Digitaltechnik – Kapitel 5

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Prof. Sarah Harris, Ph.D.
Fachgebiet Eingebettete Systeme und ihre Anwendungen (ESA)
Fachbereich Informatik

WS 15/16

Nachrichten

- **Klausur:**
  - **01.03.2016 (Dienstag)**
  - **11:00 Uhr – 12:30 Uhr (noch immer unter Vorbehalt)**
- **Kurs Evaluationen:**
  - **Nächste Woche (27.01, Mittwoch, 09:50 Uhr)**
  - **am Anfang der Vorlesung**

Zeitplan

TECHNISCHE UNIVERSITÄT DARMSTADT

- Noch 3 Vorlesungen (20./27.01. und 03.02)
- Am 10.02 werden wir das Material des Semesters wiederholen
  - mit Fragen bereits kommen
  - bei Moodle sich melden was Sie gern wiederholen möchten
- 01.03.2016: Klausur

Embedded Systems & Applications

Kapitel 5 : Themenübersicht

- Einleitung
- Arithmetische Schaltungen
- Zahlendarstellungen
- Sequentielle Grundelemente
- Speicherblöcke
- Programmierbare Logikfelder und -schaltungen

Embedded Systems & Applications

Einleitung

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Grundelemente** digitaler Schaltungen:
  - Gatter, Multiplexer, Decoder, Register, Arithmetische Schaltungen, Zähler, Speicher, programmierbare Logikfelder

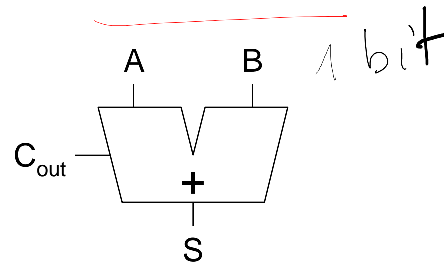
- Grundelemente veranschaulichen:
  - **Hierarchie:** Zusammensetzen aus einfacheren Elementen
  - **Modularität:** Wohldefinierte Schnittstellen und Funktionen
  - **Regularität:** Strukturen leicht auf verschiedene Größen anpassbar

- Grundelemente werden verwendet zum Aufbau eines eigenen **Mikroprozessors** (Sommer Semester)
  - Kapitel 7

Embedded Systems & Applications

1-Bit Addierer

Halb-addierer

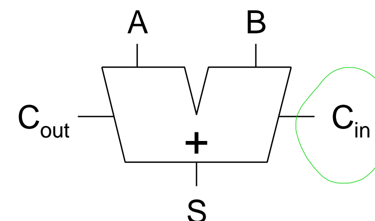


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

S =

C_{out} =

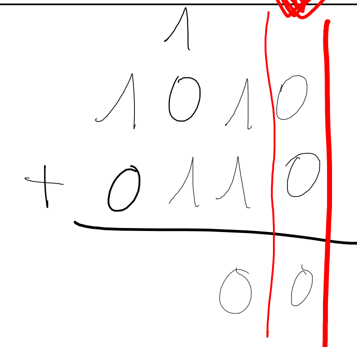
Voll-addierer



C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

S =

C_{out} =

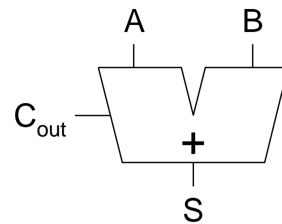


Embedded Systems & Applications

1-Bit Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

Halbaddierer

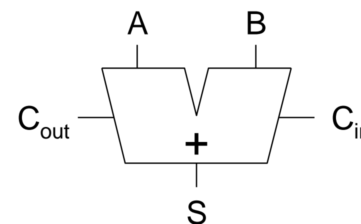


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S =$

$C_{out} =$

Volladdierer



C_{in}	A	B	C_{out}	S
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

$S =$

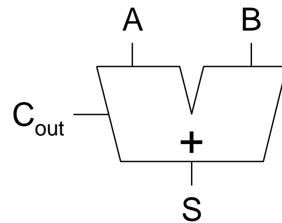
$C_{out} =$

Embedded Systems & Applications

1-Bit Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

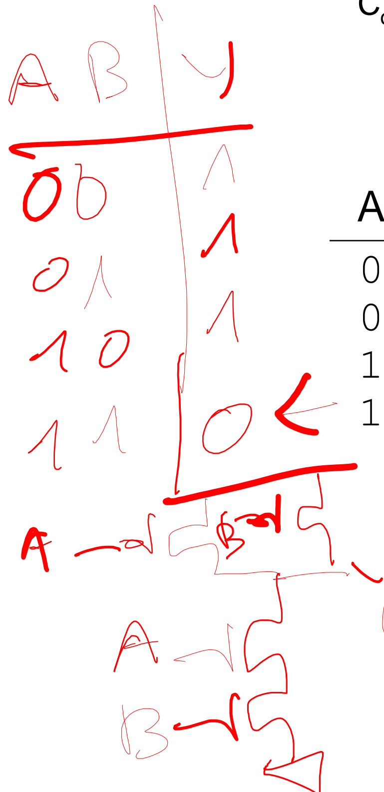
Halb-addierer



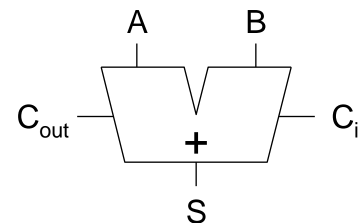
A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S = A \oplus B$

$C_{out} = AB$



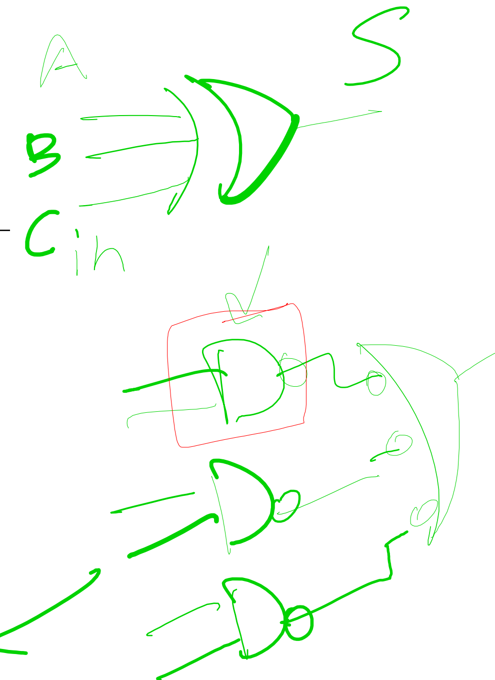
Voll-addierer



C_{in}	A	B	C_{out}	S
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

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

$C_{out} = AB + AC_{in} + BC_{in}$

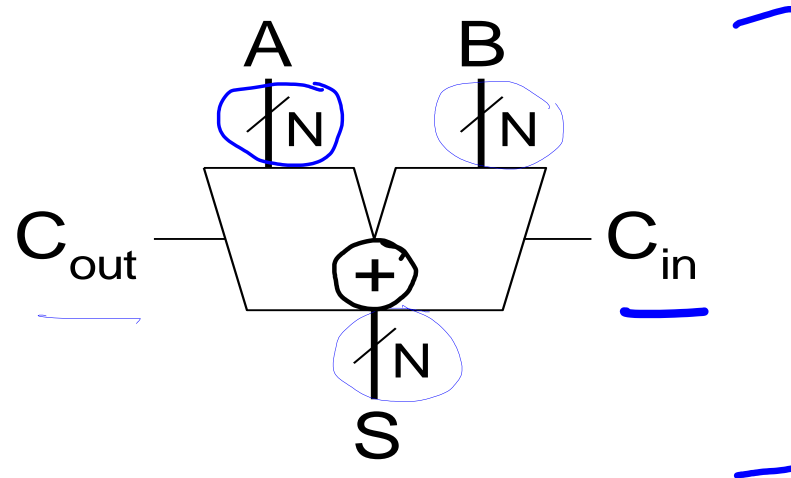


Mehrbit-Addierer mit Weitergabe von Überträgen

TECHNISCHE UNIVERSITÄT DARMSTADT

Carry-propagate adder (CPA)

Schaltsymbol



Embedded Systems & Applications

Mehrbit-Addierer mit Weitergabe von Überträgen

TECHNISCHE UNIVERSITÄT DARMSTADT

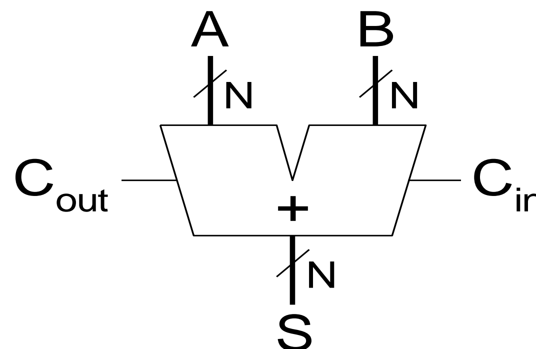
Carry-propagate adder (CPA)

- Verschiedene Typen
  - Ripple-carry-Addierer (langsam)
  - Carry-Lookahead Addierer (schnell)
  - Prefix-Addierer (noch schneller)

Einfach → kompliziert

Carry-Lookahead und Prefix-Addierer sind schneller bei breiteren Datenworten
- Benötigen aber auch mehr Fläche

Schaltsymbol



Embedded Systems & Applications

Mehrbit-Addierer mit Weitergabe von Überträgen

TECHNISCHE UNIVERSITÄT DARMSTADT

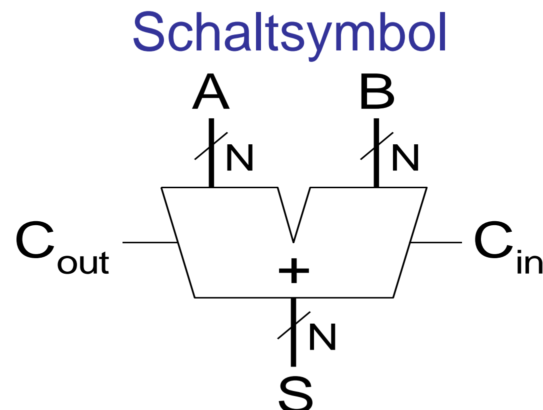
Carry-propagate adder (CPA)

- Verschiedene Typen
  - **Ripple-carry-Addierer** **(langsam)** ←
  - Carry-Lookahead Addierer (schnell)
  - Prefix-Addierer (noch schneller)

Carry-Lookahead und Prefix-Addierer sind schneller bei breiteren Datenworten

- Benötigen aber auch mehr Fläche

Schaltsymbol

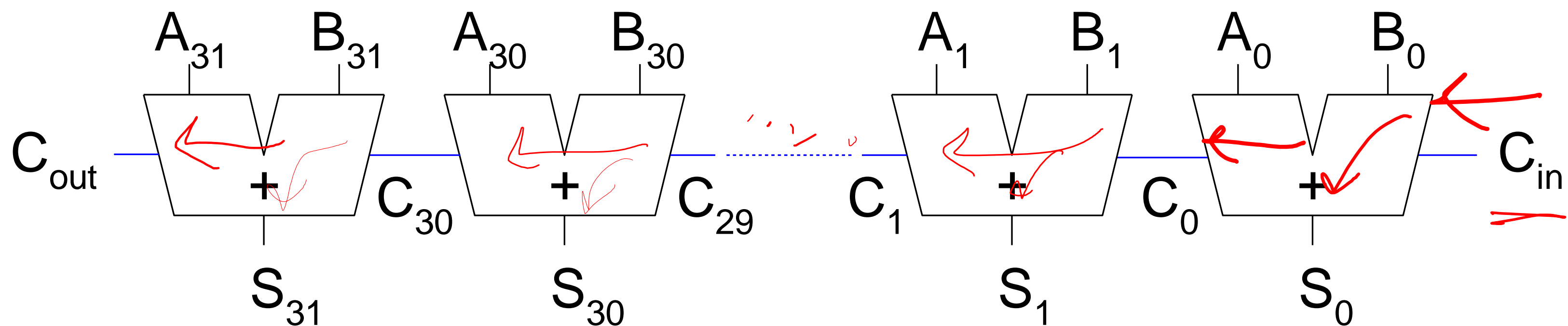


Embedded Systems & Applications

Ripple-Carry-Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- Kette von 1-bit Addierern
- Überträge werden von niedrigen zu hohen Bits weitergegeben
  - Rippeln sich durch die Schaltung
- Nachteil: **Langsam**



