0100 0001 <u>0</u>	Subtraktion	
- <u>0110 0000</u>		
1100 1111	Shift	
1110 0111 1010 0000 <u>1</u>		

Beispiel (Booth-Algorithmus)

Analyse:
 1 Addition &
 2 Subtraktionen
 statt 7 Additionen

Einsparungen vor allem bei langen Folgen von Einsen, im Mittel keine Verbesserung

Booth funktioniert für Zweierkomplementzahlen

- Betrachte

Multiplikator $a = \text{decode}_2(a_n \dots a_0) = -a_n 2^n + \sum_{i=0}^{n-1} a_i 2^i$ und Multiplikand b

- Booth betrachtet in jeder Iteration die Bits $a_i a_{i-1}$ mit folgender **Aktionstabelle**:

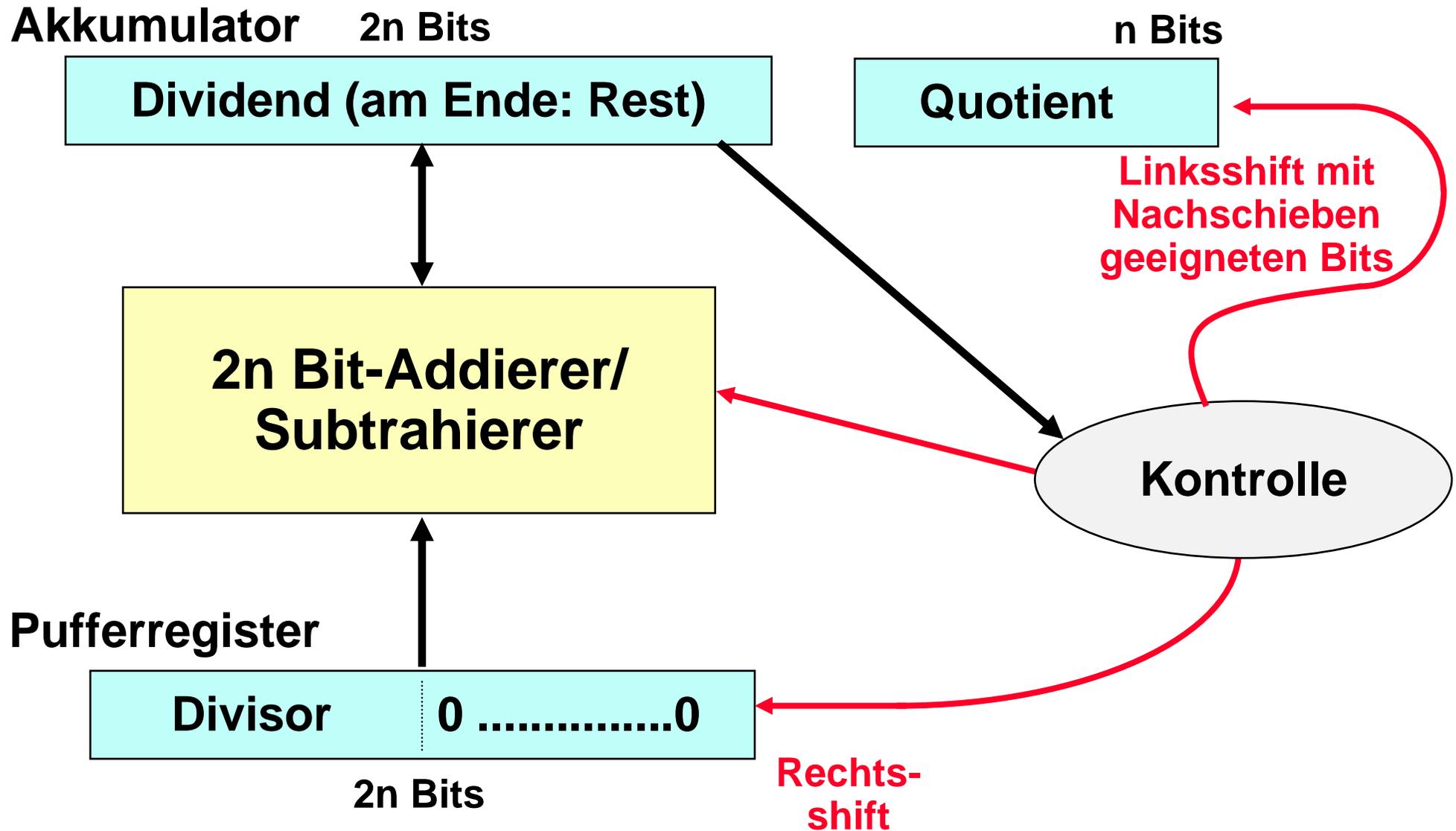
$a_i a_{i-1}$	Aktion
00	-
01	Addiere b
10	Subtrahiere b
11	-

