HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Technische Grundlagen der Informatik

Schaltnetze

Prof. Dr.-Ing. Benjamin Weinert

Gliederung

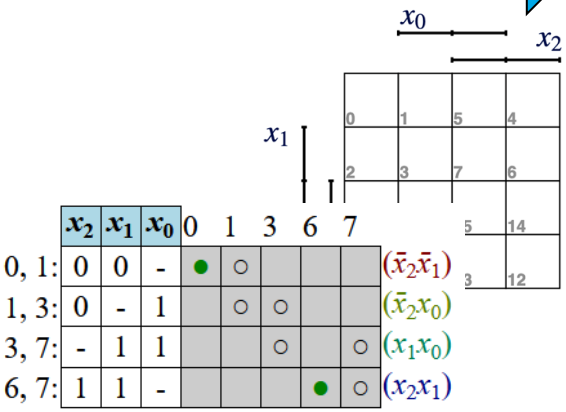
- Multiplexer / Demultiplexer
- Arithmetische Schaltungen
  - Addition
  - Inkrement
  - Subtraktion
  - Multiplikation

Motivation

Modellieren

Index i	a	b	c	y
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

Minimieren



$\overline{x_1}\,\overline{x_2}\,\overline{x_3} \vee x_1\,\overline{x_2}\,\overline{x_3} \vee x_1\,\overline{x_2}\,x_3 \vee \overline{x_1}\,\overline{x_2}\,x_3 \vee \overline{x_1}\,x_2\,\overline{x_3} \vee x_1\,x_2\,\overline{x_3}$

$= \overline{x_1}\,\overline{x_2}\,\overline{x_3} \vee x_1\,\overline{x_2}\,\overline{x_3} \vee x_1\,\overline{x_2}\,x_3 \vee \overline{x_1}\,\overline{x_2}\,x_3 \vee x_2\,\overline{x_3}$

$= \overline{x_1}\,\overline{x_2}\,\overline{x_3} \vee x_1\,\overline{x_2}\,\overline{x_3} \vee \overline{x_2}\,x_3 \vee x_2\,\overline{x_3}$

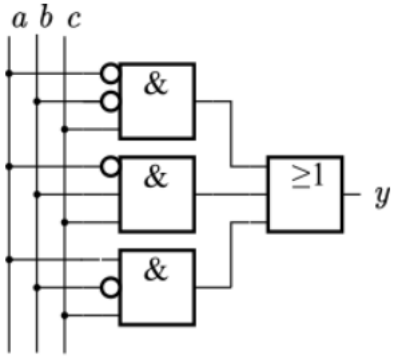
$= \overline{x_2}\,\overline{x_3} \vee \overline{x_2}\,x_3 \vee x_2\,\overline{x_3}$

$= \overline{x_2}\,\overline{x_3} \vee \overline{x_2}\,x_3 \vee \overline{x_2}\,\overline{x_3} \vee x_2\,\overline{x_3}$

$= \overline{x_2} \vee \overline{x_2}\,\overline{x_3} \vee x_2\,\overline{x_3}$

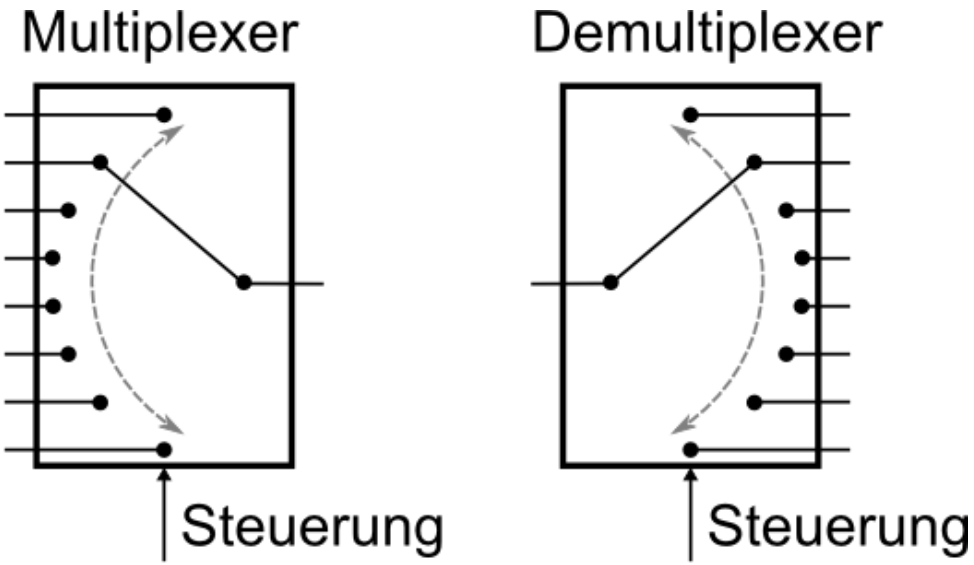
$= \overline{x_2} \vee \overline{x_3}$

Synthetisieren



Multiplexer/Demultiplexer

- Ein Multiplexer (auch "Mux") schaltet von vielen Eingängen auf einen Ausgang
- Ein Demultiplexer (auch "Demux") schaltet von einem Eingang auf viele Ausgänge

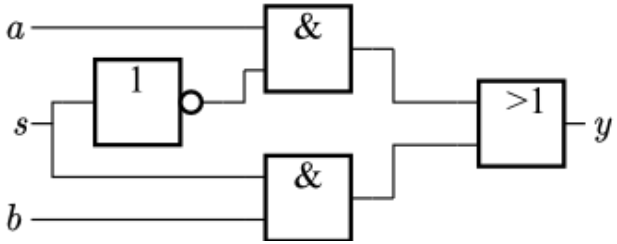


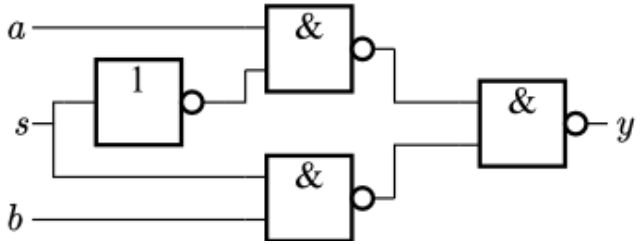
Multiplexer

- Für einen 2-zu-1 Multiplexer (auch "2:1 Mux") gilt:
  - Wenn das Steuersignal **s=1**,
    - dann ist y=b,
    - ansonsten y=a
- Diese entspricht einer if-then-else-Anweisung

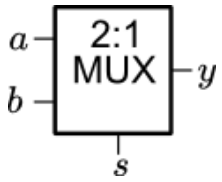
```
if (s) then y = b; else y = a;
```

- Ein 2-zu-1 Multiplexer kann leicht aus mehreren Logikgattern aufgebaut werden:



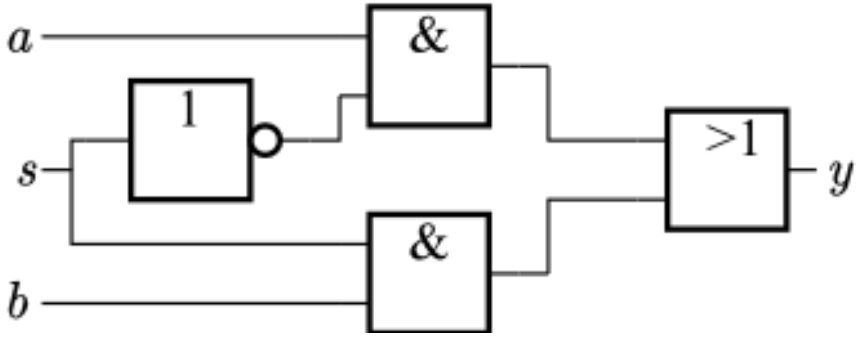


- Da Multiplexer häufig Verwendung finden, gibt es ein spezielles Symbol



Wahrheitstabelle 2-zu-1 Mux

Index i	a	b	s	y
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1



Multiplexer

- 2-zu-1-Multiplexer mit n=2 Dateneingängen

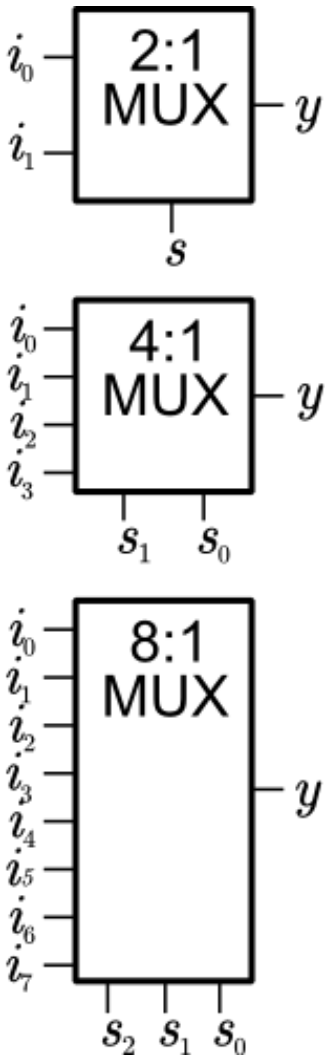
$$y = \bar{s}_0 i_0 \vee s_0 i_1$$

- 4-zu-1-Multiplexer mit n=4 Dateneingängen

$$y = \bar{s}_1 \bar{s}_0 i_0 \vee \bar{s}_1 s_0 i_1 \vee s_1 \bar{s}_0 i_2 \vee s_1 s_0 i_3$$

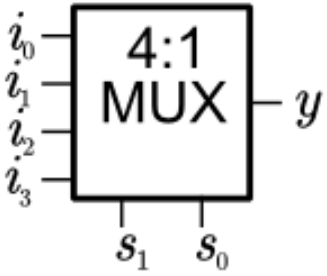
- 8-zu-1-Multiplexer mit n=8 Dateneingängen