Embedded Systems & Applications

Verzögerung durch Ripple-Carry-Addierer

Verzögerung durch einen *N*-bit Ripple-Carry-Addierer ist:

$$t_{\text{ripple}} = N\, t_{FA}$$

t_{FA} ist die Verzögerung durch einen Volladdierer

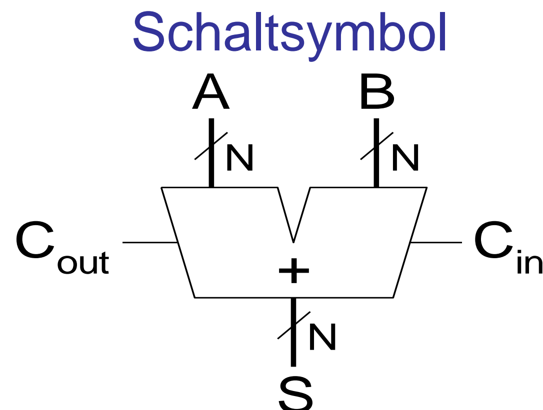
Mehrbit-Addierer mit Weitergabe von Überträgen

Carry-propagate adder (CPA)

- Verschiedene Typen
  - Ripple-carry-Addierer (langsam)
  - **Carry-Lookahead Addierer (schnell)**
  - Prefix-Addierer (noch schneller)

Carry-Lookahead und Prefix-Addierer sind schneller bei breiteren Datenworten

- Benötigen aber auch mehr Fläche



Carry-Lookahead-Addierer (CLA)

- Überträge nicht mehr von Bit-zu-Bit
- Stattdessen: Berechne Übertrag C_{out} aus Block von k Bits

Carry-Lookahead-Addierer (CLA)

- **Überträge nicht mehr von Bit-zu-Bit**
- **Stattdessen: Berechne Übertrag C_{out} aus Block von *k* Bits**
  - Nun zwei Signale
    - ***Generate*** (erzeuge neuen Übertrag)
    - ***Propagate*** (leite eventuellen Übertrag weiter)

Embedded Systems & Applications

Carry-Lookahead-Addierer (CLA)

- **Überträge nicht mehr von Bit-zu-Bit**
- **Stattdessen: Berechne Übertrag C_{out} aus Block von k Bits**
  - Nun zwei Signale
    - ***Generate*** (erzeuge neuen Übertrag)
    - ***Propagate*** (leite eventuellen Übertrag weiter)
- **Bits werden in Spalten organisiert**
  - Haben wir eben beim Ripple-Carry-Addierer auch schon gemacht
  - War aber nicht spannend: Es gab nur eine Zeile
  - … ändert sich jetzt

Carry-Lookahead-Addierer: Definitionen

- **Eine Spalte *i* erzeugt** (**g**enerates) einen Übertrag falls A_i und B_i beide 1 sind.

$$G_i = A_i B_i$$

- **Eine Spalte leitet einen eingehenden Übertrag weiter** falls A_i oder B_i 1 ist (weitergeleitet = **p**ropagate)

$$P_i = A_i + B_i$$

Embedded Systems & Applications

Carry-Lookahead-Addierer: Definitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Eine Spalte *i* erzeugt** (generates) einen Übertrag falls A_i und B_i beide 1 sind.

$$G_i = A_i B_i$$

- **Eine Spalte leitet einen eingehenden Übertrag weiter** falls A_i oder B_i 1 ist (weitergeleitet = propagate)

$$P_i = A_i + B_i$$

- **Eine Spalte (Bit *i*) produziert einen Übertrag an ihrem Ausgang C_i**
  - Wenn sie den Übertrag selbst erzeugt (*Generate*, G_i) oder
  - Wenn sie einen von C_{i-1} eingehenden Übertrag weiterleitet (*Propagate*, P_i)

Embedded Systems & Applications

Carry-Lookahead-Addierer: Definitionen

- **Eine Spalte *i* erzeugt** (**g**enerates) einen Übertrag falls A_i und B_i beide 1 sind.

$$G_i = A_i B_i$$

- **Eine Spalte leitet einen eingehenden Übertrag weiter** falls A_i oder B_i 1 ist (weitergeleitet = **p**ropagate)

$$P_i = A_i + B_i$$

- **Eine Spalte (Bit *i*) produziert einen Übertrag an ihrem Ausgang C_i**
  - Wenn sie den Übertrag selbst erzeugt (*Generate, G_i*) oder
  - Wenn sie einen von C_{i-1} eingehenden Übertrag weiterleitet (*Propagate, P_i*)

- Damit ist der Übertrag C_i aus der Spalte *i* heraus

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$

Embedded Systems & Applications

$N = 4$

Carry Spalte

	3	2	1	0	$-1 \in C_{in}$	C_i
	(1)	1	1	0		
		1	0	1	0	A_i
+		0	1	1	0	B_i
		0	0	0	0	S_i ✓
		0	0	1	0	G_i
		1	1	1	0	P_i
Spalte		3	2	1	0	

-6
6
0

2 Komplement

Überlauf? Nein

0101
0110

10
+ 6
16

Überlauf

$P_i = A_i + B_i$

$G_i = A_i B_i$

$C_0 = G_0 + P_0 C_{in}$
$= 0 + 0 \cdot 0$
$= 0$ ✓

$C_{in} = C_{-1}$

$C_1 = G_1 + P_1 C_0$
$= 1 + 1 \cdot 0$
$= 1$

erzeugt

$C_2 = G_2 + P_2 C_1$
$= 0 + 1 \cdot 1$
$= 1$

$C_3 = \ldots\ldots$
$0 + 1 \cdot 1$

weitergeleitet

Addition im Carry-Lookahead-Verfahren

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Schritt 1:** Berechne G und P-Signale für einzelne Spalten (Einzelbits)
- **Schritt 2:** Berechne G und P Signale für Gruppen von *k* Spalten (*k* Bits)
- **Schritt 3:** Leite C_{in} nun **nicht** einzelbitweise, sondern in *k*-Bit Sprüngen weiter
  - Jeweils durch einen *k*-bit Propagate/Generate-Block

Embedded Systems & Applications

Addition im Carry-Lookahead-Verfahren

- **Schritt 1:** Berechne G und P-Signale für einzelne Spalten (Einzelbits)

$$G_i = A_i B_i$$

$$P_i = A_i + B_i$$

Addition im Carry-Lookahead-Verfahren

- Schritt 1: Berechne G und P-Signale für einzelne Spalten (Einzelbits)
- **Schritt 2: Berechne G und P Signale für Gruppen von *k* Spalten (*k* Bits)**
- Schritt 3: Leite C_{in} nun nicht einzelbitweise, sondern in *k*-Bit Sprüngen weiter
  - Jeweils durch einen *k*-bit Propagate/Generate-Block

Embedded Systems & Applications

Beispiel: Carry-Lookahead Addierer

Schritt 2: Berechne G und P Signale für Gruppen von *k* Spalten (*k* Bits)

- Bestimme $P_{3:0}$ und $G_{3:0}$ Signale für einen 4b Block
- Überlegung: Der 4b Block leitet einen Übertrag direkt weiter wann?

TECHNISCHE UNIVERSITÄT DARMSTADT
Embedded Systems & Applications

$c_{in} = 1$

$$
\begin{array}{r}
10100 \\
+\ 0110 \\
\hline
1
\end{array}
$$

$P_0 = 0$

$P_3 \cdot P_2 \cdot P_1 \cdot P_0$

$P_{3:0} = ?\ 0$

$G_{3:0} = ?\ 1$

$P_3 = 1\quad P_2 = 1\quad P_1 = 1\quad P_0 = 1$

$P_{3:0} = 1$

$$
\begin{array}{r}
1011 \\
0100 \\
\hline
\end{array}
$$

A_i

B_i

$P_{3:0} = 0$

$P_1 = 0\quad P_0 = 1$

Beispiel: Carry-Lookahead Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

Schritt 2: Berechne G und P Signale für Gruppen von *k* Spalten (*k* Bits)

- Bestimme $P_{3:0}$ und $G_{3:0}$ Signale für einen 4b Block
- Überlegung: Der 4b Block leitet einen Übertrag direkt weiter
  - … wenn alle Spalten den Übertrag weiterleiten

$$P_{3:0} = P_3 P_2 P_1 P_0$$

Beispiel: Carry-Lookahead Addierer

Schritt 2: Berechne G und P Signale für Gruppen von *k* Spalten (*k* Bits)

- Bestimme $P_{3:0}$ und $G_{3:0}$ Signale für einen 4b Block
- Überlegung: Der 4b Block leitet einen Übertrag direkt weiter
  - … wenn alle Spalten den Übertrag weiterleiten

1-bit + $P_{3:0} = P_3 P_2 P_1 P_0$

- Überlegung: 4b Block erzeugt Übertrag wann?

1-bit $G_{3:0} = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3$

$= G_3 + P_3 [G_2 + P_2 (G_1 + G_0 P_1)]$

Beispiel: Carry-Lookahead Addierer

Schritt 2: Berechne G und P Signale für Gruppen von *k* Spalten (*k* Bits)

- Bestimme $P_{3:0}$ und $G_{3:0}$ Signale für einen 4b Block
- Überlegung: Der 4b Block leitet einen Übertrag direkt weiter
  - … wenn alle Spalten den Übertrag weiterleiten

$$P_{3:0} = P_3 P_2 P_1 P_0$$

- Überlegung: 4b Block erzeugt Übertrag wenn
  - … Spalte 3 einen Übertrag erzeugt (G_3=1) oder
  - … Spalte 3 einen Übertrag weiterleitet (P_3=1), der vorher erzeugt wurde

$$G_{3:0} = G_3 + P_3 [G_2 + P_2 (G_1 + P_1 G_0)]$$

Beispiel: Carry-Lookahead Addierer

Schritt 2: Berechne G und P Signale für Gruppen von *k* Spalten (*k* Bits)

- Bestimme $P_{3:0}$ und $G_{3:0}$ Signale für einen 4b Block
- Überlegung: Der 4b Block leitet einen Übertrag direkt weiter
  - … wenn alle Spalten den Übertrag weiterleiten

$$P_{3:0} = P_3 P_2 P_1 P_0$$

- Überlegung: 4b Block erzeugt Übertrag wenn
  - … Spalte 3 einen Übertrag erzeugt (G_3=1) oder
  - … Spalte 3 einen Übertrag weiterleitet (P_3=1), der vorher erzeugt wurde

