# Technische Informatik
# Arithmetik Schaltungen

Thorsten Thormählen
02. Dezember 2021
Teil 7, Kapitel 2

Dies ist die Druck-Ansicht.

Aktiviere Präsentationsansicht

## Steuerungstasten

→ nächste Folie (auch Enter oder Spacebar).
← vorherige Folie
d schaltet das Zeichnen auf Folien ein/aus
p wechselt zwischen Druck- und Präsentationsansicht
CTRL + vergrößert die Folien
CTRL - verkleinert die Folien
CTRL 0 setzt die Größenänderung zurück

# Notation

| Typ | Schriftart | Beispiele |
|---|---|---|
| Variablen (Skalare) | kursiv | $a, b, x, y$ |
| Funktionen | aufrecht | $\mathrm{f}, \mathrm{g}(x), \max(x)$ |
| Vektoren | fett, Elemente zeilenweise | $\mathbf{a}, \mathbf{b} = \begin{pmatrix} x \\ y \end{pmatrix} = (x, y)^\top$, $\mathbf{B} = (x, y, z)^\top$ |
| Matrizen | Schreibmaschine | $\mathtt{A}, \mathtt{B} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ |
| Mengen | kalligrafisch | $\mathcal{A}, B = \{a, b\}, b \in \mathcal{B}$ |
| Zahlenbereiche, Koordinatenräume | doppelt gestrichen | $\mathbb{N}, \mathbb{Z}, \mathbb{R}^2, \mathbb{R}^3$ |

## Inhalt

## Addition

Zur Entwicklung einer Schaltung für die Addition kann sich an der schriftlichen Addition orientiert werden:

$$\begin{array}{r} 13 \\ +\ 5 \\ \hline = 18 \end{array} \qquad \begin{array}{rl} 1101 & (13) \\ +0101 & (05) \\ \hline = 10010 & (18) \end{array}$$

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

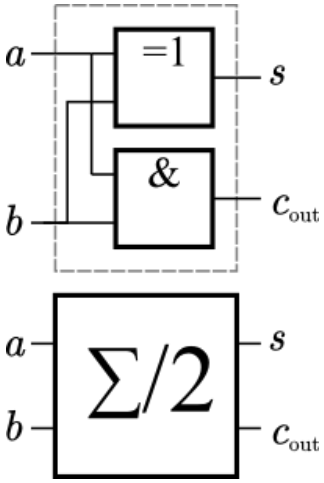
Es kann zu einem Übertrag (Carry) kommen, der beim nächsten Bit berücksichtigt werden muss

# Halbaddierer

Beim niederwertigsten Bit müssen nur alle Kombinationen der zwei Summanden $a$ und $b$ betrachtet werden (4 Fälle)

Frage: Wie lautet jeweils das Ergebnis der Addition $s$ und der Übertrag $c_{\text{out}}$ (Carry-out)?

| $a$ | $b$ | $s$ | $c_{\text{out}}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |



Halbaddierer

Damit ergibt sich:

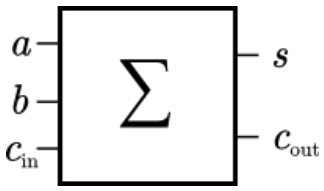
$$s = \bar{a}b \vee a\bar{b} = a \nleftrightarrow b$$

$$c_{\text{out}} = ab$$

Die Realisierung mit Logikgattern wird *Halbaddierer* genannt und erhält ein eigenes Symbol (siehe rechts)

# Volladdierer

Bei den nachfolgenden Bits muss nun ebenfalls der Übertrag $c_{\text{in}}$ der vorangegangen Bit-weisen Addition berücksichtigt werden (Carry-in)



Volladdierer

| $a$ | $b$ | $c_{\text{in}}$ | $s$ | $c_{\text{out}}$ |
|---|---|---|---|---|
| 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:

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

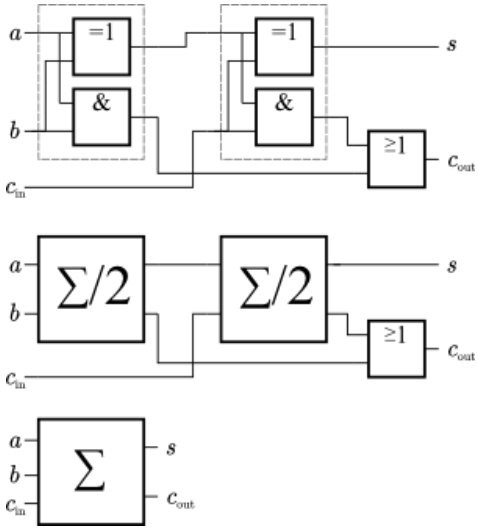
$$c_{\text{out}} = \bar{a}bc_{\text{in}} \vee a\bar{b}c_{\text{in}} \vee ab\bar{c}_{\text{in}} \vee abc_{\text{in}}$$

# Volladdierer

Durch Umformung der booleschen Ausdrücke ergibt sich:

$$
\begin{aligned}
s &= \bar{a}\bar{b}c_{\text{in}} \vee \bar{a}b\bar{c}_{\text{in}} \vee a\bar{b}\bar{c}_{\text{in}} \vee abc_{\text{in}} \\
&\overset{(6)}{=} \left(\bar{a}\bar{b}c_{\text{in}} \vee abc_{\text{in}}\right) \vee \left(\bar{a}b\bar{c}_{\text{in}} \vee a\bar{b}\bar{c}_{\text{in}}\right) \\
&\overset{(8)}{=} \left(\left(\bar{a}\bar{b} \vee ab\right)c_{\text{in}}\right) \vee \left(\left(\bar{a}b \vee a\bar{b}\right)\bar{c}_{\text{in}}\right) \\
&= \left((a \leftrightarrow b)c_{\text{in}}\right) \vee \left((a \nleftrightarrow b)\bar{c}_{\text{in}}\right) \\
&= \left(\left(\overline{a \nleftrightarrow b}\right)c_{\text{in}}\right) \vee \left((a \nleftrightarrow b)\bar{c}_{\text{in}}\right) \\
&= (a \nleftrightarrow b) \nleftrightarrow c_{\text{in}}
\end{aligned}
$$

$$
\begin{aligned}
c_{\text{out}} &= \left(\bar{a}bc_{\text{in}} \vee a\bar{b}c_{\text{in}}\right) \vee \left(ab\bar{c}_{\text{in}} \vee abc_{\text{in}}\right) \\
&\overset{(8)}{=} \left(\left(\bar{a}b \vee a\bar{b}\right)c_{\text{in}}\right) \vee \left(ab\left(\bar{c}_{\text{in}} \vee c_{\text{in}}\right)\right) \\
&\overset{(5)}{=} (a \nleftrightarrow b)c_{\text{in}} \vee ab
\end{aligned}
$$




Volladdierer

# Carry-Ripple-Addierer

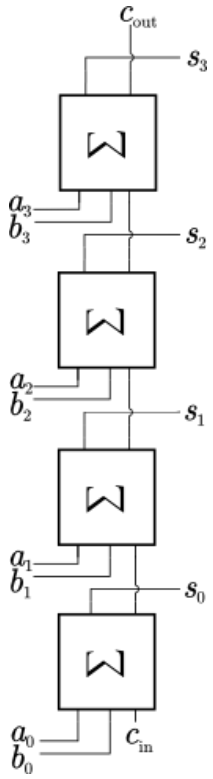
Ein Addierwerk für Binärzahlen der Stelligkeit $n$ kann mit $n$ Volladdierern realisiert werden

Jeder Ein-Bit-Addierer ist für eine Ziffer $s_i$ der Summe verantwortlich

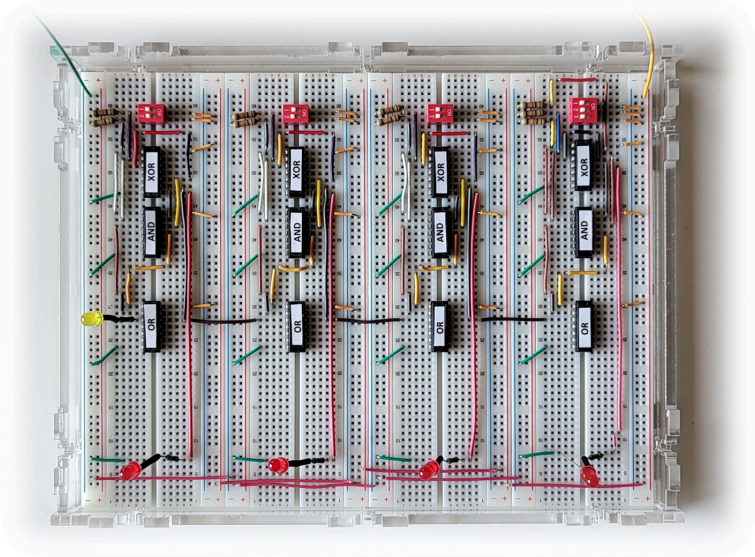
Der $c_{\text{out}}$-Ausgang wird jeweils mit dem $c_{\text{in}}$ des nächsten Ein-Bit-Addierers verbunden