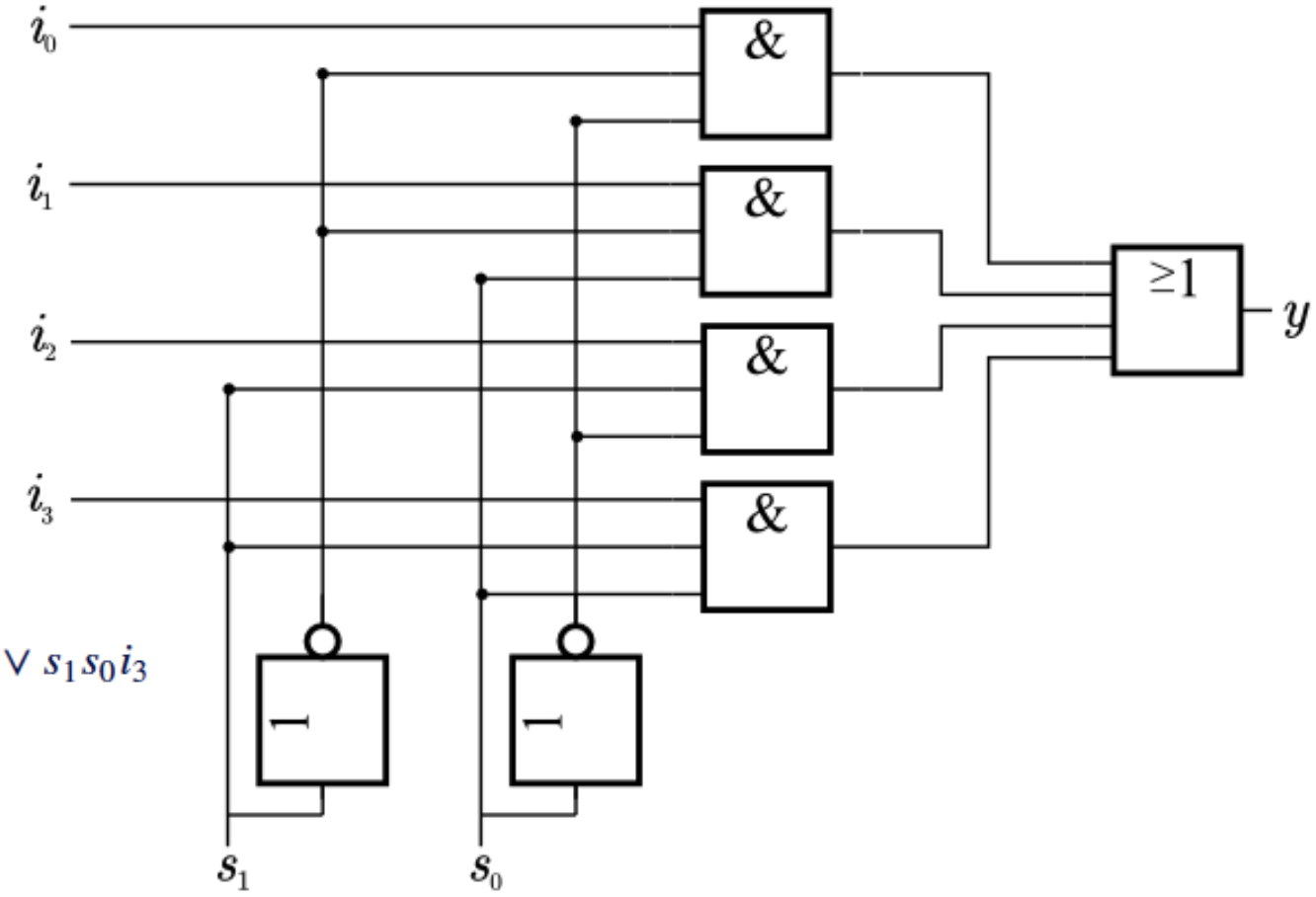
$$y = \bar{s}_2 \bar{s}_1 \bar{s}_0 i_0 \vee \bar{s}_2 \bar{s}_1 s_0 i_1 \vee \bar{s}_2 s_1 \bar{s}_0 i_2 \vee \bar{s}_2 s_1 s_0 i_3$$
$$s_2 \bar{s}_1 \bar{s}_0 i_4 \vee s_2 \bar{s}_1 s_0 i_5 \vee s_2 s_1 \bar{s}_0 i_6 \vee s_2 s_1 s_0 i_7$$



Multiplexer - Realisierung

- 4:1 Mux als zweistufige Logik

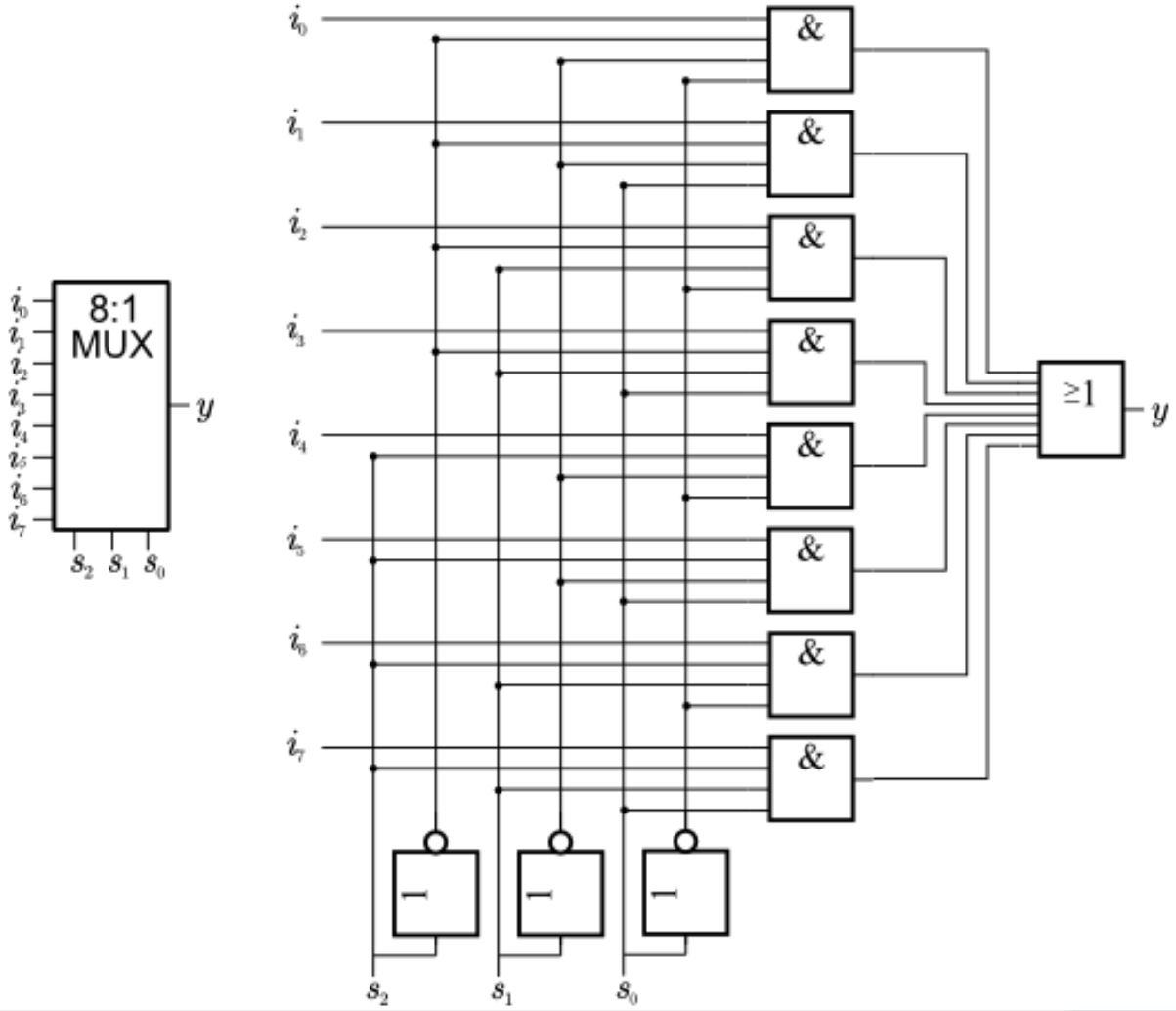




$$y = \bar{s}_1 \bar{s}_0 i_0 \vee \bar{s}_1 s_0 i_1 \vee s_1 \bar{s}_0 i_2 \vee s_1 s_0 i_3$$

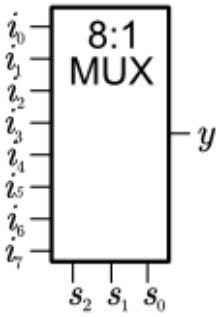
Multiplexer - Realisierung

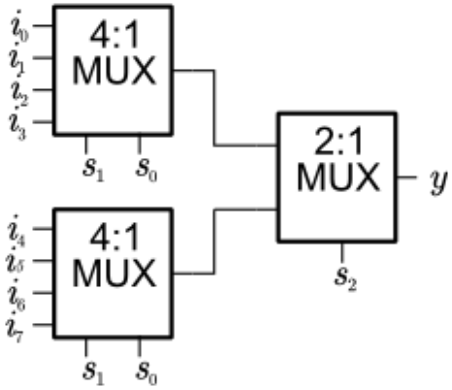
- Ein 8:1 Mux als zweistufige Logik

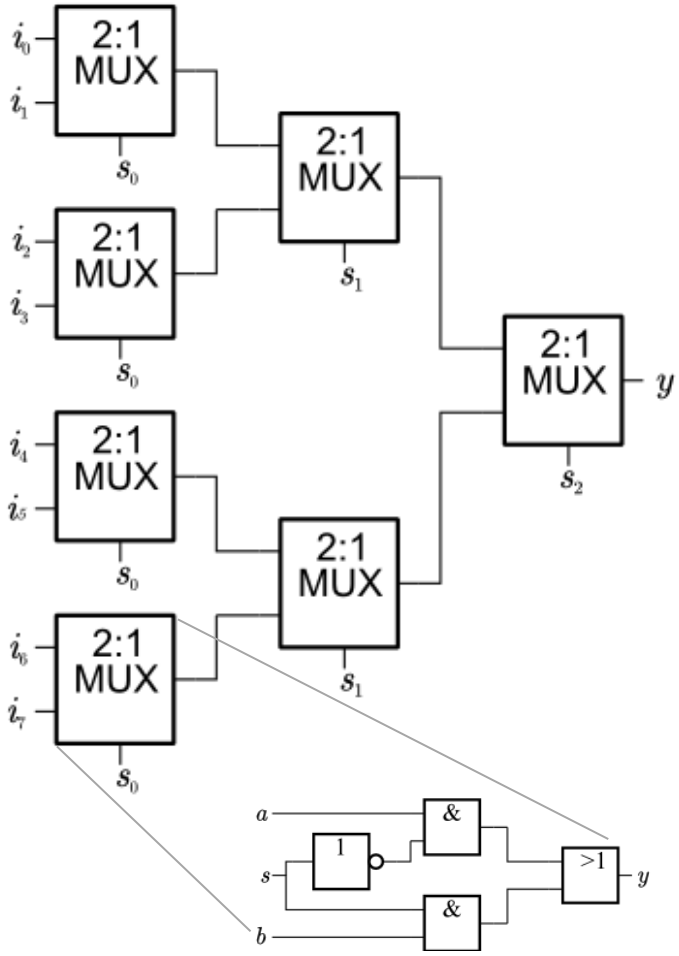


Kaskadierung von Multiplexern

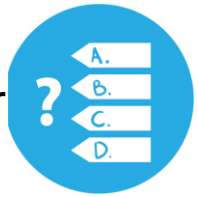
- Große Multiplexer können durch Kaskadierung von kleinen Multiplexern aufgebaut werden