$$G_{3:0} = G_3 + P_3 [G_2 + P_2 (G_1 + P_1 G_0)]$$

- Damit ist der Übertrag durch einen *i:j* Bit breiten Block C_i

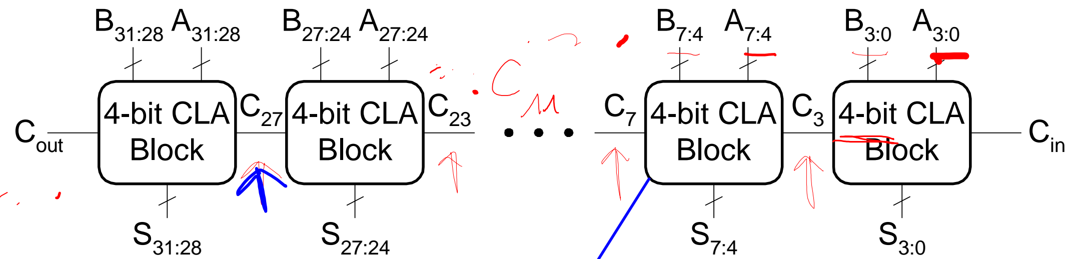
$$C_i = G_{i:j} + P_{i:j} C_{j-1}$$

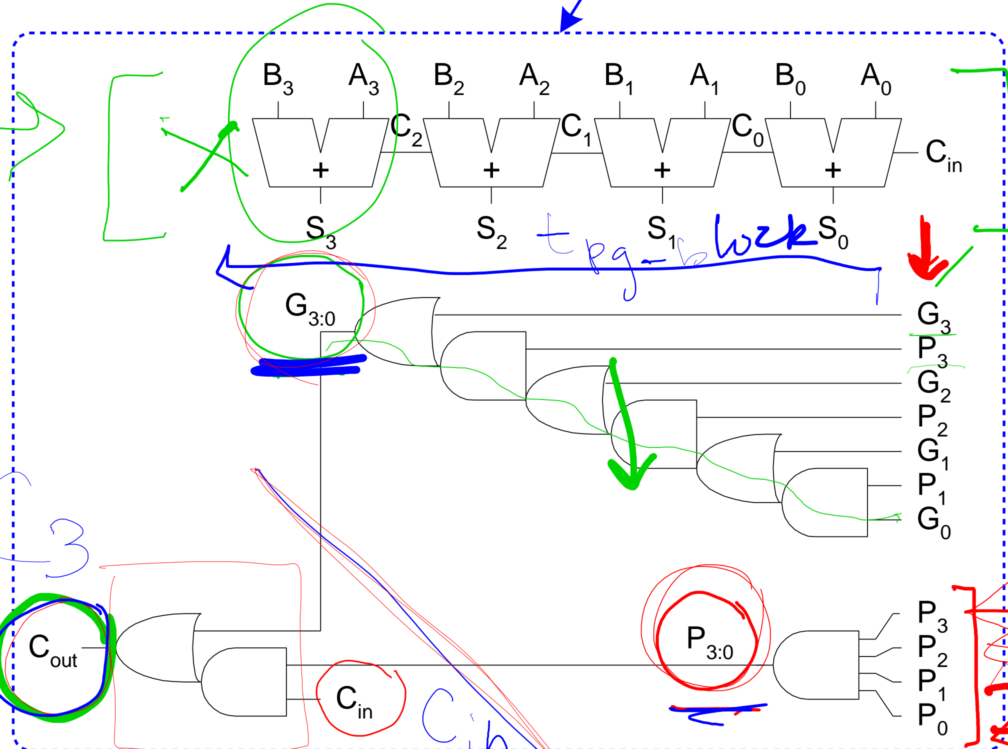
Addition im Carry-Lookahead-Verfahren

- Schritt 1: Berechne G und P-Signale für einzelne Spalten (Einzelbits)
- Schritt 2: Berechne G und P Signale für Gruppen von *k* Spalten (*k* Bits) k=4
- **Schritt 3: Leite C_{in} nun nicht einzelbitweise, sondern in *k*-Bit Sprüngen weiter**
  - Jeweils durch einen *k*-bit Propagate/Generate-Block

Embedded Systems & Applications

32-bit CLA mit 4b Blöcken

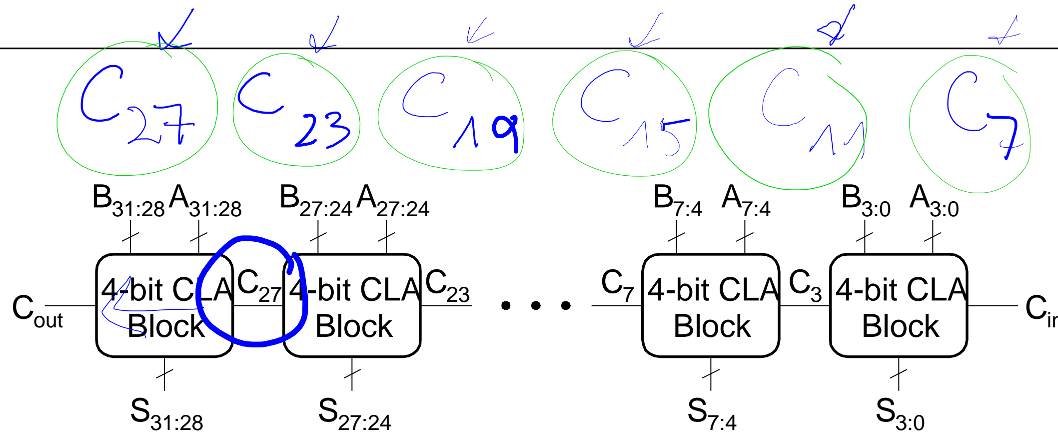


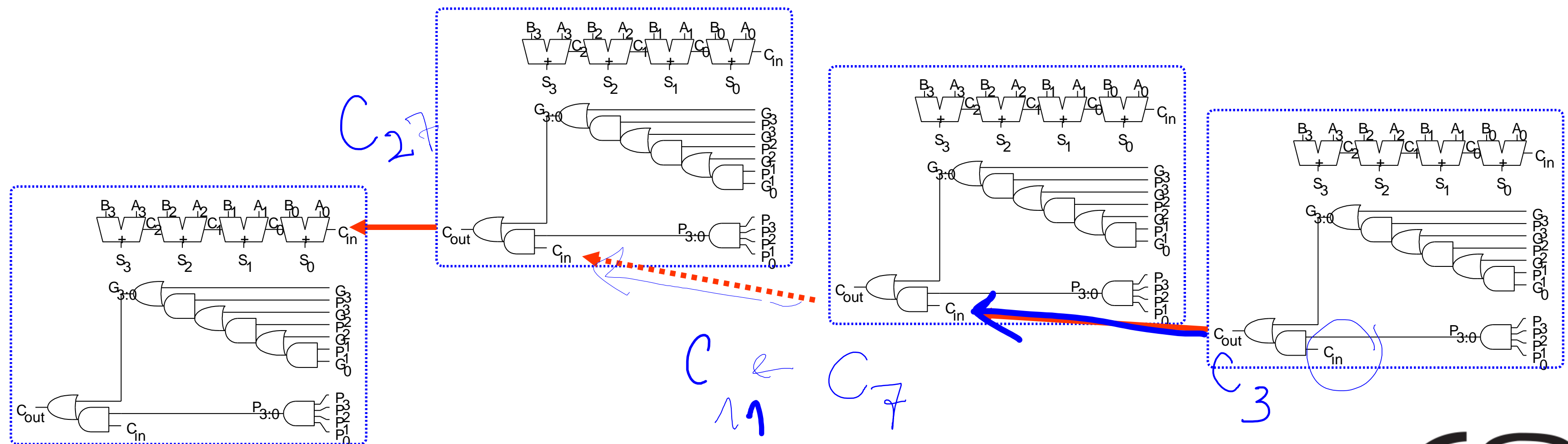


$G_i = A_i B_i$

$P_i = A_i + B_i$

32-bit CLA mit 4b Blöcken





Carry-Lookahead Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- Verzögerung durch *N*-bit carry-lookahead Addierer mit *k*-Bit Blöcken

$$t_{CLA} = t_{pg} + t_{pg_block} + (N/k - 1)\, t_{AND_OR} + k\, t_{FA}$$

wobei

- t_{pg} : Verzögerung P, G Berechnung für eine Spalte (ganz rechts)
- t_{pg_block}: Verzögerung P, G Berechnung für einen Block (rechts)
- t_{AND_OR}: Verzögerung durch AND/OR je *k*-Bit CLA Block ("Weiche")
- $k\, t_{FA}$: Verzögerung zur Berechnung der k höchstwertigen Summenbits

- Für N > 16 ist ein CLA oftmals schneller als ein Ripple-Carry-Addierer
- Aber: Verzögerung hängt immer noch von N ab
  - Im wesentlichen linear

Präfix-Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- Nun nicht mehr N / k Stufen
- Sondern $\log_2 N$ Stufen
  - Breite der Operanden geht also nur noch logarithmisch in Verzögerung ein
- Allerdings: Sehr viel Hardware erforderlich!

Embedded Systems & Applications

Präfix-Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- Führt Ideen des CLA weiter
- Berechnet den Übertrag C_{i-1} in jede Spalte i so schnell wie möglich
- Bestimmt damit die Summe jeder Spalte

$$S_i = (A_i \oplus B_i) \oplus C_{i-1}$$

- Vorgehen zur schnellen Berechnung aller C_i
  - Berechne P und G für größer werdende Blöcke
    - 1b, 2b, 4b, 8b, …
    - Bis die Eingangsüberträge für alle Spalten bereitstehen

Embedded Systems & Applications

Präfix-Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ein Übertrag wird entweder
  - … in einer Spalte *i* generiert
  - … oder aus einer Vorgängerspalte *i*-1 propagiert

Embedded Systems & Applications

Präfix-Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ein Übertrag wird entweder
  - … in einer Spalte *i* generiert
  - … oder aus einer Vorgängerspalte *i*-1 propagiert
- Definition: Eingangsübertrag C_{in} in den ganzen Addierer kommt aus Spalte -1

$$G_{-1} = C_{in},\ P_{-1} = 1$$

- Eingangsübertrag in eine Spalte *i* ist Ausgangsübertrag C_{i-1} der Spalte *i*-1

$$C_{i-1} = G_{i-1:-1}$$

$G_{i-1:-1}$ ist das Generate-Signal von Spalte -1 bis Spalte *i-1*

$C_3 = G_{3:-1}$ ← Präfix

Interpretation: Ein Ausgangsübertrag aus Spalte *i*-1 entsteht

- … wenn der Block *i*-1:-1 selbst Übertrag in *i*-1 generiert oder aus *i*-2,3, … weiterleitet

Embedded Systems & Applications

Präfix-Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- Damit Summenformel für Spalte *i* umschreibbar zu

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

- **Deshalb nun Ziel der Hardware-Realisierung:**
  - Bestimme so schnell wie möglich:

    $G_{0:-1}, G_{1:-1}, G_{2:-1}, G_{3:-1}, G_{4:-1}, G_{5:-1}, \ldots$
  - Sogenannte Präfixe

Embedded Systems & Applications

Präfix-Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Berechnung von P und G für variabel großen Block**
  - Höchstwertiges Bit: *i*
  - Niederwertiges Bit: *j*
  - Unterteilt in zwei Teilblöcke (*i*:*k*) und (*k*-1:*j*)

- Für einen Block *i*:*j*

$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

$$P_{i:j} = P_{i:k} P_{k-1:j}$$

Embedded Systems & Applications

Präfix-Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Berechnung von P und G für variabel großen Block**
  - Höchstwertiges Bit: *i*
  - Niederwertiges Bit: *j*
  - Unterteilt in zwei Teilblöcke (*i*:*k*) und (*k*-1:*j*)

- Für einen Block *i*:*j*

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j}$$
$$P_{i:j} = P_{i:k} P_{k-1:j}$$

- **Bedeutung**
  - G: Ein Block erzeugt einen Ausgabeübertrag, falls
    - … in seinem oberen Teil (*i*:*k*) ein Übertrag erzeugt wird oder
    - … der obere Teil einen Übertrag weiterleitet, der im unteren Teil (*k*-1:*j*) erzeugt wurde
  - P: Ein Block leitet einen Eingabeübertrag als Ausgabeübertrag weiter, falls
    - Sowohl der untere als auch der obere Teil den Übertrag weiterleiten

Embedded Systems & Applications

3	2	1	0	-1
	1	1		
	0	1	1	0
	1	0	1	0
			0	0
1	0	0		
	0	0	1	0
	1	1	1	0
			1	0
	3	2		

$G_i \cdot P_i$

$G_{0:0} = G_0$

$G_{-1:-1} = G_{-1} = C_{in}$

$C_0 = G_{0:-1} = G_{0:0} + P_{0:0} G_{-1:-1}$

$= 0 + 0 \cdot 0$

$= 0$ ✓

$C_1 = G_{1:-1} = G_{1:1} + P_{1:1} G_{0:-1}$

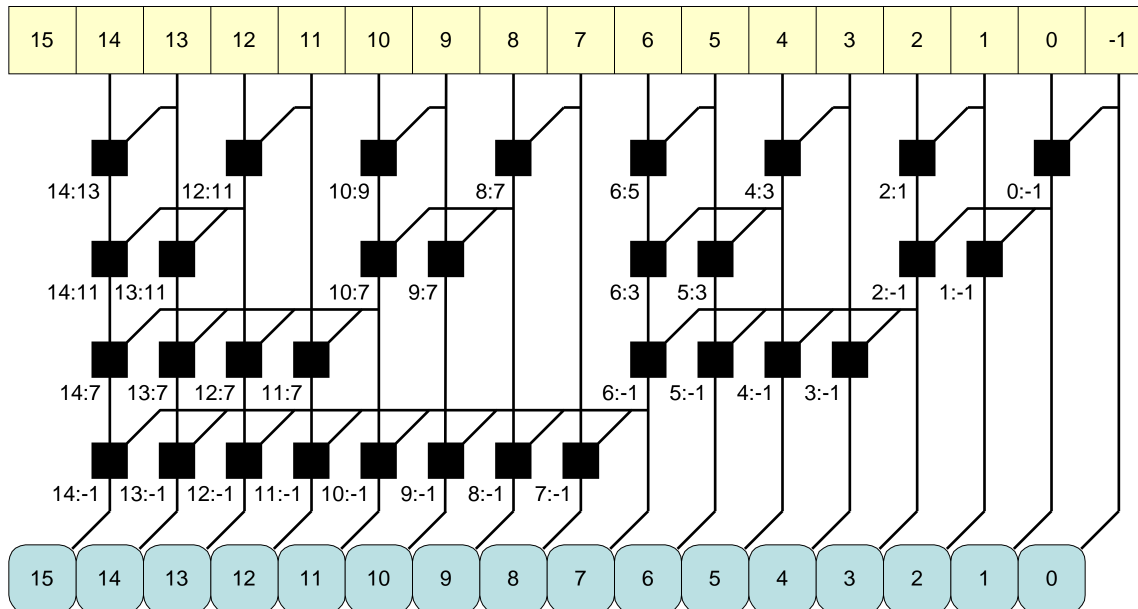
$= 1 + 1 \cdot 0$

$= 1$ ✓

Aufbau eines Präfix-Addierers

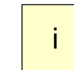
$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k\text{-}1:j}$$