=> **addiere immer $(a_{i-1} - a_i) * b$**

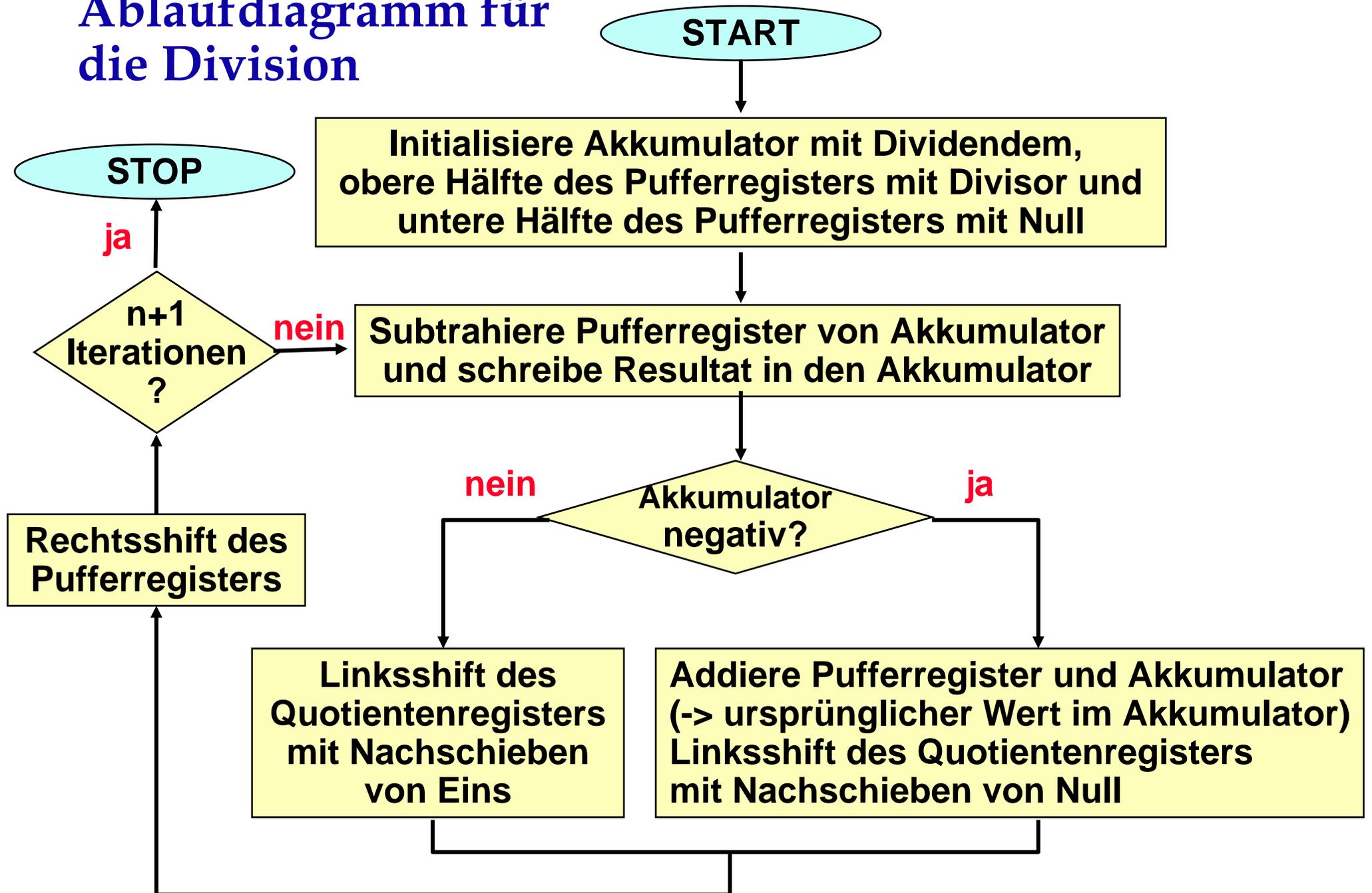
Der Rechtsshift des Zwischenproduktes entspricht einer Multiplikation mit 2 pro Iteration.