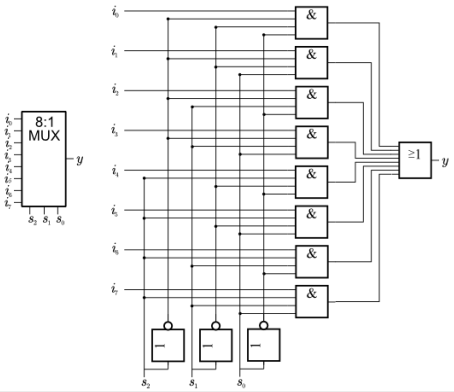


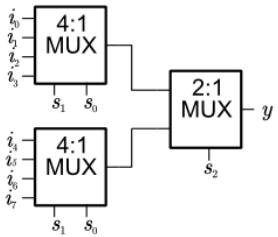
- Was ist sinnvoller
  - Kaskadierung oder
  - zweistufige Logik?

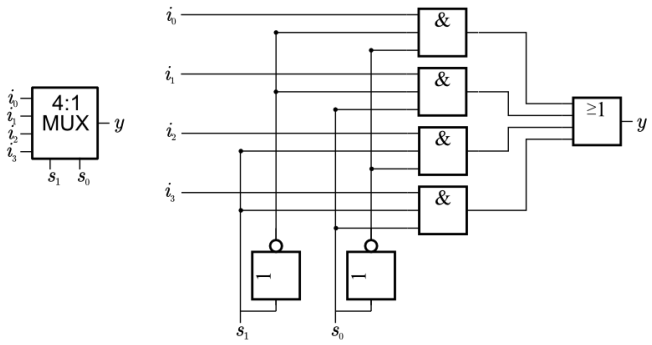


Kaskadierung von Multiplexern

- Kaskadierung oder zweistufige Logik ist abhängig vom Optimierungsziel
  - Bei zweistufiger Logik entstehen kleinere Gatterlaufzeiten
  - Eine Kaskadierung benötigt insgesamt weniger Eingänge
    - kann daher auf kleinerer Chipfläche realisiert werden
- Beispiel
  - zweistufige Logik benötigt UND-Gatter mit je $\log_2(n) + 1$ Eingängen
  - kaskadierte Lösung benötigt mehr UND-Gatter, diese haben aber **weniger Eingänge**
  - 8:1-Mux
    - zweistufig: 8 Gatter mit je *($log_2(8) + 1 = 4$)* vier Eingängen = 32 Eingänge
    - kaskadiert: 2 * 4 Gatter mit je 3 Eingängen + 2 Gatter mit 2 Eingängen = 28 Eingänge
    - Flächenaufwand proportional zur Anzahl der Eingänge
      ➔ kaskadierte Lösung hat geringeren Flächenverbrauch







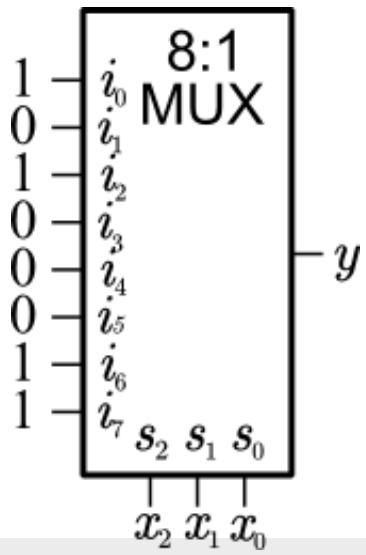
Multiplexer als universelle Logikbausteine

- Multiplexer sind nicht nur zur Steuerung von Datenflüssen, sondern auch zur Realisierung logischer Funktionen verwendbar
- n-zu-1-Multiplexer können jede Funktion mit $\log_2(n)$-Variablen implementieren
  - Die **Variablen** $x_{k-1}, ..., x_0$ werden als Steuersignale $s_{k-1}, ..., s_0$ verwendet
  - Die Dateneingänge $i_{n-1}, ..., i_0$ werden auf 0 oder 1 gelegt
- Beispiel $y = m_0 \vee m_2 \vee m_6 \vee m_7$

$$\bar{x}_2\bar{x}_1\bar{x}_0 \vee \bar{x}_2 x_1 \bar{x}_0 \vee x_2 x_1 \bar{x}_0 \vee x_2 x_1 x_0$$

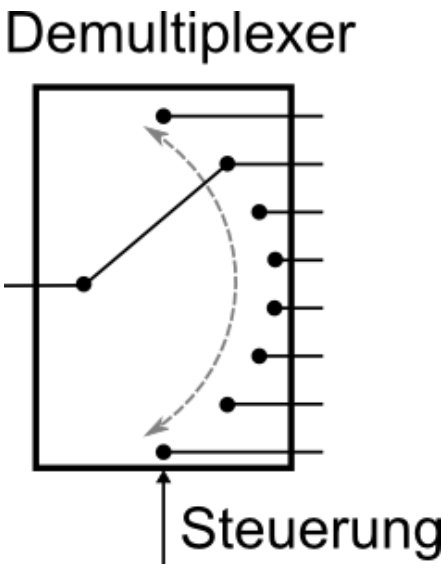
Wahrheitstafel:

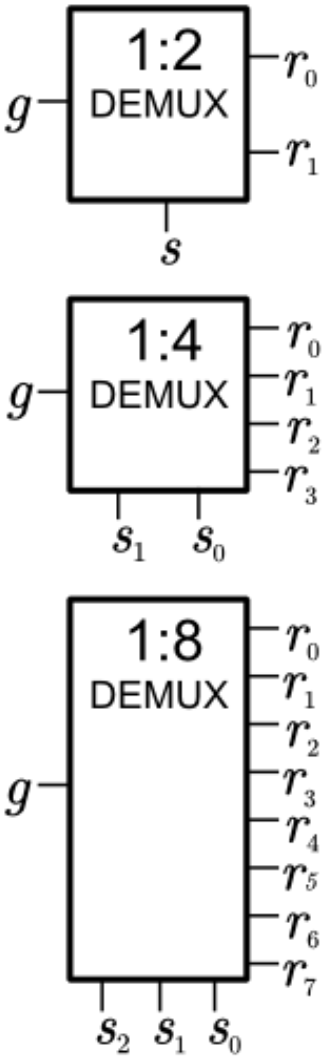
	x_2	x_1	x_0	y
0:	0	0	0	1
1:	0	0	1	0
2:	0	1	0	1
3:	0	1	1	0
4:	1	0	0	0
5:	1	0	1	0
6:	1	1	0	1
7:	1	1	1	1



Demultiplexer

- Der Demultiplexer stellt quasi die inverse Funktion zum Multiplexer dar
  - 1 Eingang g, k Steuereingänge s_{k-1},.., s_0 und n = 2^k Ausgänge r_{n-1}, ..., r_0
  - Der Eingang g wird häufig "Enabler" genannt
  - Abhängig von den Steuereingängen
    - wird der Eingang auf einen der Ausgänge geschaltet
    - die anderen Ausgänge sind 0





Demultiplexer

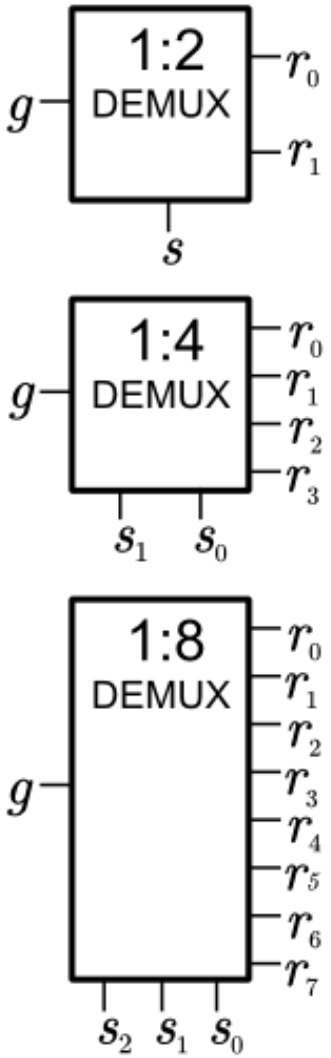
- 1-zu-2 Demultiplexer

$$r_0 = g\,\bar{s}$$
$$r_1 = g\,s$$

- 1-zu-4 Demultiplexer

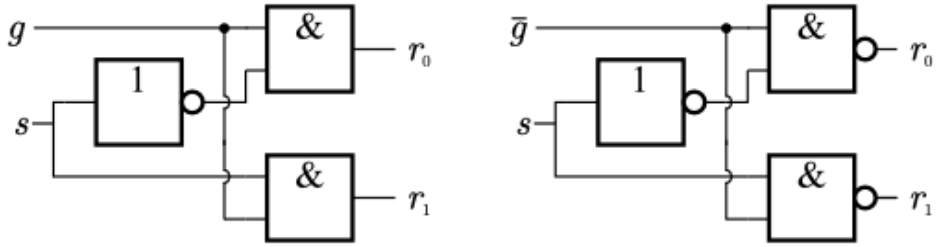
$$r_0 = g\,\bar{s}_1\,\bar{s}_0$$
$$r_1 = g\,\bar{s}_1\,s_0$$
$$r_2 = g\,s_1\,\bar{s}_0$$
$$r_3 = g\,s_1\,s_0$$

- ...