Die Laufzeit ist linear in der Anzahl der Stellen, da jeder Ein-Bit-Addierer zunächst auf den Übertrag seines Vorgängers warten muss (d.h. ein 8-Bit-Addierwerk braucht doppelt so lange, wie ein 4-Bit-Addierwerk)

Die Behandlung des Übertrags (Carry) limitiert die Laufzeit. Das Carry "rieselt" (engl. "ripple") langsam durch die Schaltung.

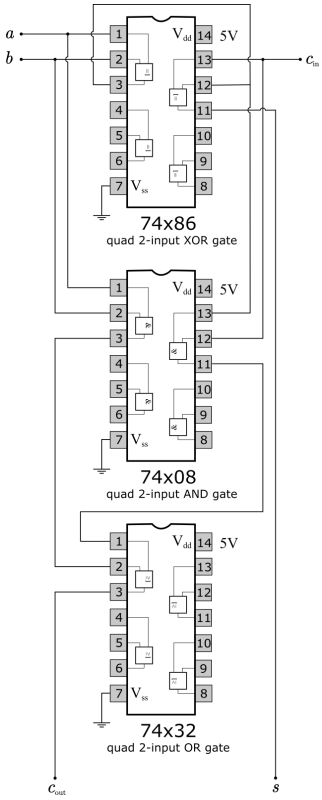


# Aufbau eines Carry-Ripple-Addierers aus Standard-ICs




Im Wintersemester 2024/25 hat Célestine Schönau einen 4-Bit-Carry-Ripple-Addierer aus Standard-ICs der [74xx-Familie] aufgebaut

Die Schaltung kann in der Vorlesung ausprobiert werden



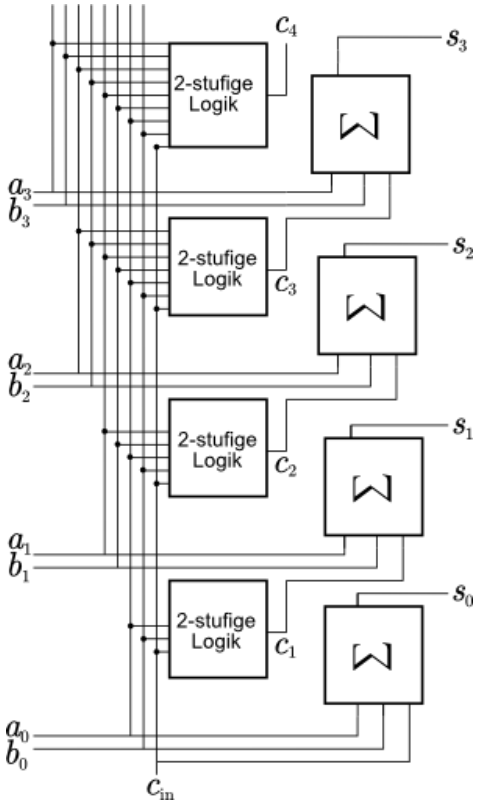
# Carry-Look-Ahead-Addierer

Die Idee des Carry-Look-Ahead-Addierers ist, die Carry-Bits für alle Ein-Bit-Addierer durch eine separate Logikschaltung zu erzeugen

Wie in den vorangegangenen Kapiteln gezeigt wurde, kann jede boolesche Funktion als zweistufige Logik realisiert werden

Beim Carry-Look-Ahead-Addierer können alle Ein-Bit-Addierer parallel arbeiten und liefern zeitgleich ihr Ergebnis