- Booth berechnet somit: $\sum_{i=0}^n (a_{i-1} - a_i) * b * 2^i = \dots = a * b$

Iterative Realisierung der Division

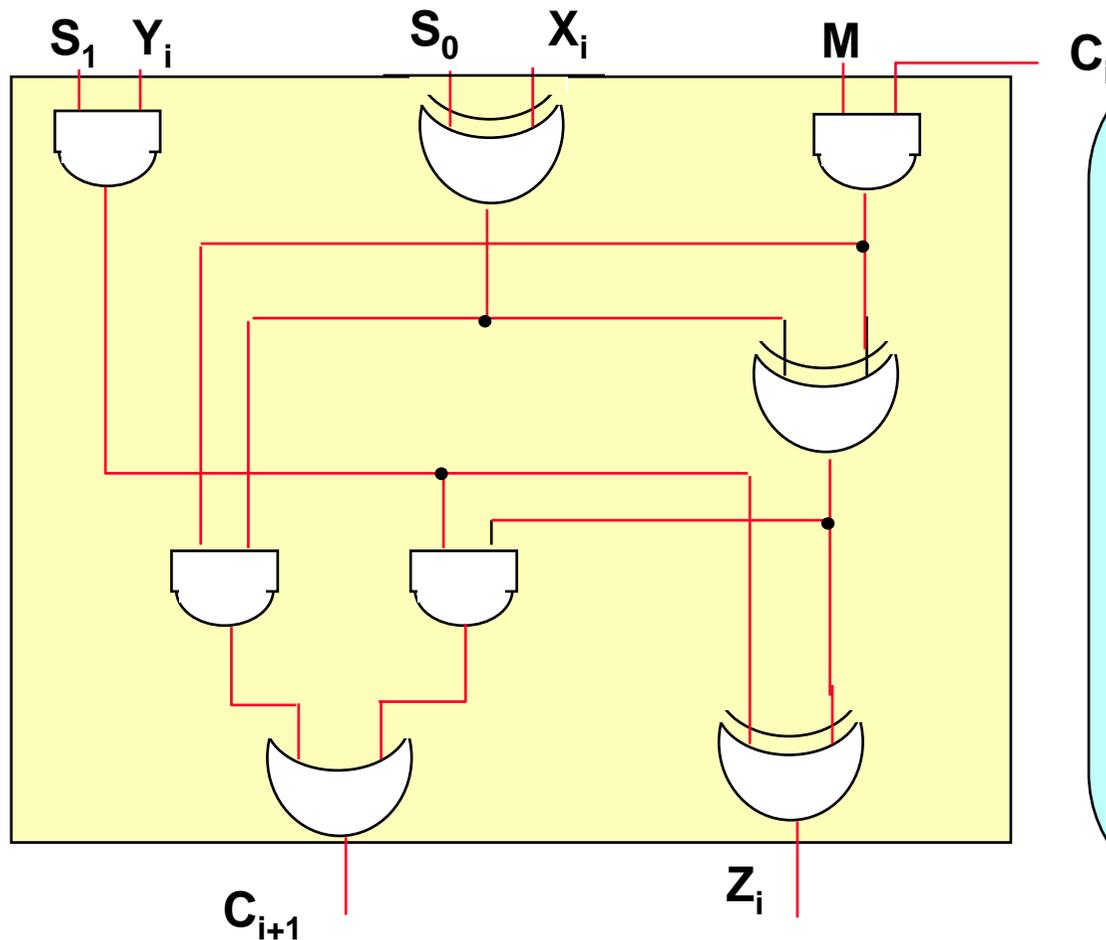


Ablaufdiagramm für die Division



Ein-Bit-ALU

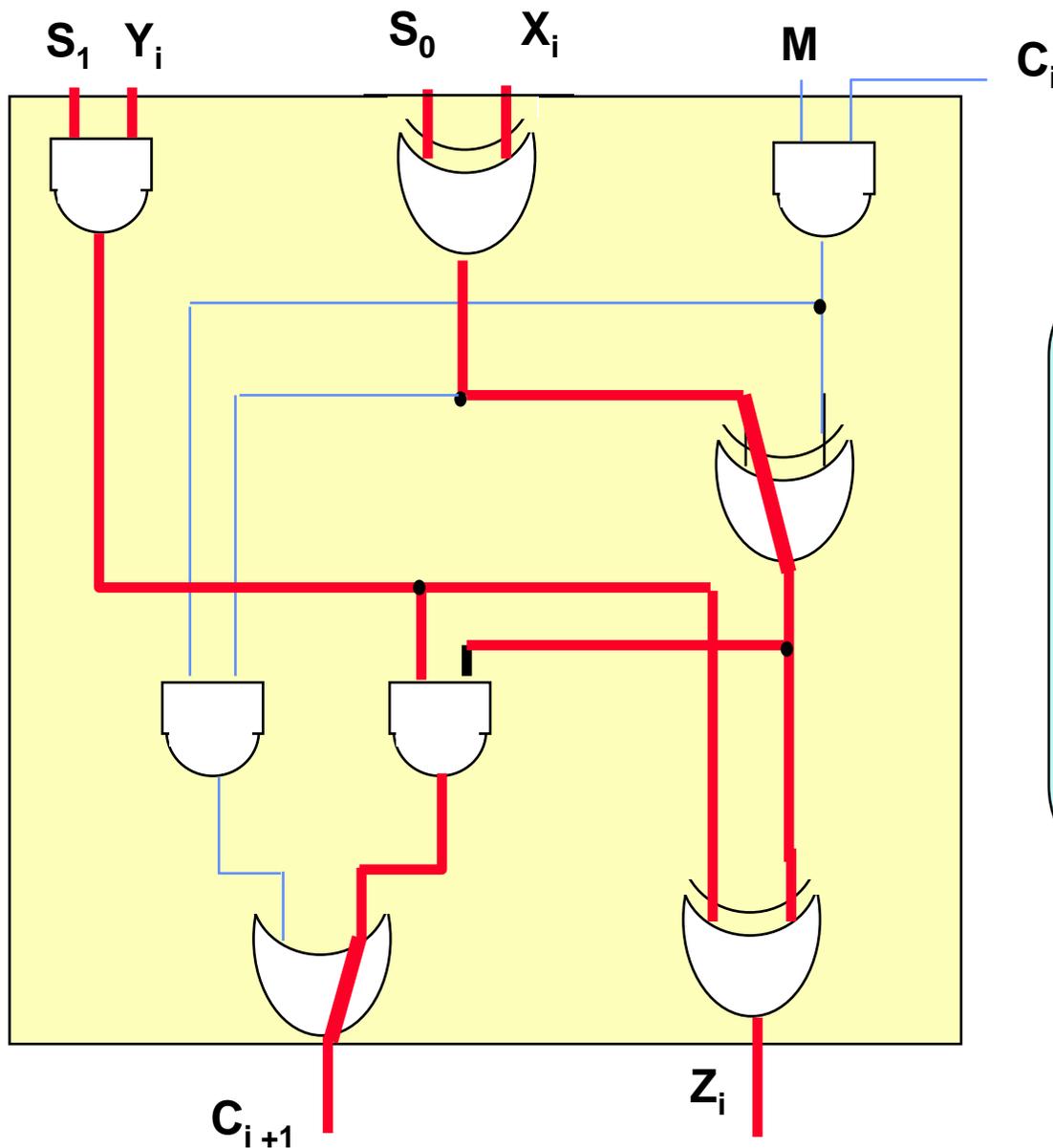
Eine 1-Bit-Alu ist eine Schaltung, die abhängig von der Stellung von Schaltern die grundlegenden arithmetischen und logischen Operationen durchführen kann. Durch die Berücksichtigung eines Carry-Bits kann sie mit anderen zu einer Mehr-Bit-Alu ergänzt werden.