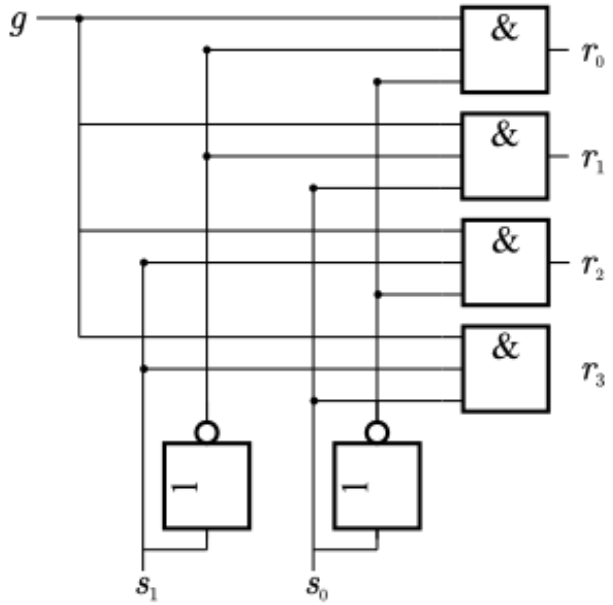


Realisierung von Demultiplexern

- 1-zu-2 Demultiplexer

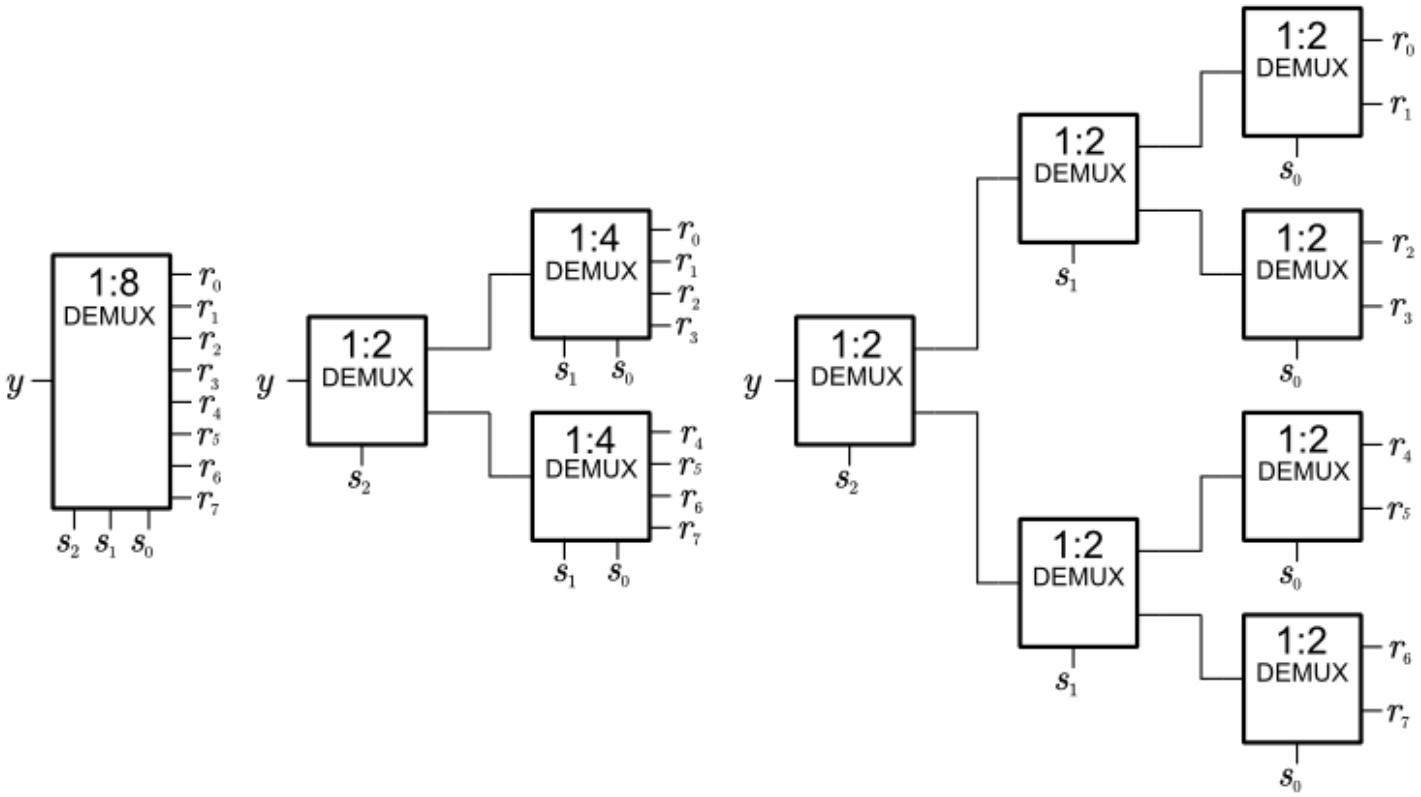


- 1-zu-4 Demultiplexer



Kaskadierung von Demultiplexern

- Große Demultiplexer können durch Kaskadierung von kleinen Demultiplexern aufgebaut werden



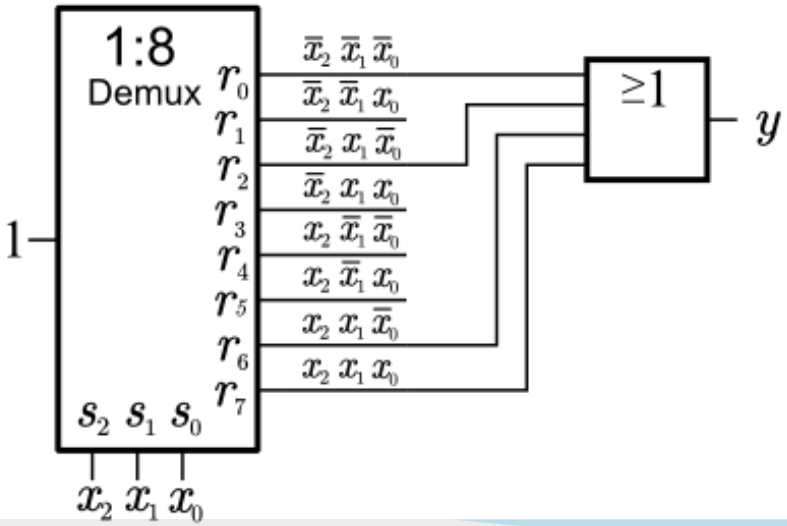
Demultiplexer als universelle Logikbausteine

- 1-zu- Demultiplexer können jede Funktion von k = $\log_2(n)$ Variablen implementieren
  - Die Variablen x_{k-1}, ..., x_0 werden als Steuervariablen s_{k-1}, ..., s_0 verwendet
  - Der Eingang wird auf 1 gelegt
  - An den Ausgängen liegen die Minterme der Steuervariablen an und können durch ein ODER-Gatter verknüpft werden, um die Funktion zu implementieren
- Beispiel

$y = m_0 \vee m_2 \vee m_6 \vee m_7$

Wahrheitstafel:

	x_2	x_1	x_0	y
0:	0	0	0	1
1:	0	0	1	0
2:	0	1	0	1
3:	0	1	1	0
4:	1	0	0	0
5:	1	0	1	0
6:	1	1	0	1
7:	1	1	1	1

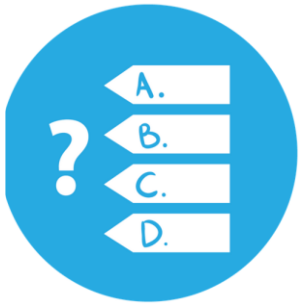


Addition

- Zur Entwicklung einer Schaltung für die Addition kann man sich an der schriftlichen Addition orientieren

```
  13       1101
+  5     + 0101
----     ------
  18      10010
```

- Die Addition wird Bit-weise von rechts nach links durchgeführt (angefangen beim niederwertigsten Bit)



- Warum ist eine Schaltung für die Addition so extrem wichtig?

Halbaddierer

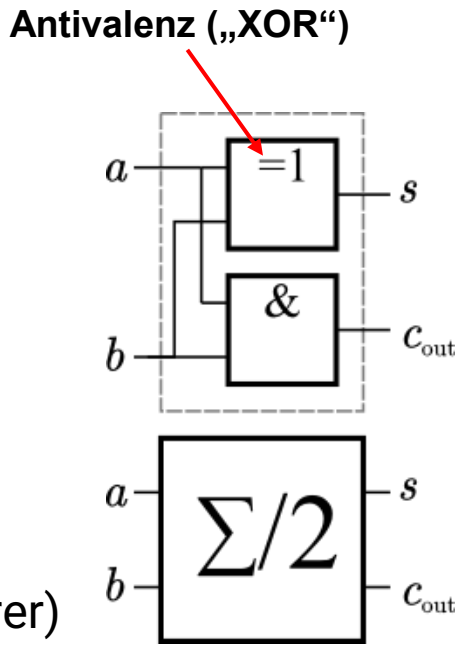
- Für die Addition zweier Dualzahlen
  - Summe und Übertrag entstehen als Ergebnis
- Beim niederwertigsten Bit müssen **nur** alle Kombinationen der zwei Summanden a und b betrachtet werden (4 Fälle)
  - ohne Übertrag
- **Wahrheitstabelle** von Eingangsvariablen a und b sowie das Ergebnis der Addition s und Übertrag c_{out} (Carry-out)