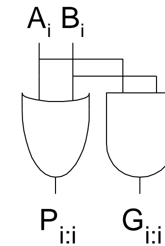
$$P_{i:j} = P_{i:k} P_{k\text{-}1:j}$$

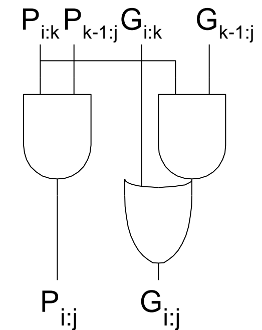


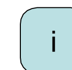
$$S_i = (A_i \oplus B_i) \oplus G_{i\text{-}1:\text{-}1}$$

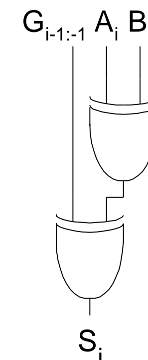
Legende











Embedded Systems & Applications

Themen für Heute

TECHNISCHE UNIVERSITÄT DARMSTADT

- Präfix Addierer
- Subtrahierer
- Vergleicher
- ALU
- Shiftern
- Multiplizierer
- Dividierer

Präfix-Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- Damit Summenformel für Spalte *i* umschreibbar zu

$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

- **Deshalb nun Ziel der Hardware-Realisierung:**
  - Bestimme so schnell wie möglich:

    $G_{0:-1}, G_{1:-1}, G_{2:-1}, G_{3:-1}, G_{4:-1}, G_{5:-1}, \ldots$
  - Sogenannte Präfixe

Embedded Systems & Applications

Präfix-Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Berechnung von P und G für variabel großen Block**
  - Höchstwertiges Bit: i
  - Niederwertiges Bit: j
  - Unterteilt in zwei Teilblöcke (i:k) und (k-1:j)

- Für einen Block i:j

$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

$$P_{i:j} = P_{i:k} P_{k-1:j}$$

Embedded Systems & Applications

Präfix-Addierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Berechnung von P und G für variabel großen Block**
  - Höchstwertiges Bit: i
  - Niederwertiges Bit: j
  - Unterteilt in zwei Teilblöcke (i:k) und (k-1:j)

- Für einen Block i:j

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j}$$

$$P_{i:j} = P_{i:k} P_{k-1:j}$$

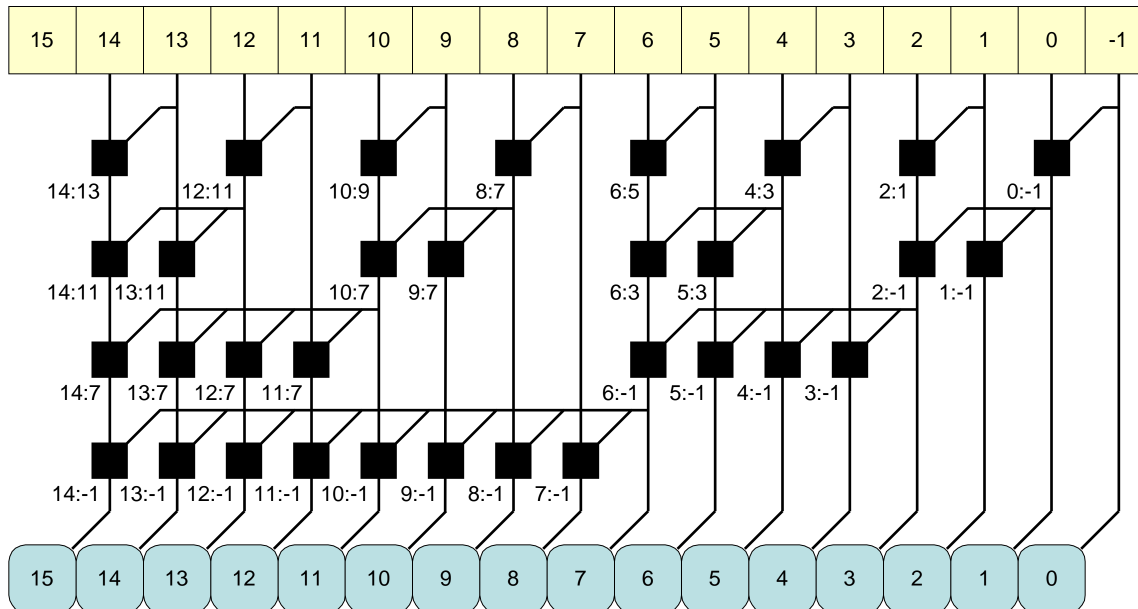
- **Bedeutung**
  - Ein Block erzeugt einen Ausgabeübertrag, falls
    - … in seinem oberen Teil (i:k) ein Übertrag erzeugt wird oder
    - … der obere Teil einen Übertrag weiterleitet, der im unteren Teil (k-1:j) erzeugt wurde
  - Ein Block leitet einen Eingabeübertrag als Ausgabeübertrag weiter, falls
    - Sowohl der untere als auch der obere Teil den Übertrag weiterleiten

Aufbau eines Präfix-Addierers

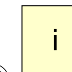
16-bit

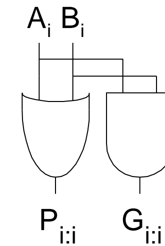
$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

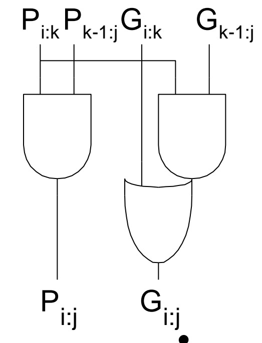
$$P_{i:j} = P_{i:k} P_{k-1:j}$$

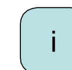


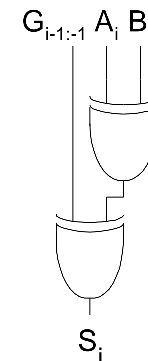
Legende









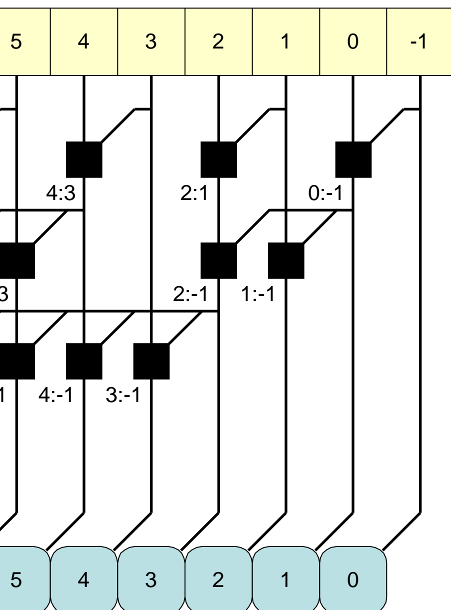


$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

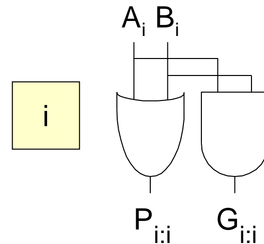
TECHNISCHE UNIVERSITÄT DARMSTADT

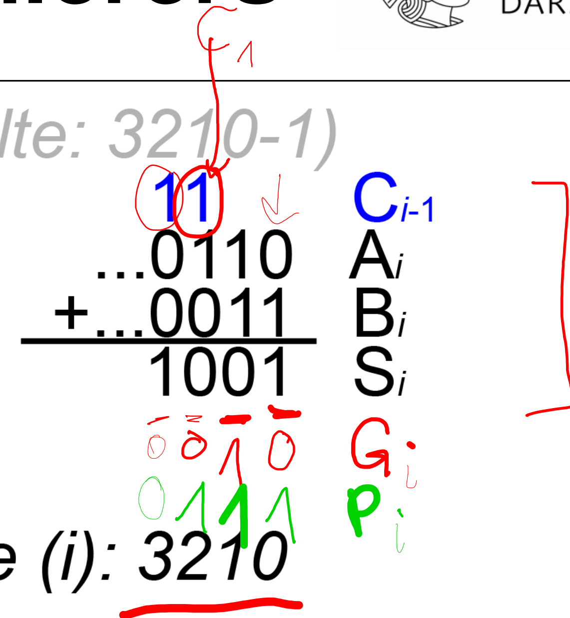
Embedded Systems & Applications

Aufbau eines Präfix-Addierers



i





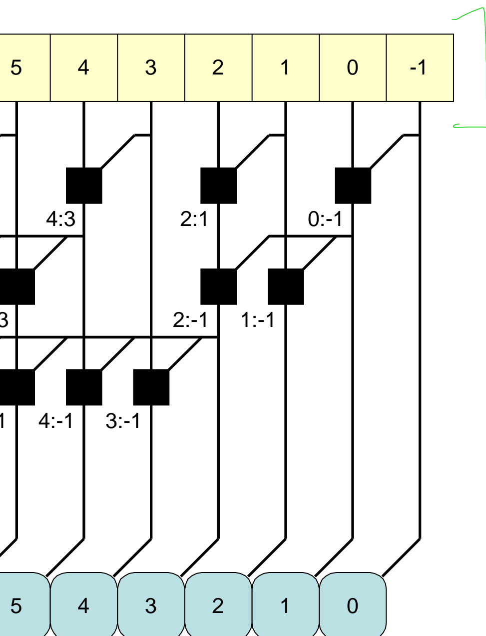
Embedded Systems & Applications

Aufbau eines Präfix-Addierers

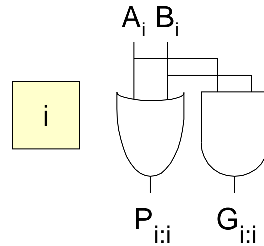
(carry Spalte: 3210-1)

	11	C_{i-1}
	...0110	A_i
+	...0011	B_i
	1001	S_i
	0010	$G_{i:i}$
	0111	$P_{i:i}$

Spalte (i): 3210



i



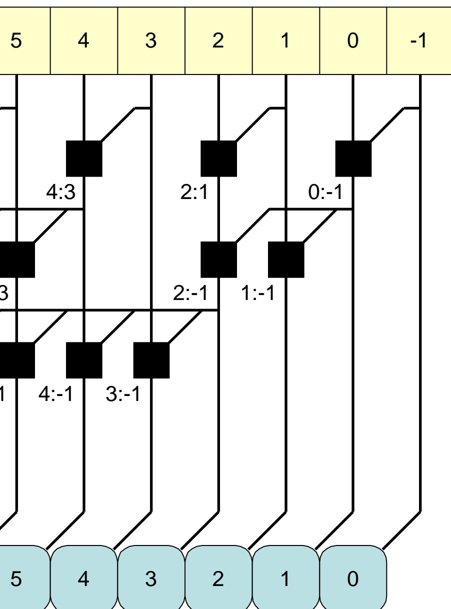
Embedded Systems & Applications

Aufbau eines Präfix-Addierers

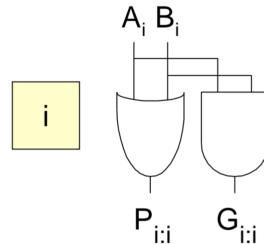
(carry Spalte: 3210-1)