Allerdings steigt mit jeder Ziffer der Aufwand für die Realisierung der Übertragsberechnung mittels zweistufiger Logik, da mit jeder Ziffer zwei Eingänge hinzukommen



# Carry-Look-Ahead-Addierer

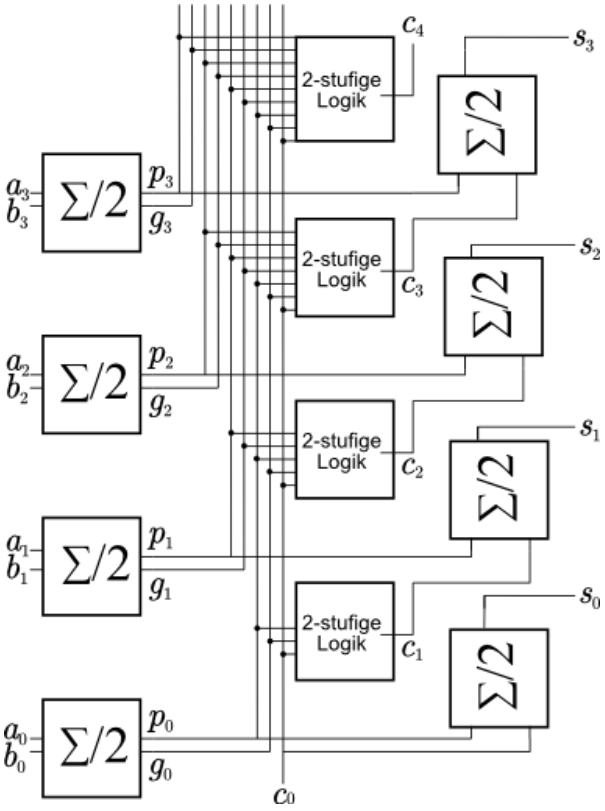
Um die Anzahl der Gatter innerhalb der 2-stufigen Logik zu reduzieren, wird der boolesche Ausdruck zur Berechnung des Übertrags beim Volladdierer betrachtet

Bei gegebenem Übertrag $c_i$ des vorherigen Ein-Bit-Addierers berechnet sich $c_{i+1}$

$$c_{i+1} = \underbrace{(a_i \leftrightarrow b_i)}_{p_i} c_i \vee \underbrace{a_i b_i}_{g_i}$$

Dabei sind $p_i$ und $g_i$ gerade die Ausgänge eines entsprechenden Halbaddierers

Um diese Werte bei der Berechnung des Übertrags abzugreifen, werden die Volladdierer in zwei Halbaddierer aufgespaltet



# Carry-Look-Ahead-Addierer

Durch rekursive Anwendung des Ausdrucks zur Berechnung des Übertrags

$$c_{i+1} = \underbrace{(a_i \nleftrightarrow b_i)}_{p_i} c_i \vee \underbrace{a_i b_i}_{g_i}$$

kann die benötigte 2-stufige Logik ausgerechnet werden:

$$\begin{aligned}
c_1 &= p_0 c_0 \vee g_0 \\
c_2 &= p_1 c_1 \vee g_1 \\
&= p_1 (p_0 c_0 \vee g_0) \vee g_1 \\
&= p_1 p_0 c_0 \vee p_1 g_0 \vee g_1 \\
c_3 &= p_2 c_2 \vee g_2 \\
&= p_2 (p_1 p_0 c_0 \vee p_1 g_0 \vee g_1) \vee g_2 \\
&= p_2 p_1 p_0 c_0 \vee p_2 p_1 g_0 \vee p_2 g_1 \vee g_2 \\
c_4 &= p_3 c_3 \vee g_3 \\
&= p_3 (p_2 p_1 p_0 c_0 \vee p_2 p_1 g_0 \vee p_2 g_1 \vee g_2) \vee g_3 \\
&= p_3 p_2 p_1 p_0 c_0 \vee p_3 p_2 p_1 g_0 \vee p_3 p_2 g_1 \vee p_3 g_2 \vee g_3
\end{aligned}$$

## Inkrement

Das Inkrementieren einer Zahl (a=a+1 bzw. kurz a++) ist eine sehr häufige arithmetische Operation

Z.B. muss nach jeder Befehlsausführung der Programmzähler erhöht werden, oder, beim sequentiellen Lesen aus dem Speicher, der Adresszähler