S_0 , S_1 und M bestimmen die Funktion, die die ALU berechnet.

- logische Funktionen : $M=0$
- arithmetische Funktionen:
 $M=1$
- Falls $M=1$ muß man noch unterscheiden zwischen $C_0=0$ und $C_0=1$.

Ein-Bit-ALU

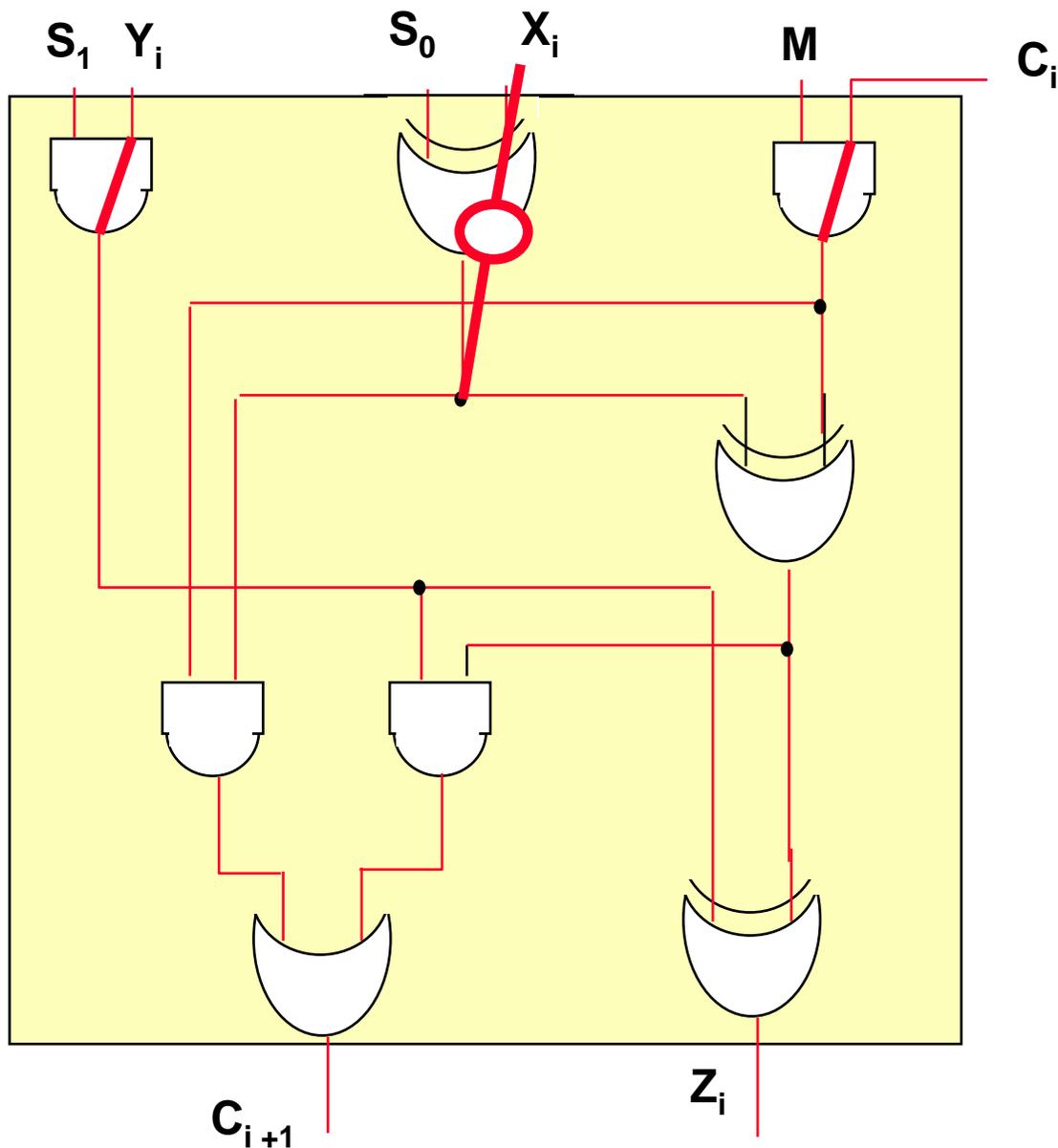


Für $M=0$ ist die ALU-Schaltung äquivalent zur hier gezeichneten.