11	C_{i-1}
...0110	A_i
+...0011	B_i
1001	S_i
0010	$G_{i:i}$
0111	$P_{i:i}$

Spalte (i): 3210



i

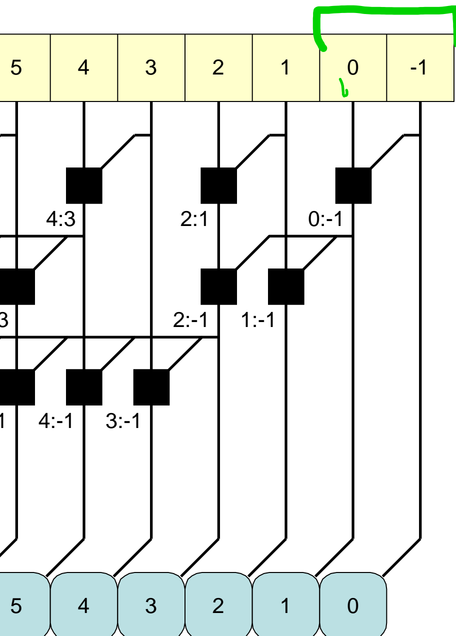


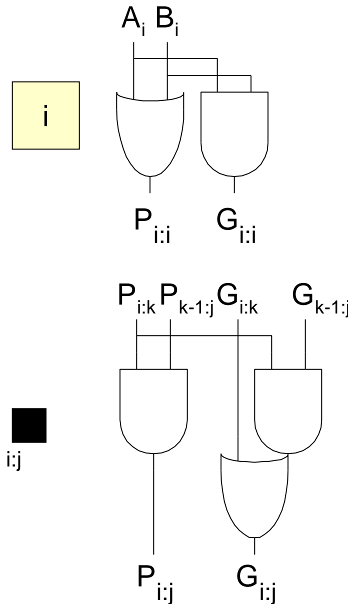
Embedded Systems & Applications

Aufbau eines Präfix-Addierers

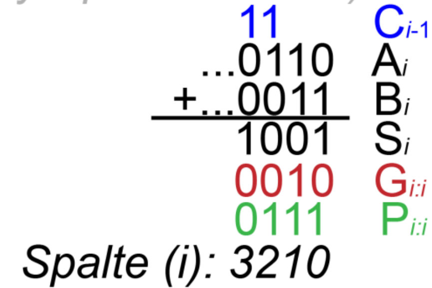
$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k\text{-}1:j}$$

$$P_{i:j} = P_{i:k} P_{k\text{-}1:j}$$





(carry Spalte: 3210-1)



Embedded Systems & Applications

Aufbau eines Präfix-Addierers

TECHNISCHE UNIVERSITÄT DARMSTADT

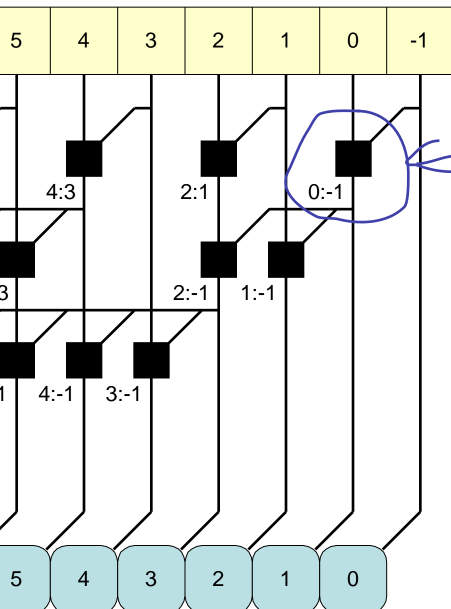
$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j}$$

$$P_{i:j} = P_{i:k} P_{k-1:j}$$

(carry Spalte: 3210-1)

	11	C_{i-1}
	...0110	A_i
+	...0011	B_i
	1001	S_i
	0010	$G_{i:i}$
	0111	$P_{i:i}$

Spalte (i): 3210

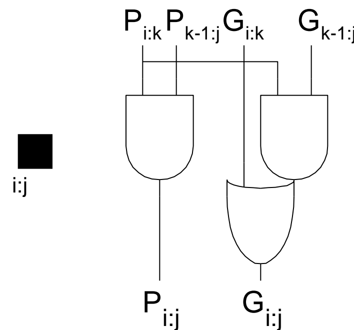


Block 0:-1 $G_{0:-1} = G_{0:0} + P_{0:0} G_{-1:-1}$

$= 0 + 1 * C_{in} = 0$

$C_{in} = 0$

i:j

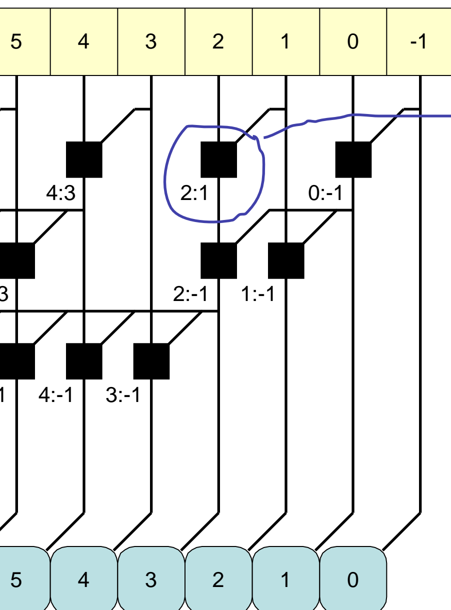


Embedded Systems & Applications

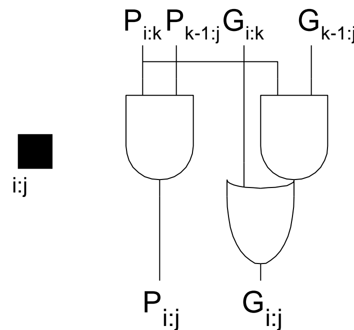
Aufbau eines Präfix-Addierers

$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

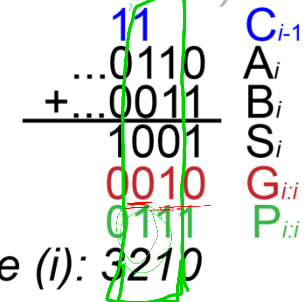
$$P_{i:j} = P_{i:k} P_{k-1:j}$$



i:j



(carry Spalte: 3210-1)



C_{i-1}
A_i
B_i
S_i
$G_{i:i}$
$P_{i:i}$

Spalte (i): 3210

Block 0:-1 $G_{0:-1} = G_{0:0} + P_{0:0} G_{-1:-1}$

$= 0 + 1 * C_{in} = 0$

Block 2:1 $G_{2:1} = G_{2:2} + P_{2:2} G_{1:1}$

$= 0 + 1 * 1 = 1$

TECHNISCHE UNIVERSITÄT DARMSTADT

Embedded Systems & Applications

Aufbau eines Präfix-Addierers

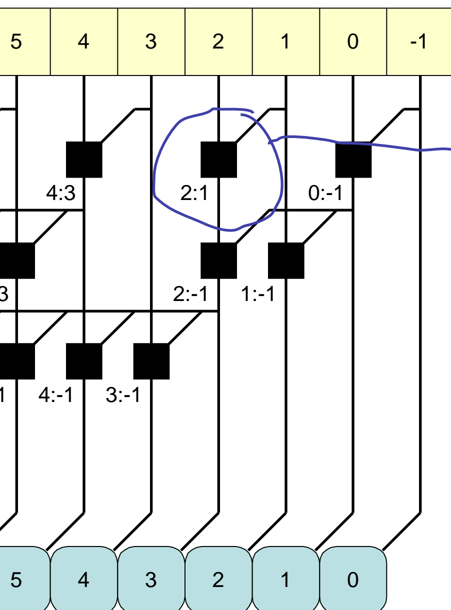
$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

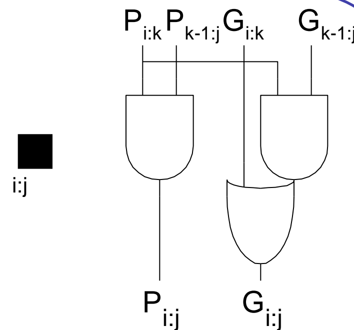
$$P_{i:j} = P_{i:k} P_{k-1:j}$$

(carry Spalte: 3210-1)

	11	C_{i-1}
	...0110	A_i
+	...0011	B_i
	1001	S_i
	0010	$G_{i:i}$
	0111	$P_{i:i}$

Spalte (i): 3210





Block 0:-1 $G_{0:-1} = G_{0:0} + P_{0:0} G_{-1:-1}$

$= 0 + 1 * C_{in} = 0$

Block 2:1 $G_{2:1} = G_{2:2} + P_{2:2} G_{1:1}$

$= 0 + 1 * 1 = 1$

$P_{2:1} = P_{2:2} * P_{1:1}$

$= 1 * 1 = 1$

TECHNISCHE UNIVERSITÄT DARMSTADT

Embedded Systems & Applications

Aufbau eines Präfix-Addierers

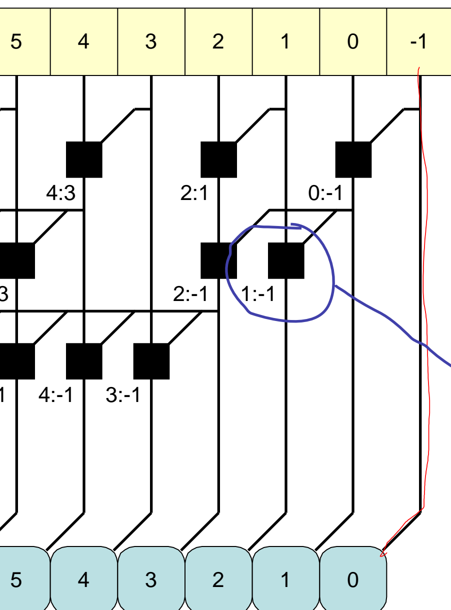
$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

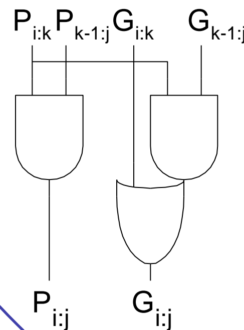
$$P_{i:j} = P_{i:k}P_{k-1:j}$$

(carry Spalte: 3210-1)

11	C_{i-1}
...0110	A_i
+...0011	B_i
1001	S_i
0010	$G_{i:i}$
0111	$P_{i:i}$

Spalte (i): 3210





Block 0:-1 $G_{0:-1} = G_{0:0} + P_{0:0}G_{-1:-1}$

$= 0 + 1 * C_{in} = 0$

Block 2:1 $G_{2:1} = G_{2:2} + P_{2:2}G_{1:1}$

$= 0 + 1 * 1 = 1$

$P_{2:1} = P_{2:2} * P_{1:1}$

$= 1 * 1 = 1$

Block 2:-1 $G_{2:-1} = G_{2:1} + P_{2:1}G_{0:-1}$

$= 1 + 1 * 0 = 1$

Aufbau eines Präfix-Addierers

TECHNISCHE UNIVERSITÄT DARMSTADT

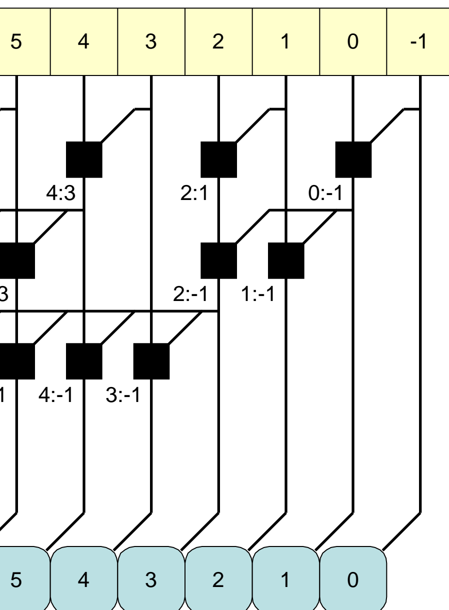
$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

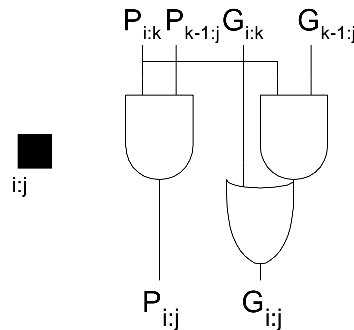
$$P_{i:j} = P_{i:k} P_{k-1:j}$$

(carry Spalte: 3210-1)