Ein weiteres Beispiel sind Schleifen, bei denen das Inkrementieren der Zählvariablen um 1 eine typische Operation ist:

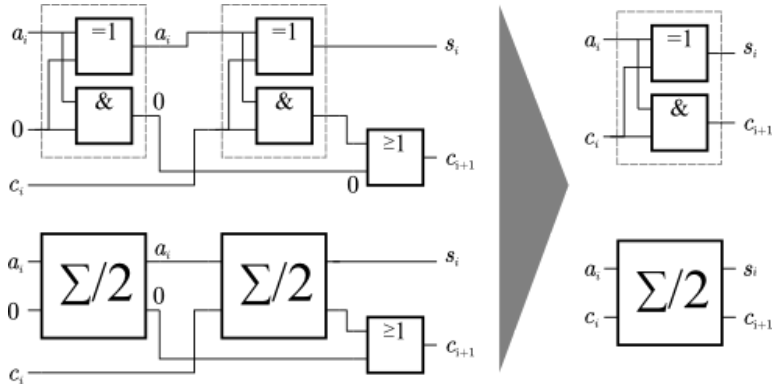
```
for(int a = 0; a < 10; a++) {
  doSomething(a);
}
```

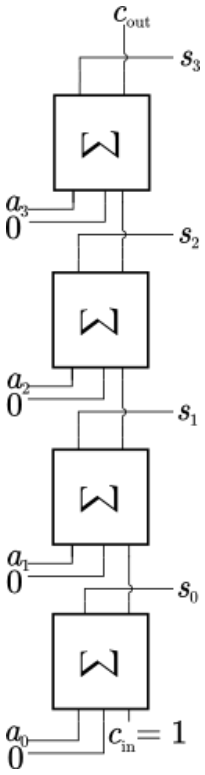
# Inkrement

Ein $n$-stelliger Inkrementierer kann mit einem $n$-stelligen Addiererwerk realisiert werden, indem alle $b_i$ gleich $0$ und der Übertrag am Eingang des niederwertigsten Bits $c_{\text{in}}$ gleich $1$ gesetzt wird

Die Abbildung rechts zeigt dies für den Carry-Ripple-Addierer

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



# Subtraktion

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

Dazu werden die Zahlen im Zweierkomplement dargestellt

Zur Wiederholung: Beim Zweierkomplement werden die negativen Zahlen durch das bitweise Komplement und einer zusätzlichen Addition von 1 gebildet

Beispiel:

$(-6)_{10} \hat{=} 1001 + 1 = 1010$

Subtraktion mit dem Zweierkomplement:

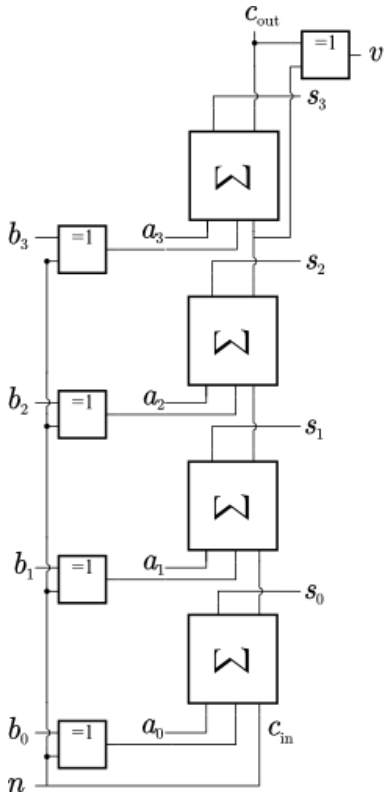
$$\begin{array}{r} 5 \\ +-6 \\ \hline = -1 \end{array} \qquad \begin{array}{rr} 0101 & (5) \\ +1010 & (-6) \\ \hline = 1111 & (-1) \end{array}$$

# Subtraktion

Ein $n$-stelliger Subtrahierer kann mit einem $n$-stelligen Addiererwerk realisiert werden, indem eine zusätzliche Logik zu Berechnung des Zweierkomplements hinzugefügt wird

Die Abbildung rechts zeigt dies am Beispiel des Carry-Ripple-Addierers

Der Eingang $n$ wählt aus, ob $b$ negiert werden soll, und der Ausgang $v$ zeigt an, ob ein Überlauf (Overflow) aufgetreten ist