Für $S_1 = 1, S_0 = 0$ gilt z.B.

$$Z_i = X_i \oplus Y_i$$

Ein-Bit-ALU im arithmetischen Mode



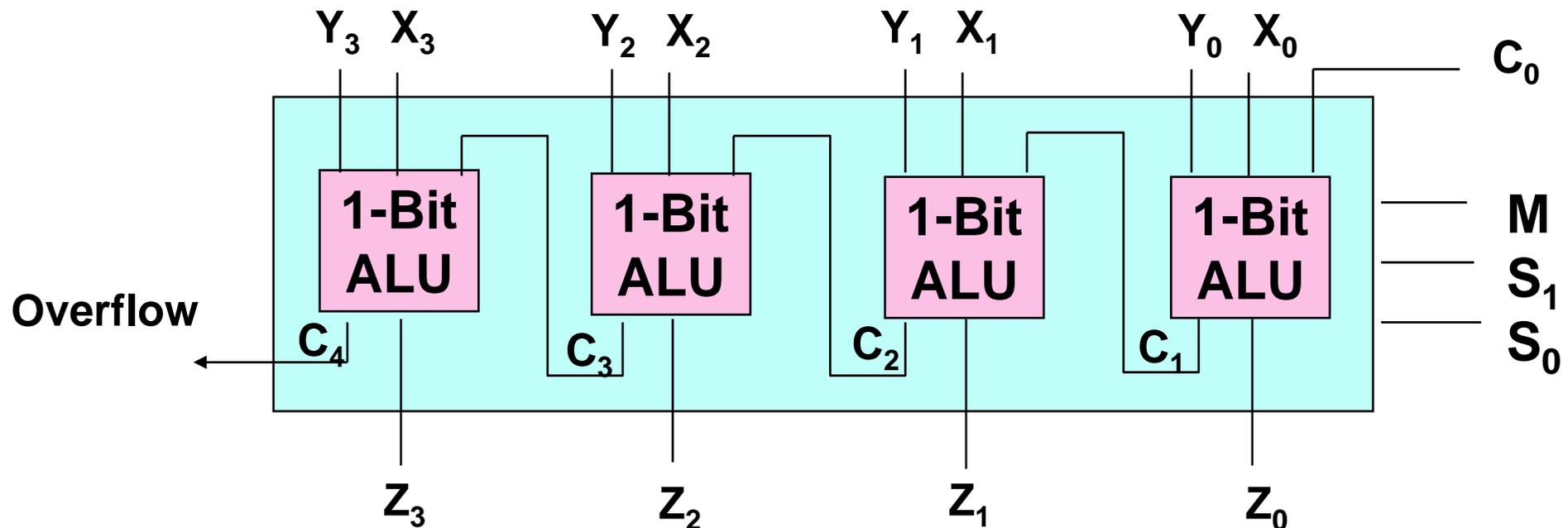
Für $M=1$ ist die ALU im arithmetischen Modus.

Für $S_1 = 1$, $S_0 = 1$ gilt z.B.

$$Z_i = Y_i - X_i$$

Eine vollständige 4-Bit ALU

S_0 , S_1 und M bestimmen die Funktion, die die ALU berechnen soll. Sie werden zu jedem Glied durchgezogen.



Schematische Darstellung der ALU

Die Alu führt die arithmetischen und logischen Berechnungen durch. Im **Mode** wird die Funktion ausgewählt, im X und Y-Register liegen die Argumente, im Z-Register liegt am Ende der Wert. Das **Flag-Register** zeigt besondere Bedingungen an, z.B. **Overflow**, **Ergebnis negativ**, **Ergebnis 0**, etc.