11	C_{i-1}
...0110	A_i
+...0011	B_i
1001	S_i
0010	$G_{i:i}$
0111	$P_{i:i}$

Spalte (i): 3210





Block 0:-1 $\mathbf{G_{0:-1}} = G_{0:0} + P_{0:0} G_{-1:-1}$

$= 0 \quad + 1 * C_{in} = 0$

Block 2:1 $\mathbf{G_{2:1}} = G_{2:2} + P_{2:2} G_{1:1}$

$= 0 \quad + 1 * 1 = 1$

$\mathbf{P_{2:1}} = P_{2:2} * P_{1:1}$

$= 1 * 1 = 1$

Block 2:-1 $\mathbf{G_{2:-1}} = G_{2:1} + P_{2:1} G_{0:-1}$

$= 1 \quad + 1 * 0 = 1$

$\mathbf{P_{2:-1}} = P_{2:1} * P_{1:-1}$

$= 1 * 1 = 1$

Aufbau eines Präfix-Addierers

TECHNISCHE UNIVERSITÄT DARMSTADT

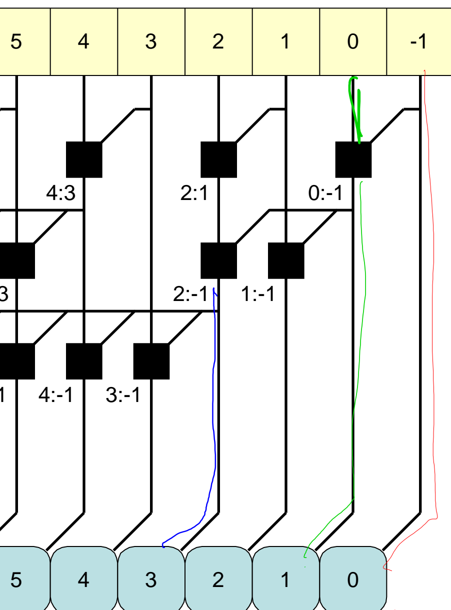
$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

$$P_{i:j} = P_{i:k} P_{k-1:j}$$

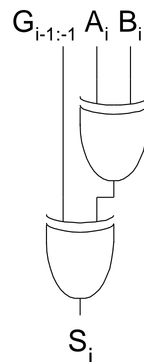
(carry Spalte: 3210-1)

11	C_{i-1}
...0110	A_i
+...0011	B_i
1001	S_i
0010	$G_{i:i}$
0111	$P_{i:i}$

Spalte (i): 3210



i



Block 0:-1 $G_{0:-1} = G_{0:0} + P_{0:0} G_{-1:-1}$

$= 0 + 1 * C_{in} = 0$

Block 2:1 $G_{2:1} = G_{2:2} + P_{2:2} G_{1:1}$

$= 0 + 1 * 1 = 1$

$P_{2:1} = P_{2:2} * P_{1:1}$

$= 1 * 1 = 1$

Block 2:-1 $G_{2:-1} = G_{2:1} + P_{2:1} G_{0:-1}$

$= 1 + 1 * 0 = 1$

$P_{2:-1} = P_{2:1} * P_{1:-1}$

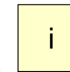
$= 1 * 1 = 1$

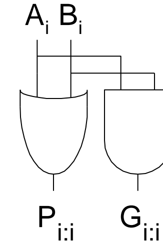
$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

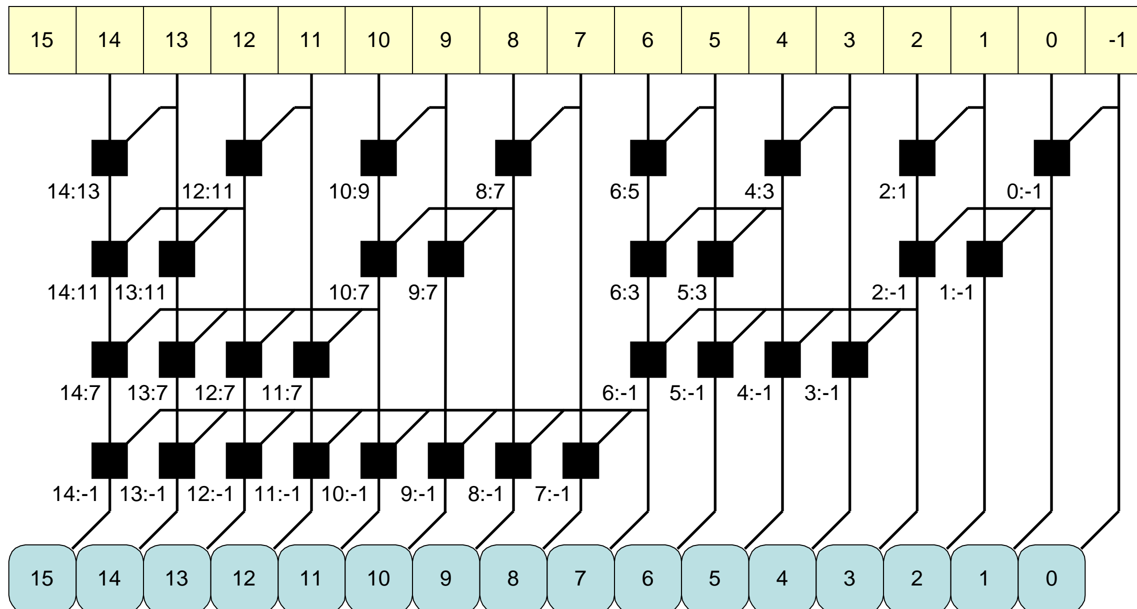
Embedded Systems & Applications

Aufbau eines Präfix-Addierers

Legende







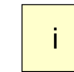
Embedded Systems & Applications

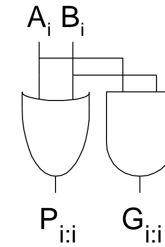
Aufbau eines Präfix-Addierers

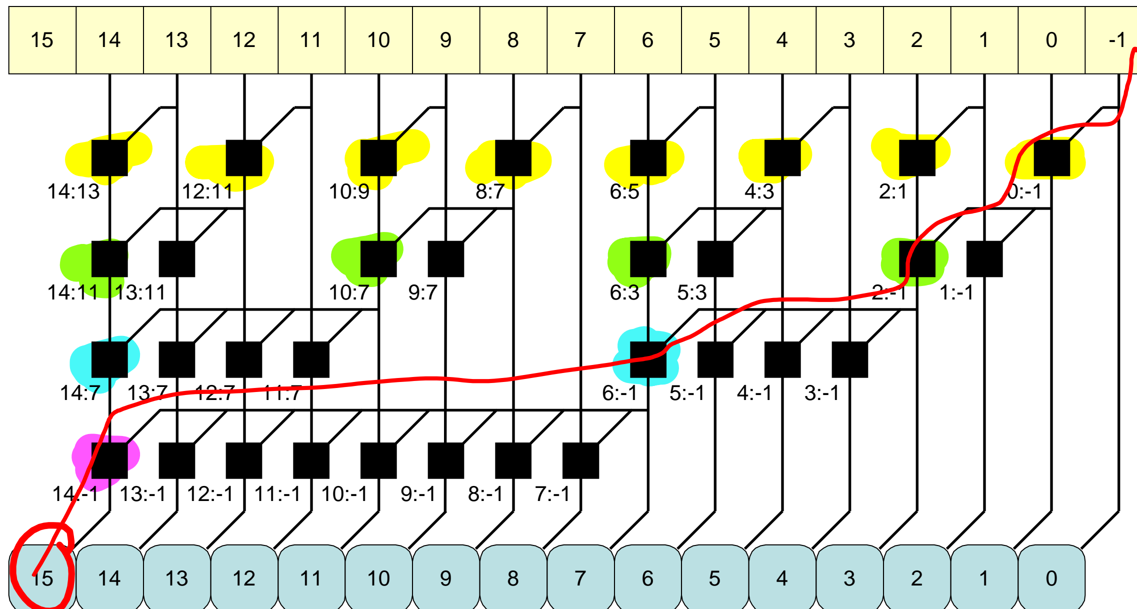
$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

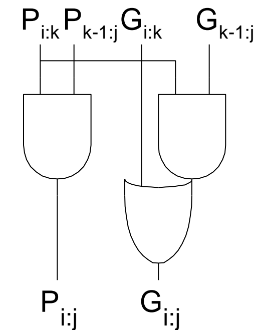
$$P_{i:j} = P_{i:k} P_{k-1:j}$$

Legende









Aufbau eines Präfix-Addierers

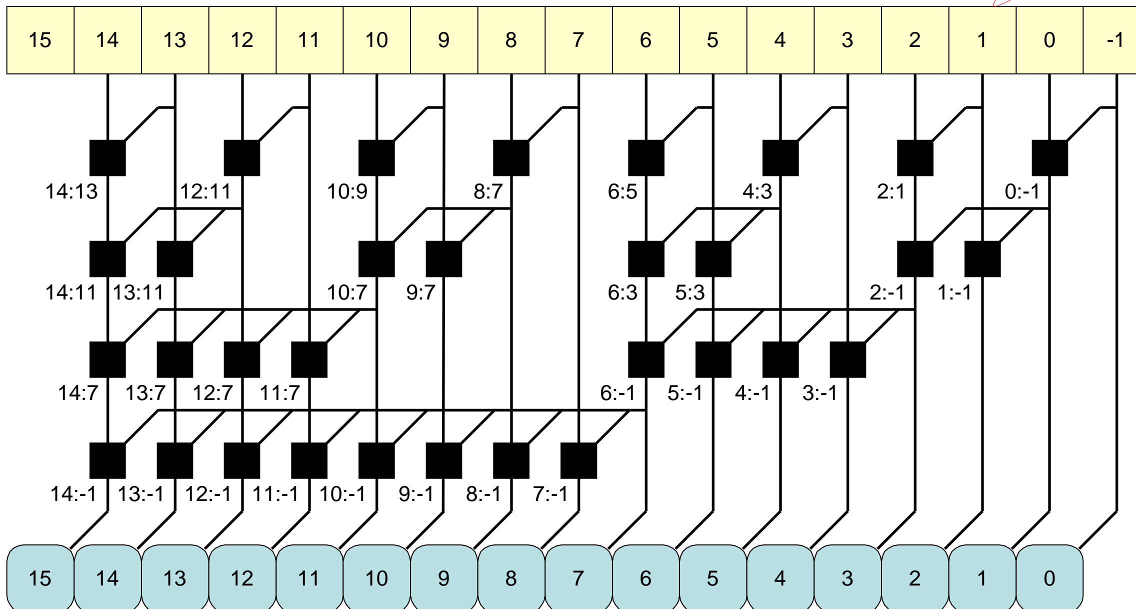
16-Bit

4 Stufen

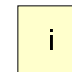
$\log_2 N$

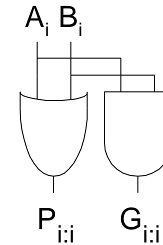
$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

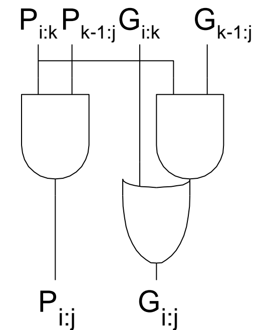
$$P_{i:j} = P_{i:k} P_{k-1:j}$$

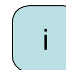


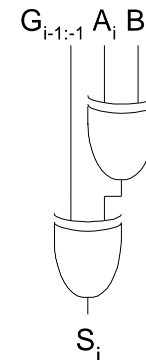
Legende











$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

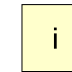
Embedded Systems & Applications

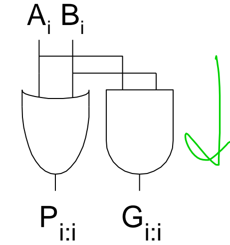
Aufbau eines Präfix-Addierers

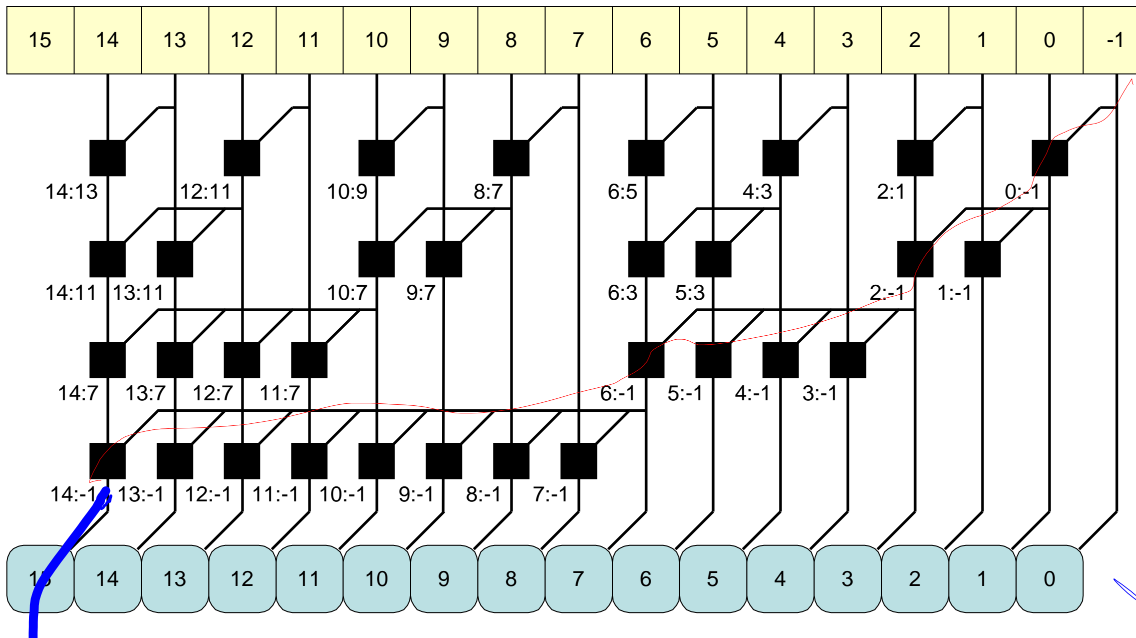
$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