## Multiplikation

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

$$\begin{array}{r} 25 \cdot 13 \\ \hline 75 \\ +25 \\ \hline = 325 \end{array} \qquad \begin{array}{r} 11001 \cdot 1101 \\ \hline 11001 \\ 00000 \\ 11001 \\ +11001 \\ \hline = 101000101 \end{array}$$

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

# Matrixmultiplizierer

Eine Schaltung, die das schriftliche Multiplizieren nachbildet, ist der Matrixmultiplizierer

Gegeben sei Multiplikator $a = a_{n-1} \dots a_1 a_0$ und Multiplikand $b = b_{n-1} \dots b_1 b_0$ mit Stelligkeit $n$

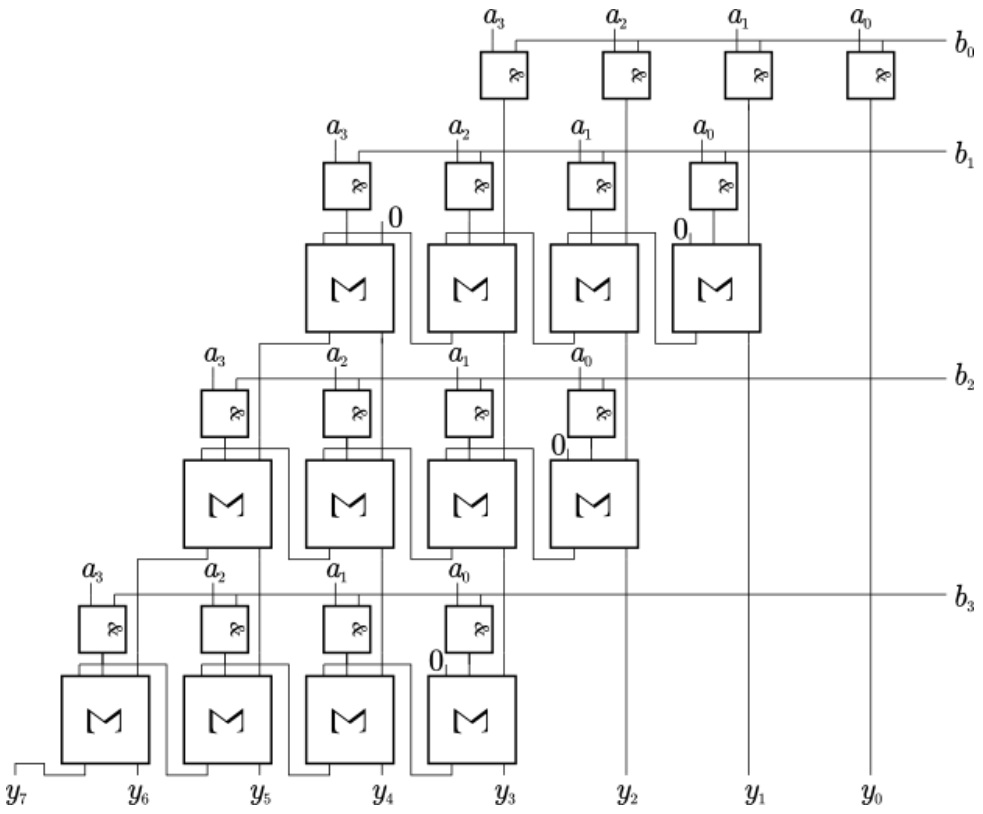
Die Anordnung der Bits beim schriftlichen Multiplizieren kann als $n \times (2n-1)$ Matrix dargestellt werden

$$\begin{bmatrix} 0 & \dots & 0 & a_{n-1}b_0 & a_{n-2}b_0 & \dots & a_2 b_0 & a_1 b_0 & a_0 b_0 \\ 0 & \dots & a_{n-1}b_1 & a_{n-2}b_1 & a_{n-3}b_1 & \dots & a_1 b_1 & a_0 b_1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n-1}b_{n-1} & \dots & a_1 b_{n-1} & a_0 b_{n-1} & 0 & \dots & 0 & 0 & 0 \end{bmatrix}$$

Diese Matrixstruktur findet sich auch in der entsprechenden Logikschaltung wieder