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

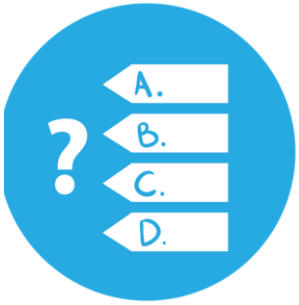
- Damit ergibt sich $s = \bar{a}b \vee a\bar{b} = a \nleftrightarrow b$

$$c_{out} = ab$$

- Die Realisierung mit Logikgattern wird Halbaddierer genannt und erhält ein eigenes Symbol (siehe rechts, 1-Bit-Halbaddierer)



Halbaddierer

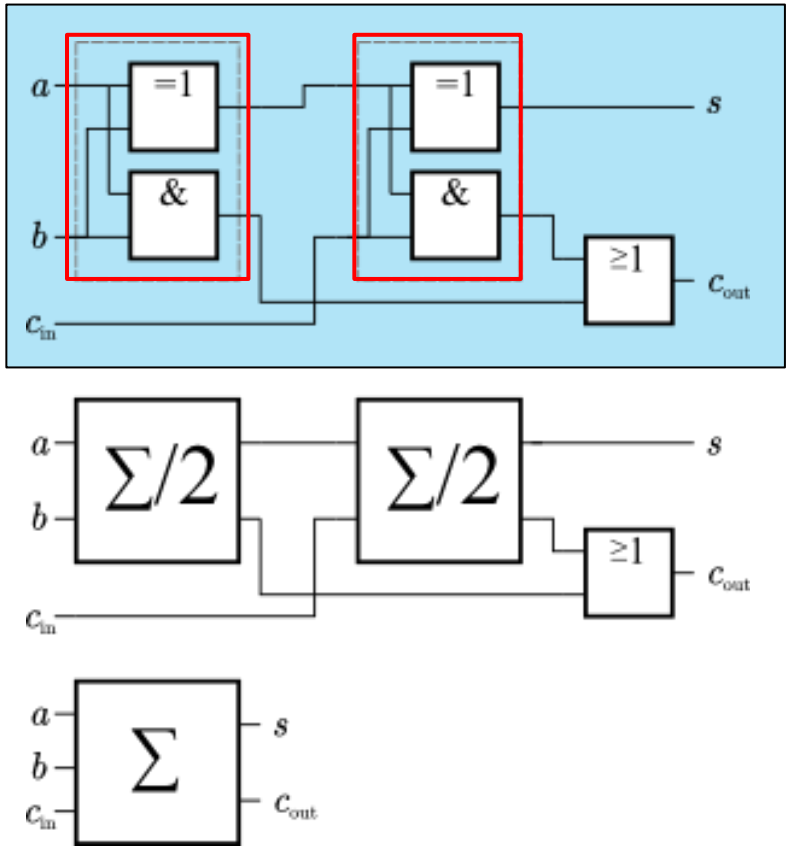


- Warum heißt es eigentlich „Halbaddierer“? Was fehlt?

Volladdierer

- Bei den nachfolgenden Bits muss nun ebenfalls der Übertrag der vorangegangen bit-weisen Addition berücksichtigt werden (Carry-in)

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



- Damit ergibt sich

$$s = \bar{a}\bar{b}c_{in} \vee \bar{a}b\bar{c}_{in} \vee a\bar{b}\bar{c}_{in} \vee abc_{in}$$

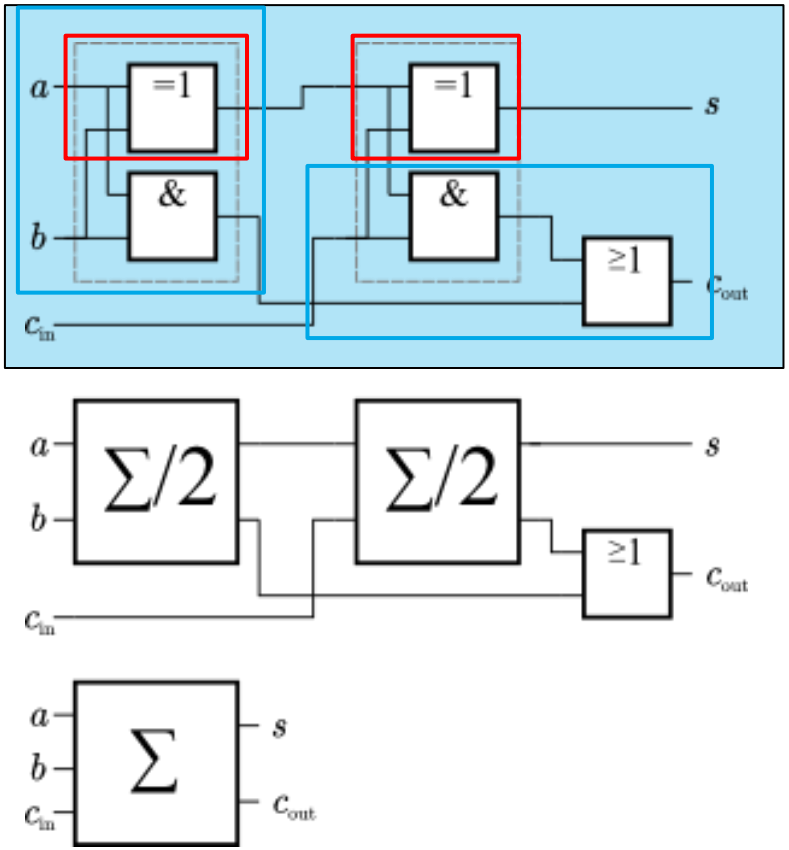
$$c_{out} = \bar{a}bc_{in} \vee a\bar{b}c_{in} \vee ab\bar{c}_{in} \vee abc_{in}$$

Volladdierer

- Umformung der boole'schen Ausdrücke

$$
\begin{aligned}
s &= \bar{a}\bar{b}c_{\text{in}} \vee \bar{a}b\bar{c}_{\text{in}} \vee a\bar{b}\bar{c}_{\text{in}} \vee abc_{\text{in}} \\
&= \left(\bar{a}\bar{b}c_{\text{in}} \vee abc_{\text{in}}\right) \vee \left(\bar{a}b\bar{c}_{\text{in}} \vee a\bar{b}\bar{c}_{\text{in}}\right) \\
&= \left(\left(\bar{a}\bar{b} \vee ab\right)c_{\text{in}}\right) \vee \left(\left(\bar{a}b \vee a\bar{b}\right)\bar{c}_{\text{in}}\right) \\
&= \left((a \leftrightarrow b)c_{\text{in}}\right) \vee \left((a \nleftrightarrow b)\bar{c}_{\text{in}}\right) \\
&= \left(\left(\overline{a \nleftrightarrow b}\right)c_{\text{in}}\right) \vee \left((a \nleftrightarrow b)\bar{c}_{\text{in}}\right) \\
&= (a \nleftrightarrow b) \nleftrightarrow c_{\text{in}}
\end{aligned}
$$

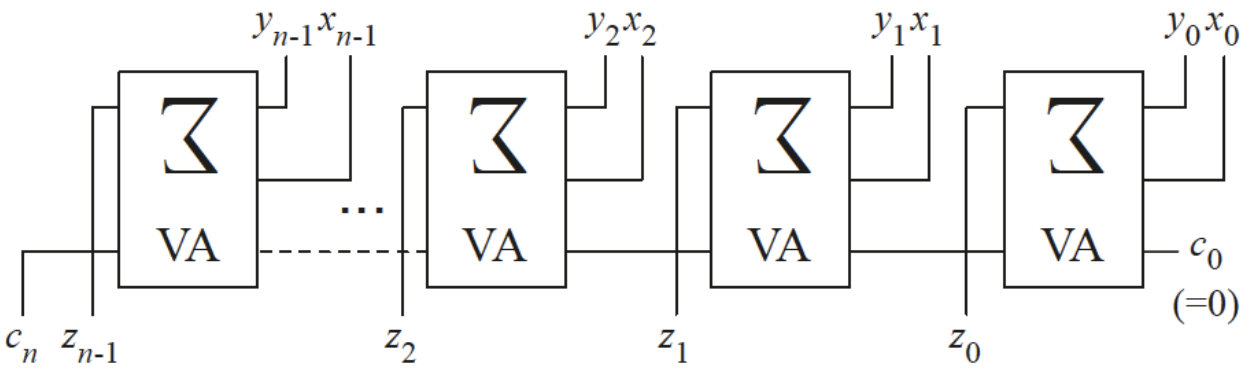
$$
\begin{aligned}
c_{\text{out}} &= \left(\bar{a}bc_{\text{in}} \vee a\bar{b}c_{\text{in}}\right) \vee \left(ab\bar{c}_{\text{in}} \vee abc_{\text{in}}\right) \\
&= \left(\left(\bar{a}b \vee a\bar{b}\right)c_{\text{in}}\right) \vee \left(ab\left(\bar{c}_{\text{in}} \vee c_{\text{in}}\right)\right) \\
&\overset{(5)}{=} (a \nleftrightarrow b)c_{\text{in}} \vee ab
\end{aligned}
$$



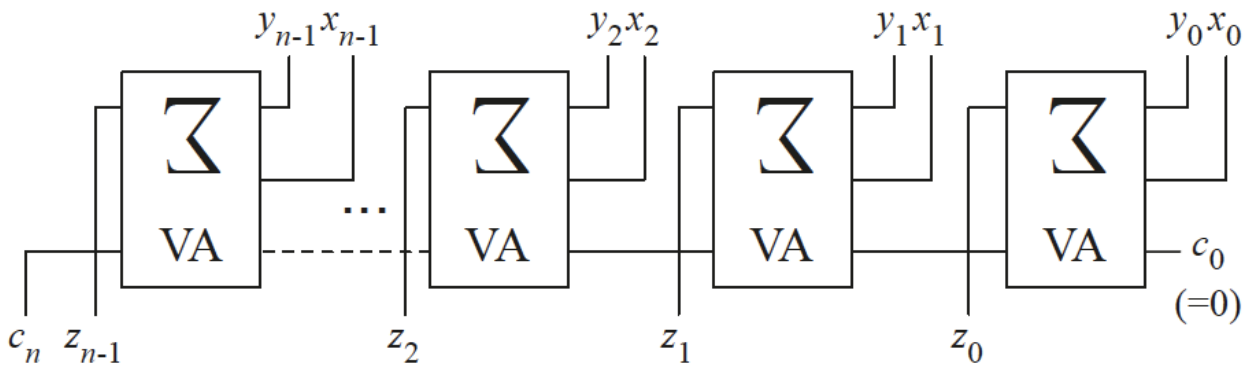
Carry-Ripple-Addierer

- Ein Addierwerk für Binärzahlen der Stelligkeit n kann mit n Volladdierern realisiert werden
  - also => Eine 4-Bit Dualzahl kann mit vier Volladdierern errechnet werden
  - 2n Eingangsvariablen, n+1 Ausgangsvariablen
- Jeder Ein-Bit-Addierer ist für eine Ziffer z_i der Summe verantwortlich
- Der c_{out}-Ausgang wird jeweils mit dem c_{in} des nächsten Ein-Bit-Addierers verbunden
- Die Laufzeit ist linear in der Anzahl der Stellen,
  - da jeder Ein-Bit-Addierer zunächst auf den Übertrag seines Vorgängers warten muss => Keine synchrone Berechnung der Bits
  - d.h. ein 8-Bit-Addierwerk braucht doppelt so lange, wie ein 4-Bit-Addierwerk
- Die Behandlung des Übertrags (Carry) ist verantwortlich für die Laufzeit
  - Das Carry "rieselt" (engl. "ripple") langsam durch die Schaltung
  - Das Durchlaufen aller Stufen muss abgewartet werden!