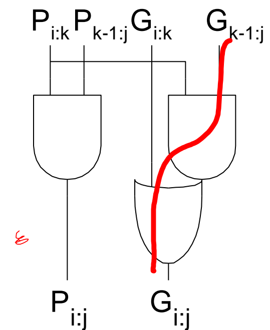
$$P_{i:j} = P_{i:k} P_{k-1:j}$$

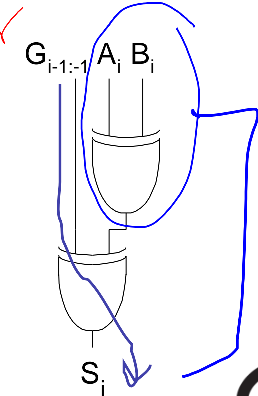
Legende











$$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$$

Embedded Systems & Applications

Verzögerung durch Präfix-Addierer

- **Verzögerung durch einen *N*-bit Präfix-Addierer**

2 gatter

$$t_{PA} = t_{pg} + (\log_2 N)\ t_{pg_prefix} + t_{XOR}$$

wobei

- $\boldsymbol{t_{pg}}$: Verzögerung durch P, G-Berechnung für Spalte *i* (ein AND bzw. OR-Gatter)
- $\boldsymbol{t_{pg_prefix}}$: Verzögerung durch eine Präfix-Stufe (AND-OR Gatter)
- $\boldsymbol{t_{XOR}}$: Verzögerung durch letztes XOR der Summenberechnung

Embedded Systems & Applications

Vergleich von Addiererverzögerungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Szenario: 32b Addition mit, Ripple-Carry, Carry-Lookahead (4-bit Blöcke), Präfix-Addierer
- Verzögerungen von Komponenten
  - Volladdierer $t_{FA} = 300$ps
  - Zwei-Eingangs Gatter $t_{\text{AND}} = t_{\text{OR}} = t_{\text{XOR}} = 100$ps

$$t_{\text{ripple}} = N\, t_{FA}$$
$$=$$

$$t_{CLA} = t_{pg} + t_{pg_\text{block}} + (N/k - 1)\, t_{\text{AND_OR}} + k\, t_{FA}$$
$$=$$

$$t_{PA} = t_{pg} + (\log_2 N)\, t_{\text{pg_prefix}} + t_{\text{XOR}}$$
$$=$$

Embedded Systems & Applications

Vergleich von Addiererverzögerungen

- Szenario: 32b Addition mit, Ripple-Carry, Carry-Lookahead (4-bit Blöcke), Präfix-Addierer
- Verzögerungen von Komponenten
  - Volladdierer t_{FA} = 300ps
  - Zwei-Eingangs Gatter $t_{\mathrm{AND}} = t_{\mathrm{OR}} = t_{\mathrm{XOR}}$ = 100ps

$$t_{\mathrm{ripple}} = N\, t_{FA} = 32\ (300\ \mathrm{ps})$$
$$= \mathbf{9{,}6\ ns}$$

$$t_{CLA} = t_{pg} + t_{pg_\mathrm{block}} + (N/k - 1)\, t_{\mathrm{AND_OR}} + k\, t_{FA}$$
$$= [100 + 600 + (7)\ 200 + 4\ (300)]\ \mathrm{ps}$$
$$= \mathbf{3{,}3\ ns}$$

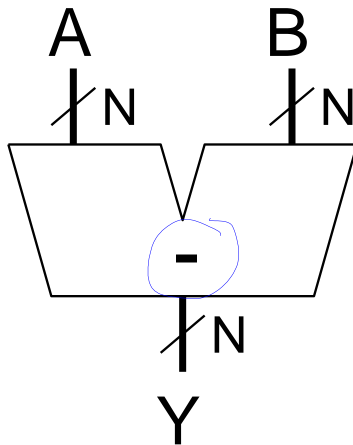
$$t_{PA} = t_{pg} + (\log_2 N)\ t_{\mathrm{pg_prefix}} + t_{\mathrm{XOR}}$$
$$= [100 + (\log_2 32)\ 200 + 100]\ \mathrm{ps}$$
$$= \mathbf{1{,}2\ ns}$$