- Die Elemente der Matrix werden durch UND-Verknüpfungen $a_j \wedge b_i$ berechnet
- Jede Spaltensumme durch ein Addierwerk (benötigt maximal $n-1$ Volladdierer)

# 4-Bit-Matrixmultiplizierer



## Multiplizierer

Die vorgestellte Realisierung eines Multiplizierers hat das gleiche Problem, wie der Carry-Ripple-Addierer: Das Warten auf den Übertrag limitiert die Laufzeit

Analog, wie beim Addierer, können Schaltungen entworfen werden, die mehr Logikgatter für die Berechnung des Übertrags bereitstellen und damit die Laufzeit reduzieren

## Arithmetisch-logische Einheit (ALU)

Die arithmetisch-logische Einheit (engl. arithmetic logic unit, ALU) dient zur Realisierung der Elementaroperationen eines Rechners

Dazu gehören

- Arithmetische Operationen: Addition, Subtraktion, Multiplikation, ...
- Logische Operationen: Bitweise UND-Verknüpfung, ODER-Verknüpfung, Prüfen auf Gleichheit, ....

# Arithmetisch-logische Einheit (ALU)

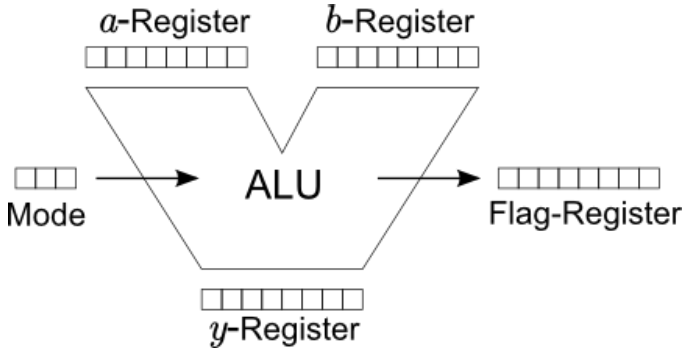
In einer ALU werden zwei Eingabewerte $a$ und $b$ zu einem Ergebniswert $y$ verknüpft

Diese Werte stehen in Registern zur Verfügung (Registerbreite kann 8, 16, 32, 64 Bits sein)

Der "Mode" selektiert, welche Funktion die ALU ausführen soll

Im "Flag" wird der Status der ALU angezeigt, z.B. wenn ein Überlauf stattgefunden hat

Zur schematischen Darstellung hat sich folgendes Symbol etabliert:

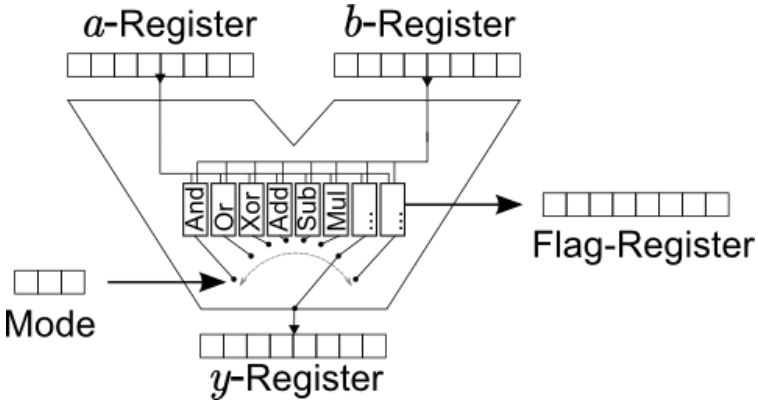


# Arithmetisch-logische Einheit (ALU)

Flag-Register (Statusregister)

- Übertrag $c$ (Carry flag): Übertrag ist aufgetreten
- Überlauf $v$ (Overflow flag): Ergebnis passt nicht ins $y$-Register
- Null $z$ (Zero flag): Ergebnis is Null
- Negativ $n$ (Negation flag): Ergebnis ist negativ
- ...