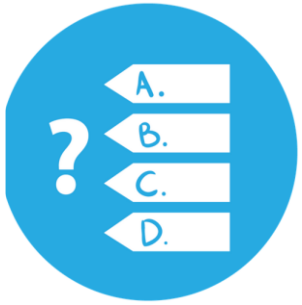
Carry-Ripple-Addierer



Carry-Ripple-Addierer



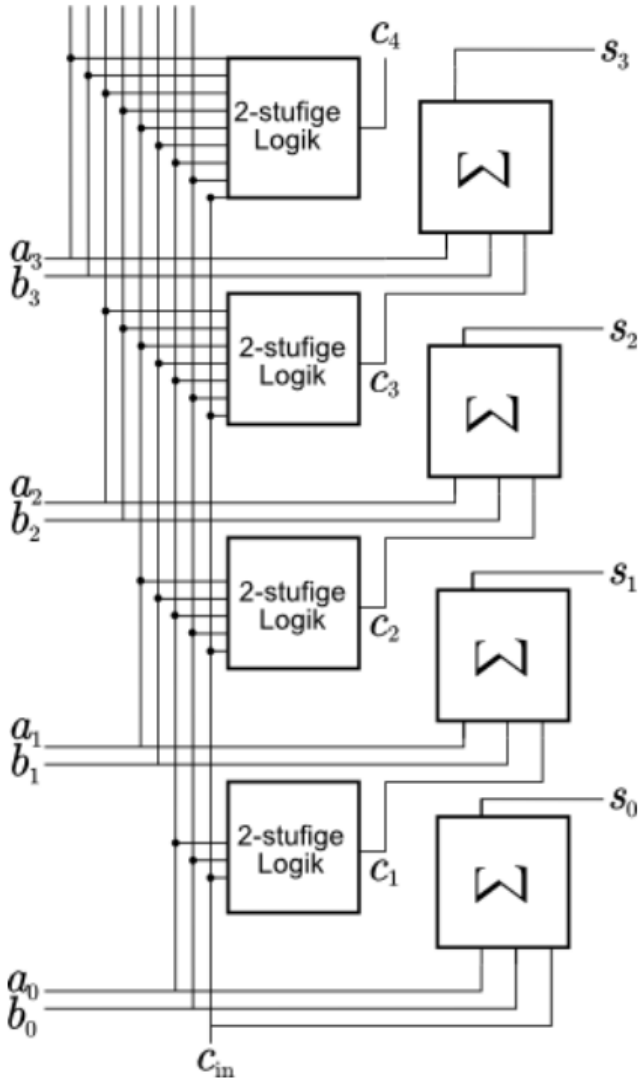
- Vorteile
  - pro Bit genau ein Volladdierer
    - Flächenbedarf steigt linear an
  - Anzahl Gatter proportional zur Wortbreite
- Nachteile
  - relativ langsam



- Kann der Carry-Ripple-Addierer auch anders realisiert werden?

Carry-Look-Ahead-Addierer

- Idee:
  Parallele Berechnung aller Überträge durch separate Logikschaltung
  - Jede boole'sche Funktion kann durch zweistufiges Schaltnetz realisiert werden
    - →Normalform, Minimalform
  - Alle Ein-Bit-Addierer arbeiten parallel → Ergebnisse werden zeitgleich geliefert
  - Mit jeder Ziffer steigt der Aufwand für die Realisierung der Übertragsberechnung mittels zweistufiger Logik, da mit jeder Ziffer zwei Eingänge hinzukommen



Carry-Look-Ahead-Addierer

- Um die Anzahl der Gatter innerhalb der 2-stufigen Logik zu reduzieren, wird der boolesche Ausdruck zur Berechnung des Übertrags beim Volladdierer betrachtet
  - Um diese Werte bei der Berechnung des Übertrags abzugreifen, werden die Volladdierer in zwei Halbaddierer aufgespaltet
- Bei gegebenem Übertrag c_i des vorherigen Ein-Bit-Addierers berechnet sich c_{i+1}

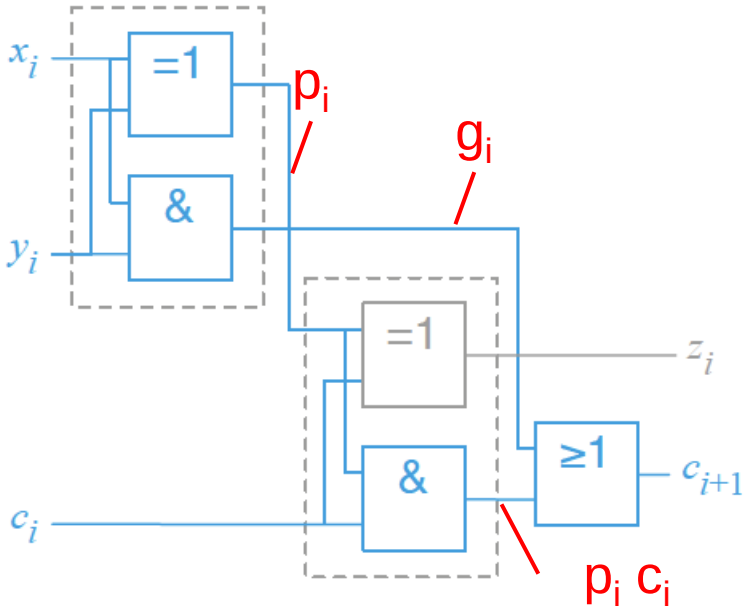
$$c_{i+1} = (x_i \wedge y_i) \vee (c_i \wedge (x_i \nleftrightarrow y_i))$$

- Wir definieren

$$g_i := x_i \wedge y_i$$

$$p_i := x_i \nleftrightarrow y_i$$

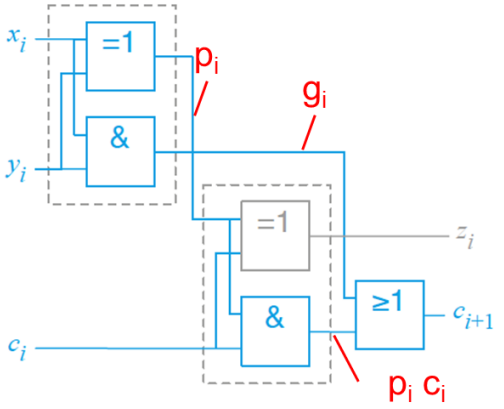
  - dies sind genau die beiden Ausgänge der beiden Halbaddierer



Carry-Look-Ahead-Addierer

- Durch rekursive Anwendung des Ausdrucks zur Berechnung des Übertrags

$$c_{i+1} = \underbrace{(x_i \wedge y_i)}_{g_i} \vee (c_i \wedge \underbrace{(x_i \leftrightarrow y_i)}_{p_i})$$



.. kann die benötigte 2-stufige Logik ausgerechnet werden:

$$c_1 = p_0 c_0 \vee g_0$$

$$c_2 = p_1 c_1 \vee g_1$$

$$= \quad \square$$

$$= \quad \square$$

$$c_3 = p_2 c_2 \vee g_2$$

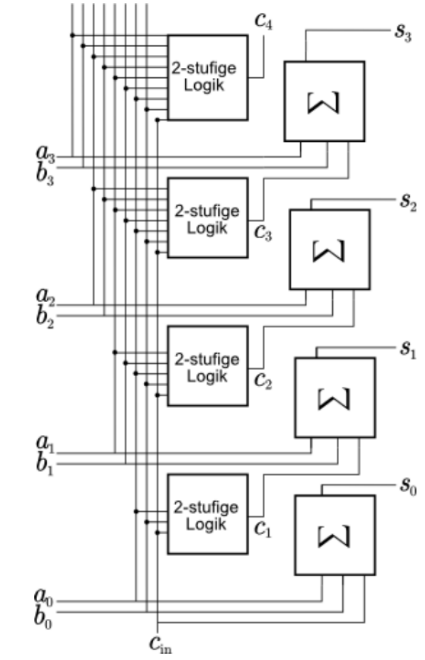
$$= \quad \square$$

$$= \quad \square$$

$$c_4 = p_3 c_3 \vee g_3$$

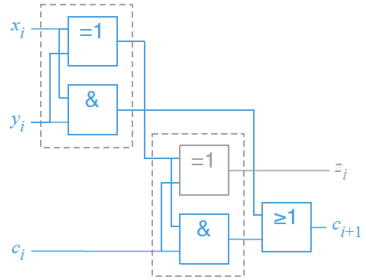
$$= \quad \square$$

$$= \quad \square$$



Carry-Look-Ahead-Addierer

Reminder: Volladdierer



Carry-Look-Ahead-Addierer

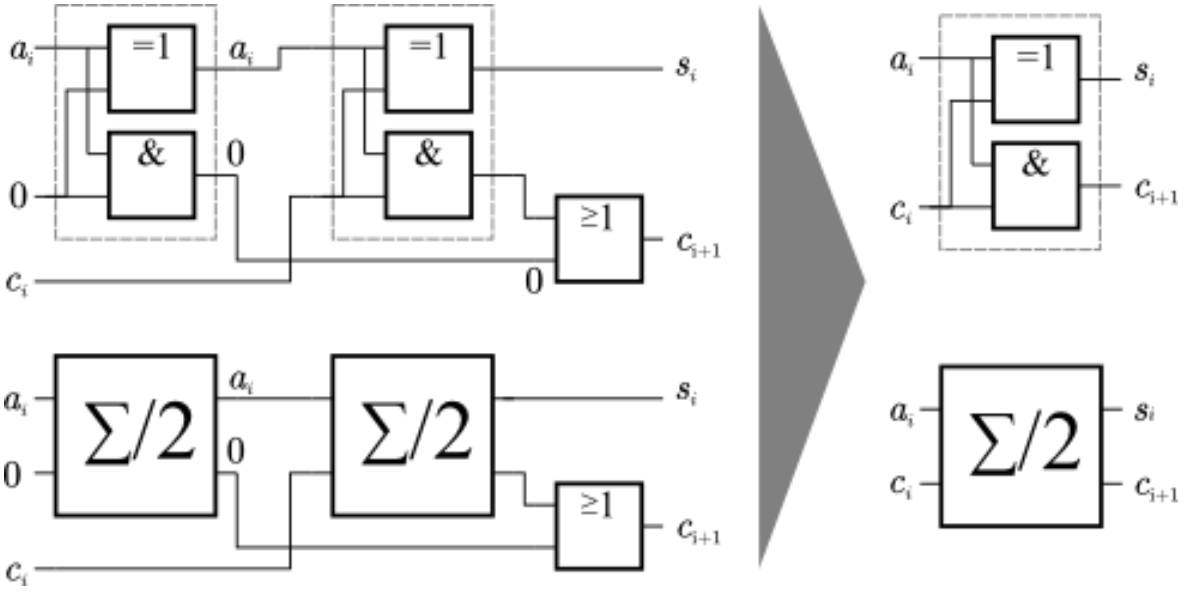
- Wie beim Carry-Ripple-Addierer werden n Volladdierer zur Addition zweier n-Bit breite Dualzahlen verwendet
- Das Übertragsbit ci wird jedoch **nicht** mehr vom Addierer zu Addierer **durchgereicht**
- Alle der n Übertragsbits werden in einem separaten zweistufigen Schaltnetz gleichzeitig berechnet und parallel in die entsprechenden Volladdierer geleitet
- **Vorteil**: Die Additionszeit wird weitgehend unabhängig von der Stellenzahl der Dualzahlen, weil die **Berechnung des Übertrags in allen Stufen sofort** („vorausschauend“) beginnen kann.
- Die Tiefe der Schaltung ist konstant
  - Unabhängig von der Anzahl der zu addierenden Bits durchläuft ein Signal von Eingang zu den Ausgängen eine konstante Anzahl an Logikgattern

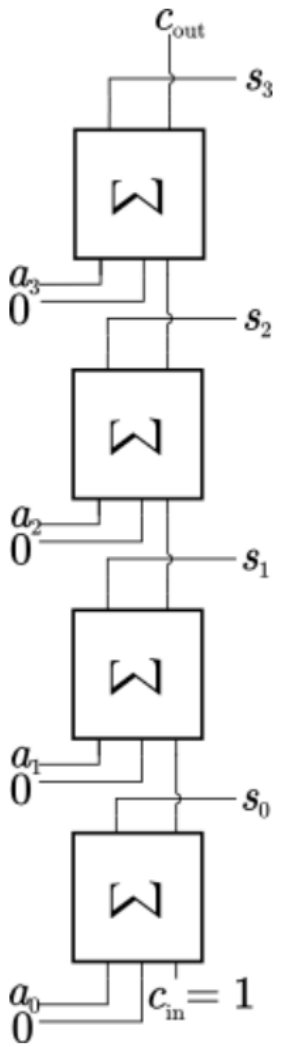
Carry-Ripple-Addierer vs Carry-Look-Ahead-Addierer

- Carry-Ripple
  - Flächenökonomisch
    - Anzahl der Gatter wächst linear
  - Langsam
    - Jeder Ein-Bit-Addierer muss zunächst auf den Übertrag seines Vorgängers warten muss
- Carry-Look-Ahead-Addierer
  - Flächenintensiv
    - Der Flächenbedarf steigt kubisch mit der Bitbreite an
  - Schnell
    - Die jeweiligen Überträge werden parallel und nicht seriell berechnet
- Weitere Addierer
  - Conditional-Sum-Addierer
  - Präfix-Addierer
  - …

Inkrementierer

- Das Inkrementieren einer Zahl ist eine häufig vorkommende Operation
  - a = a + 1;
  - a++;
- n-stelliger Inkrementierer kann durch n-stelliges Addierwerk realisiert werden
  - alle b_i gleich 0; Übertrag am niederwertigsten Bit c_0 = 1

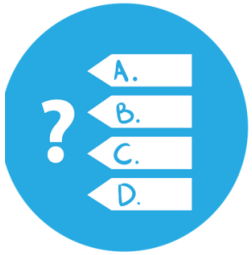




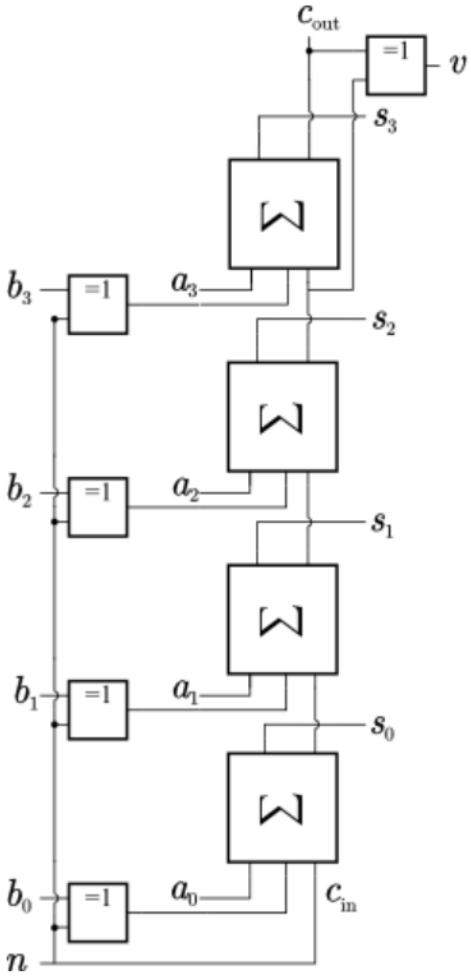
- Durch feste Verdrahtung einiger Variablen kann die Schaltung vereinfacht werden, z.B. können beim Carry-Ripple-Addierer die Volladdierer durch entsprechend beschaltete Halbaddierer ersetzt werden

Subtrahierer

- Die Subtraktion zweier Zahlen $y=a-b$ kann auf die Addition zurückgeführt werden $y=a+(-b)$
  - Addition des Zweierkomplements



- Recap: Wie werden die negativen Zahlen im Zweierkomplement gebildet?



- Ein n-stelliger Subtrahierer kann mit einem n-stelligen Addiererwerk realisiert werden, indem eine **zusätzliche Logik** zu **Berechnung** des **Zweierkomplements** hinzugefügt wird
- Der Eingang n wählt aus, ob b negiert werden soll, und der Ausgang v zeigt an, ob ein Überlauf (Overflow) aufgetreten ist
  - n=0 => Addition zweier positiver Zahlen; n=1 => Addition mit negativer Zahl
  - n ist Eingangssignal für die XOR-Gatter & für den ersten Volladdierer

Multiplizierer

- Zur Entwicklung einer Schaltung für die Multiplikation wird zunächst die schriftliche Multiplikation (wie aus der Schule bekannt) betrachtet:

$25 \cdot 13$
75
+25
$= 325$

$11001 \cdot 1101$
11001
00000
11001
+11001
$= 101000101$

- Es ist zu erkennen, dass die Multiplikation von Binärzahlen durch eine Verschiebung und Addition realisiert werden kann

- Vorzeichenlose Zahlen lassen sich so ohne weiteres multiplizieren
- Einer- und Zweierkomplementzahlen müssen bei vorzeichenbehafteten Zahlen zunächst in eine Form mit Betrag und Vorzeichen umgewandelt werden
  - Nach der Multiplikation wird das Ergebnis wieder in die vorherige Form umgewandelt
  - Das Vorzeichenbit wird separat betrachtet
    - Das Vorzeichen des Ergebnisses ergibt sich aus der Antivalenz (XOR)-Verknüpfung der Vorzeichen der beiden Eingabewerte

Matrixmultiplizierer

- Eine Schaltung, die das schriftliche Multiplizieren nachbildet, ist der Matrixmultiplizierer
- gegeben
  - Multiplikator a = a_{n-1} ... a_0
  - Multiplikant b = b_{n-1} ... b_0
- Die Anordnung der Bits beim schriftlichen Multiplizieren kann als $n \times (2n-1)$ Matrix dargestellt werden

$$\begin{bmatrix} 0 & \dots & 0 & a_{n-1}b_0 & a_{n-2}b_0 & \dots & a_2b_0 & a_1b_0 & a_0b_0 \\ 0 & \dots & a_{n-1}b_1 & a_{n-2}b_1 & a_{n-3}b_1 & \dots & a_1b_1 & a_0b_1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n-1}b_{n-1} & \dots & a_1b_{n-1} & a_0b_{n-1} & 0 & \dots & 0 & 0 & 0 \end{bmatrix}$$