Der Mode-Eingang wählt die auszuführende Operation aus



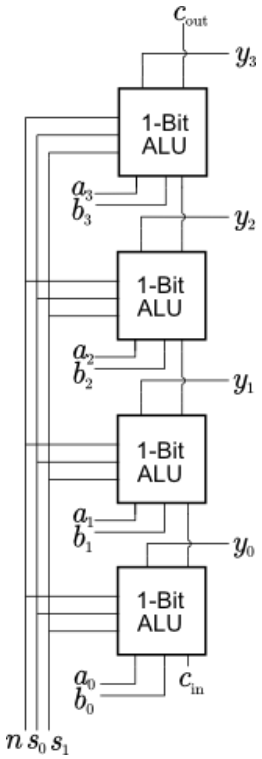
# Beispiel für eine ALU

Zur Realisierug einer einfachen 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

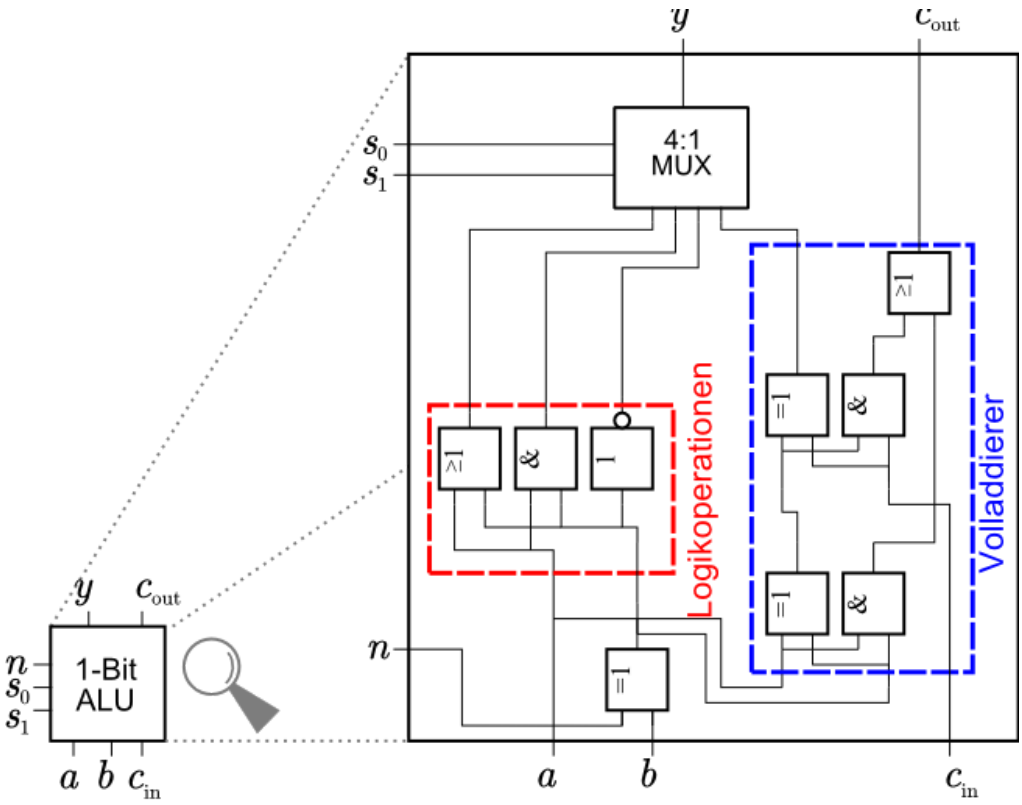
Insgesamt soll die hier gezeigte beispielhafte ALU acht verschiedene Operationen ausführen:

| $s_1$ | $s_0$ | $n$ | $y$ |
|---|---|---|---|
| 0 | 0 | 0 | $a \vee b$ |
| 0 | 0 | 1 | $a \vee \overline{b}$ |
| 0 | 1 | 0 | $a \wedge b$ |
| 0 | 1 | 1 | $a \wedge \overline{b}$ |
| 1 | 0 | 0 | $\overline{b}$ |
| 1 | 0 | 1 | $b$ |
| 1 | 1 | 0 | $a + b$ |
| 1 | 1 | 1 | $a + \overline{b}$ |




4-Bit-ALU

# 1-Bit ALU


y
$c_{out}$
$s_0$
$s_1$
4:1
MUX
Logikoperationen
Volladdierer
n
a
b
$c_{in}$
1-Bit
ALU

# Gibt es Fragen?

Anregungen oder Verbesserungsvorschläge können auch gerne per E-mail an mich gesendet werden: Kontakt

Weitere Vorlesungsfolien