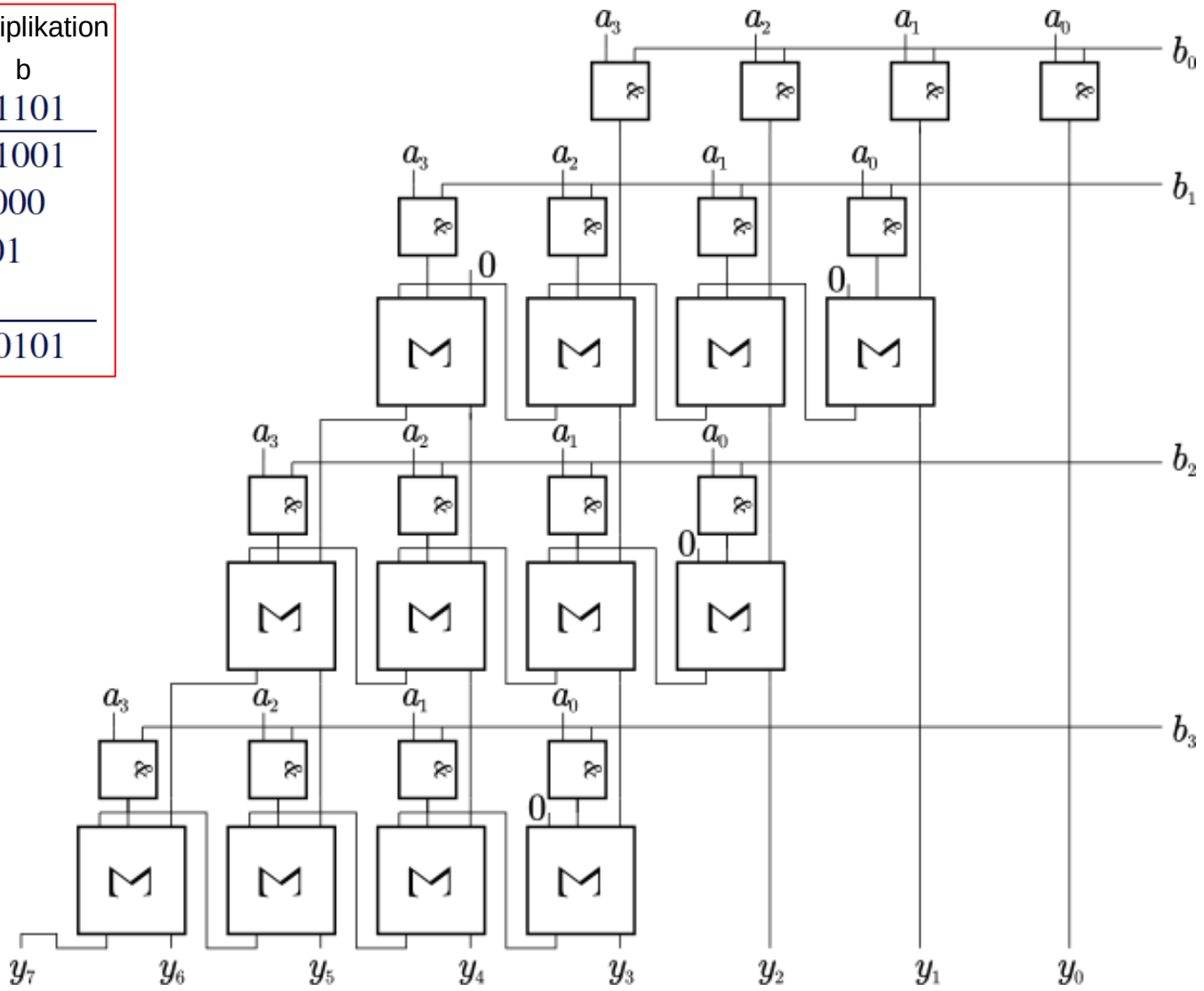
- Struktur der Matrix spiegelt sich in der Logikschaltung wieder
  - Die Elemente der Matrix werden durch UND-Verknüpfungen a_jb_i berechnet
  - Jede Spaltensumme wird durch ein Addierwerk berechnet (maximal n-1 Volladdierer)

4-Bit-Matrixmultiplizierer

Schriftliche Multiplikation

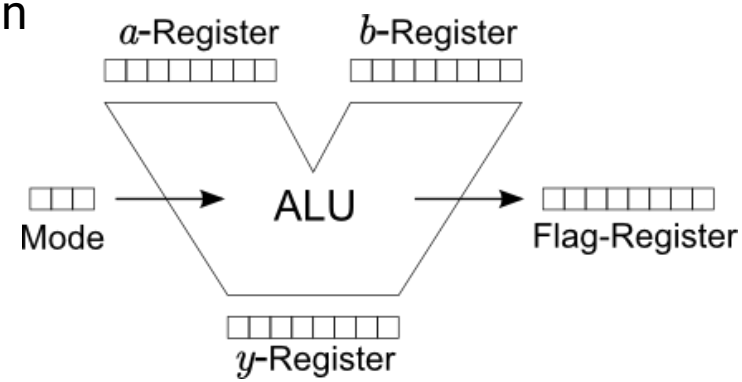
a b

$$
\begin{array}{r}
11001 \cdot 1101 \\
\hline
11001 \\
00000 \\
11001 \\
+11001 \\
\hline
= 101000101
\end{array}
$$



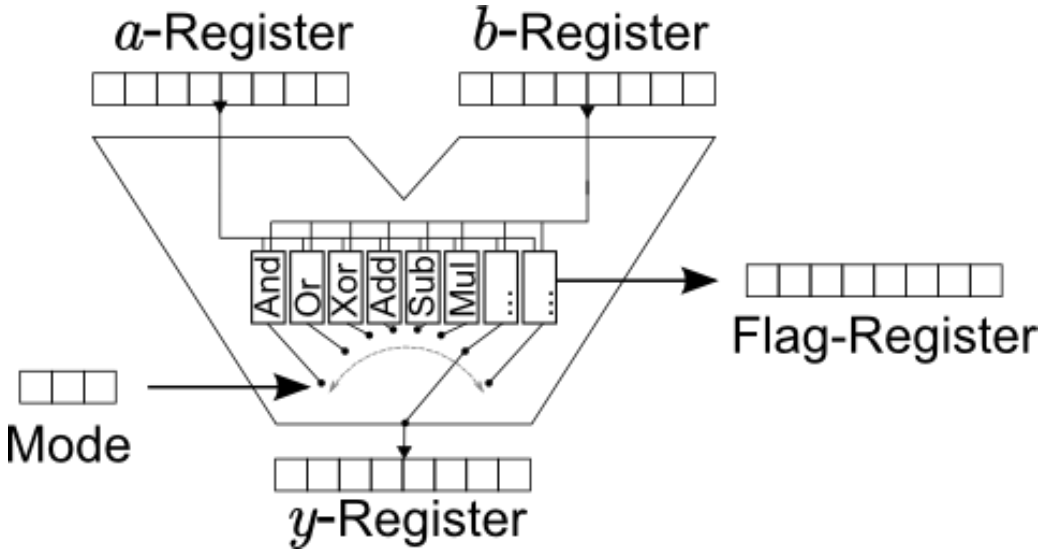
Arithmetisch-logische Einheit ALU

- Die arithmetisch-logische Einheit (engl. **a**rithmetic **l**ogic **u**nit, ALU) dient zur Realisierung der Elementaroperationen eines Rechners:
  - arithmetische Operationen
    - Addition, Subtraktion, Multiplikation, ...
  - logische Operationen
    - bitweise UND-Verknüpfung, ODER-Verknüpfung, ...
    - Test auf Gleichheit
- Verknüpfung von zwei Werten a und b zu einem y
  - werden in Registern bereitgestellt
    - Wortbreite: 8, 16, 32, 64 Bits
  - „Mode“ wählt die auszuführende Operation
  - „Flag“ liefert Status-Informationen
    - z.B. wenn ein Überlauf stattfindet



Arithmetisch-logische Einheit ALU

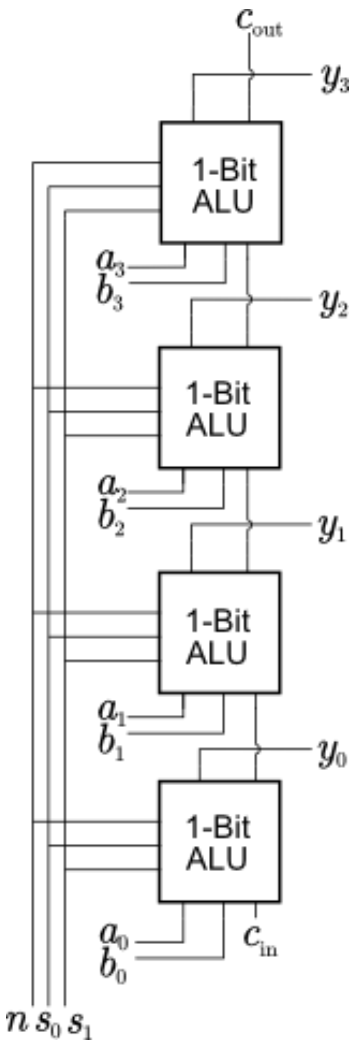
- Flag-Register (Statusregister)
  - Übertrag (Carry flag): Übertrag ist aufgetreten
  - Überlauf (Overflow flag): Ergebnis passt nicht ins y-Register
  - Null (Zero flag): Ergebnis ist Null
  - Negativ (Negation flag): Ergebnis ist negativ
- Der Mode-Eingang wählt die auszuführende Operation aus



Beispiel-ALU

- Zur Realisierung einer einfachen 4-Bit ALU soll diese aus mehreren 1-Bit-ALUs zusammengesetzt werden
- Wie bereits von den Addierwerken bekannt, muss dabei die Carry-Information an den Nachfolger weitergegeben werden
- Insgesamt soll die hier gezeigte beispielhafte ALU acht verschiedene Operationen ausführen:
- Acht Operationen, für alle ALUs einheitlich gesteuert

s_1	s_0	n	y	
0	0	0	$a \vee b$	Logisches ODER
0	0	1	$a \vee \overline{b}$	Logisches ODER mit Invertierung von b
0	1	0	$a \wedge b$	Logisches UND
0	1	1	$a \wedge \overline{b}$	Logisches UND mit Invertierung von b
1	0	0	$\overline{b}$	Invertiertes b
1	0	1	b	b
1	1	0	$a + b$	Addition
1	1	1	$a + \overline{b}$	Addition mit Invertierung von b (für Subtraktion)



Beispiel-ALU