Embedded Systems & Applications

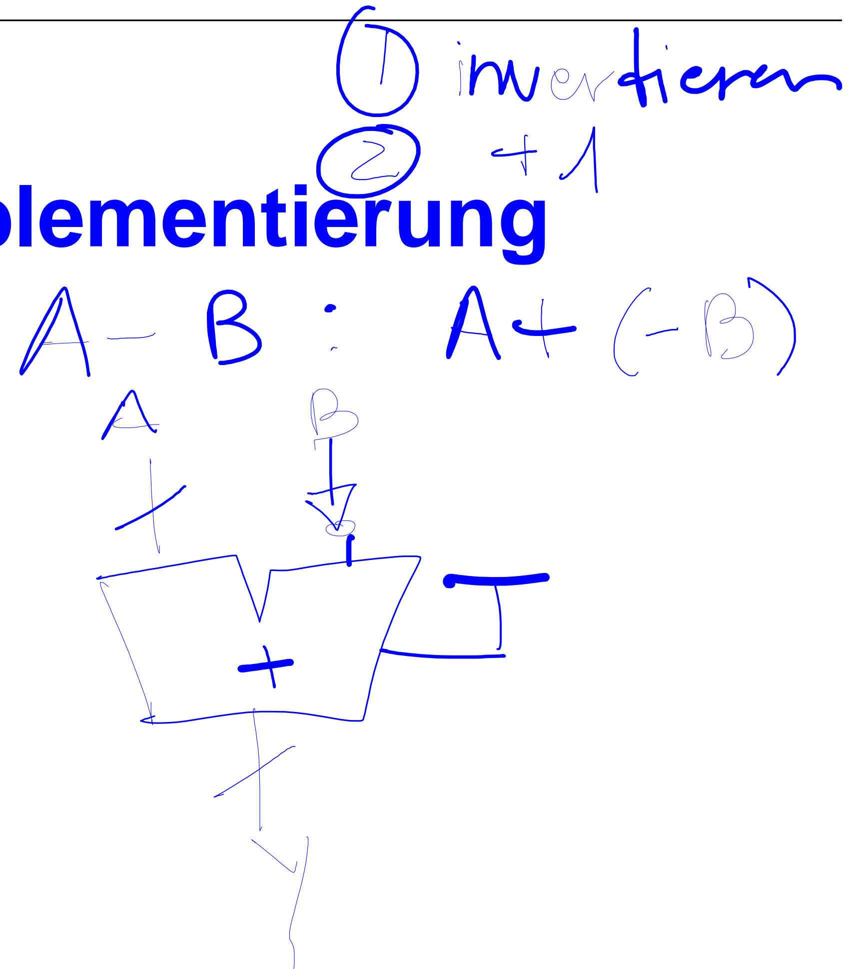
Subtrahierer

TECHNISCHE UNIVERSITÄT DARMSTADT

Symbol



Implementierung

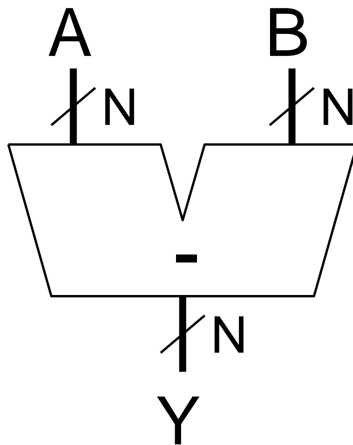


Embedded Systems & Applications

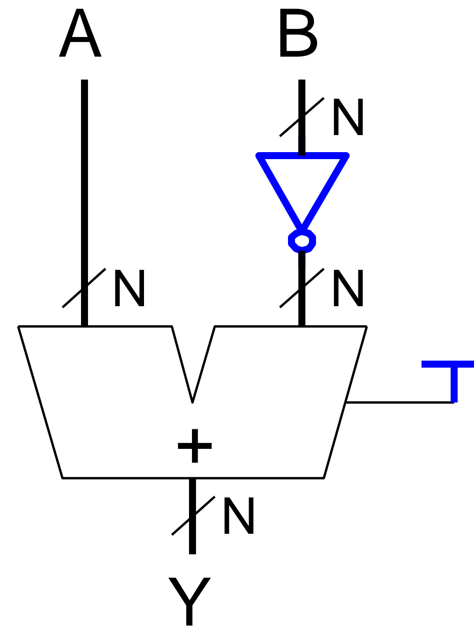
Subtrahierer

TECHNISCHE UNIVERSITÄT DARMSTADT

Symbol



Implementierung

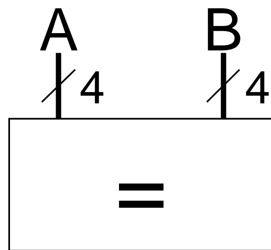


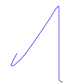
Embedded Systems & Applications

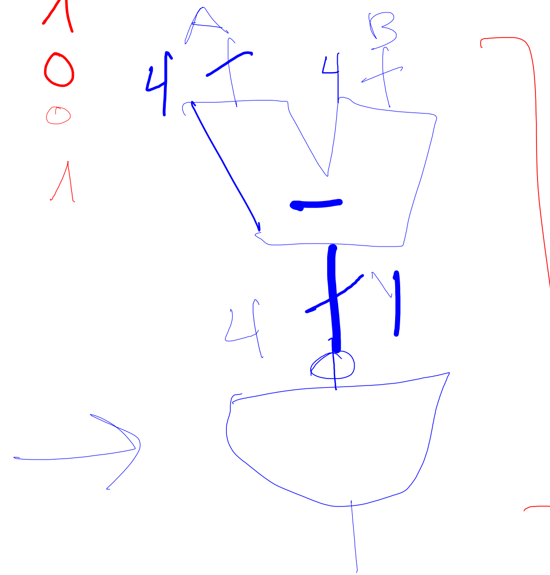
Vergleicher: Gleichheit

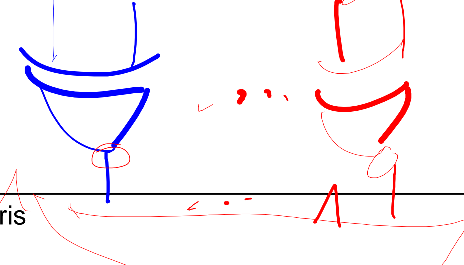
Symbol

Implementierung





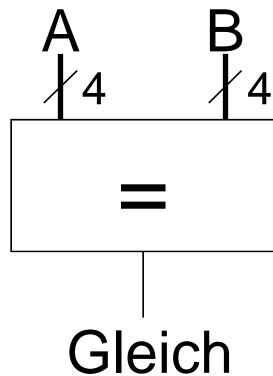




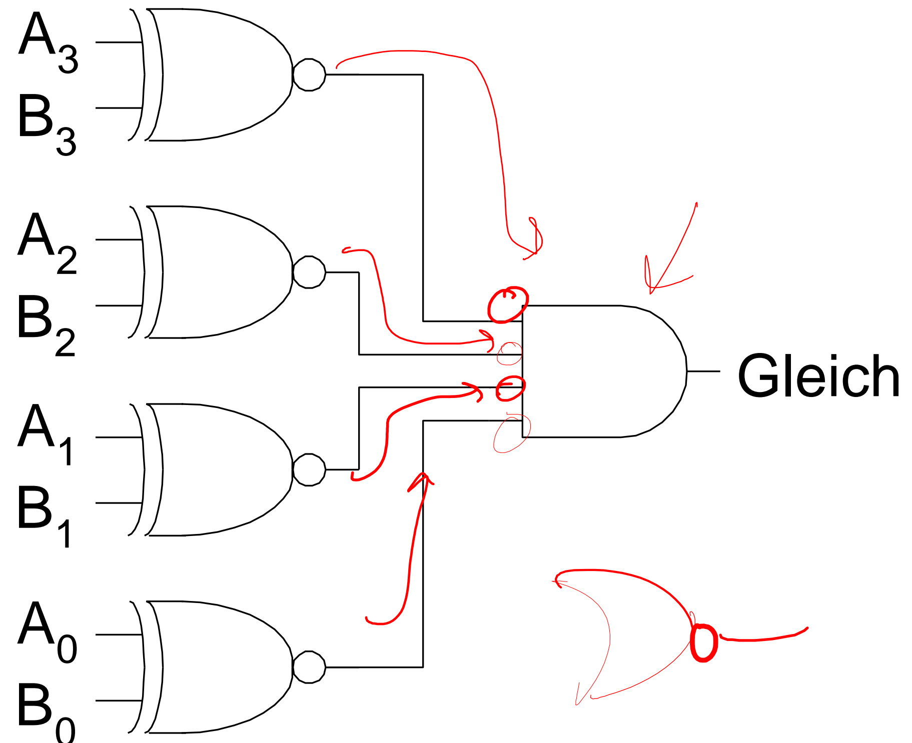
Vergleicher: Gleichheit

TECHNISCHE UNIVERSITÄT DARMSTADT

Symbol



Implementierung

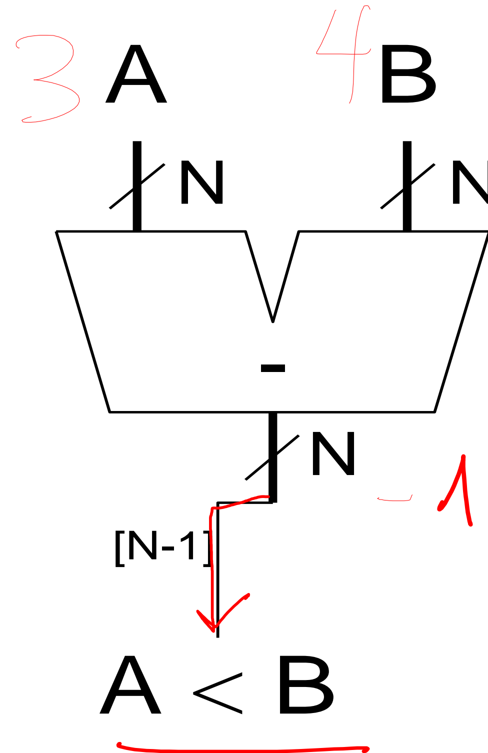


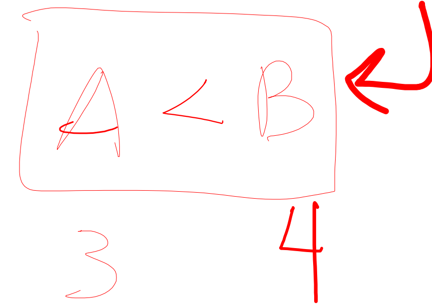
Embedded Systems & Applications

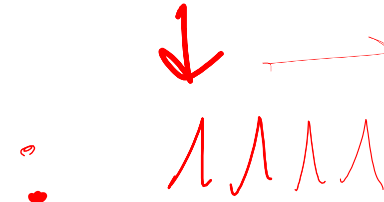
Vergleicher: Kleiner-Als

TECHNISCHE UNIVERSITÄT DARMSTADT

- Für N-Bit Zweierkomplementzahlen





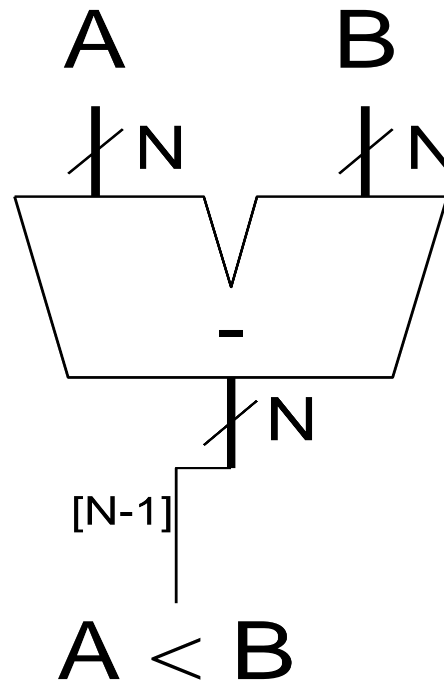


Embedded Systems & Applications

Vergleicher: Kleiner-Als

TECHNISCHE UNIVERSITÄT DARMSTADT

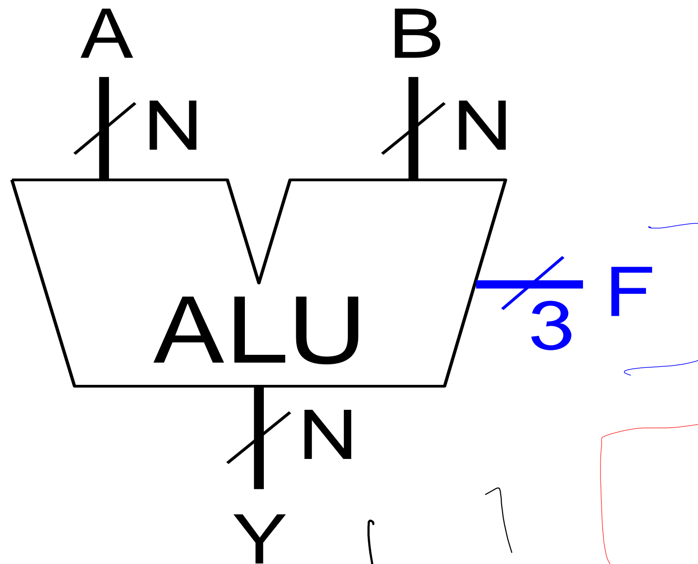
- Für N-Bit Zweierkomplementzahlen



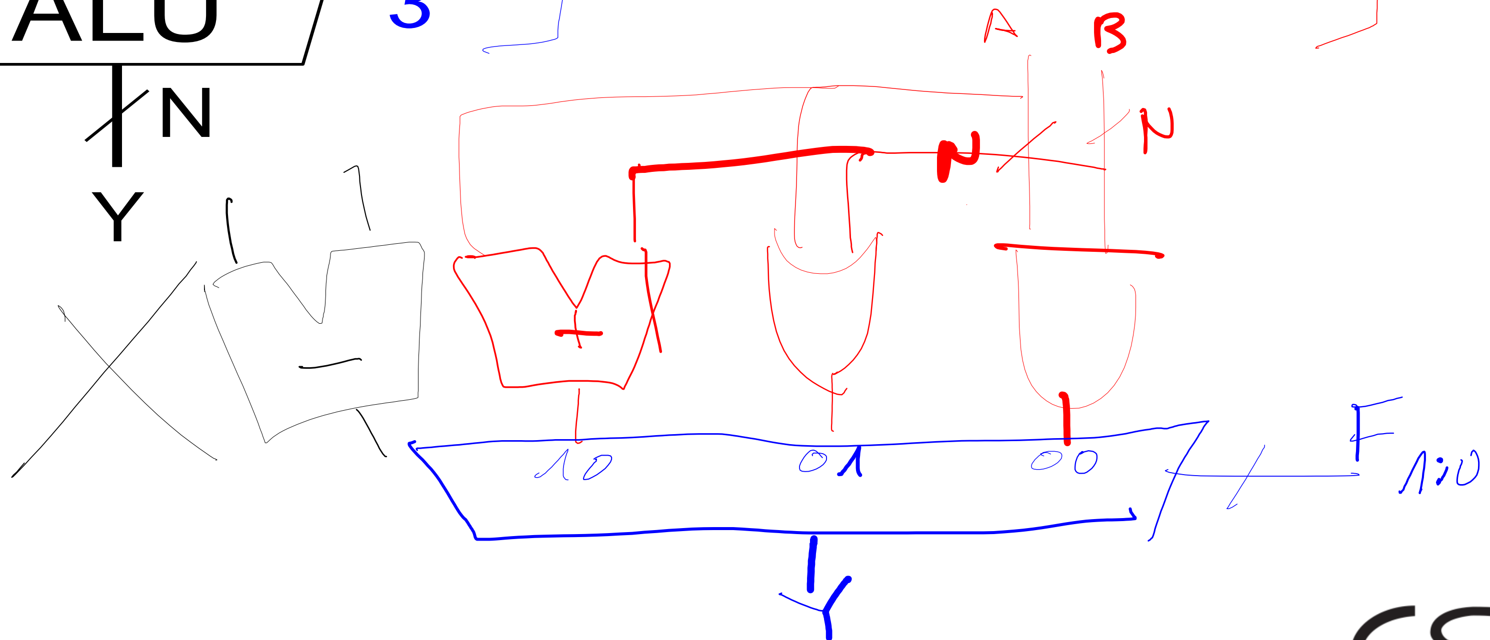
Aber Fehler beim Überlauf!

Embedded Systems & Applications

Arithmetisch-logische Einheit (*arithmetic logic unit, ALU*)

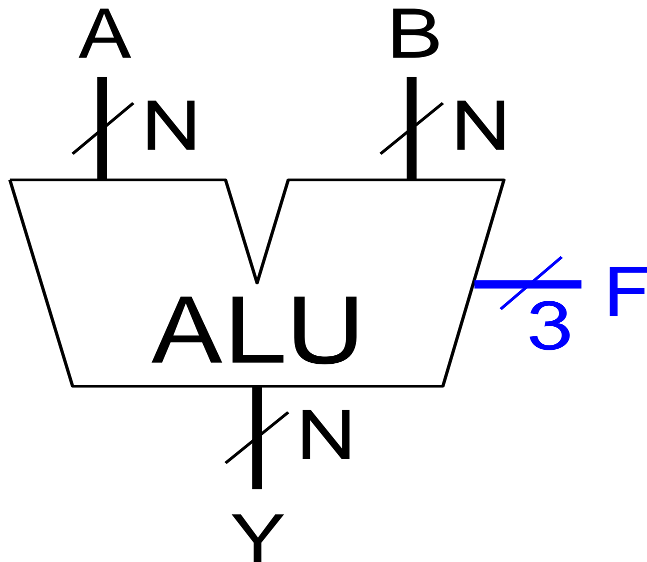


- **Funktionen:**
  - UND
  - ODER
  - Addition/Subtraktion



Arithmetisch-logische Einheit (*arithmetic logic unit, ALU*)

TECHNISCHE UNIVERSITÄT DARMSTADT

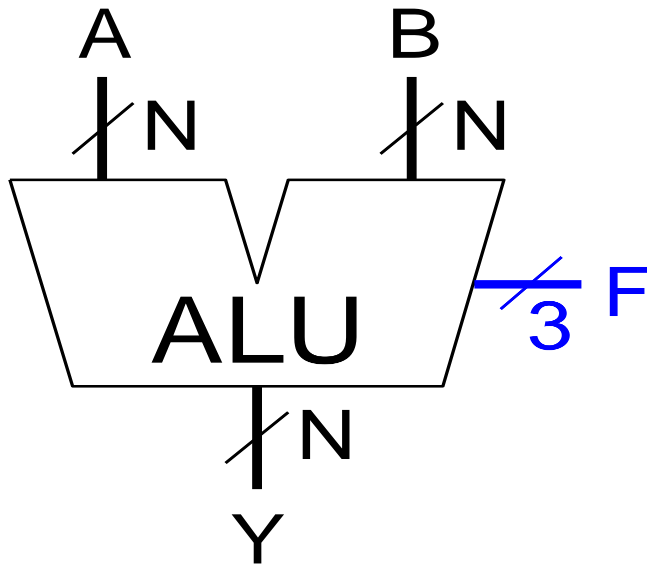


- **Funktionen:**
  - UND
  - ODER
  - Addition/Subtraktion

$F_{2:0}$	Funktion
000	A & B
001	A | B
010	A + B

Embedded Systems & Applications

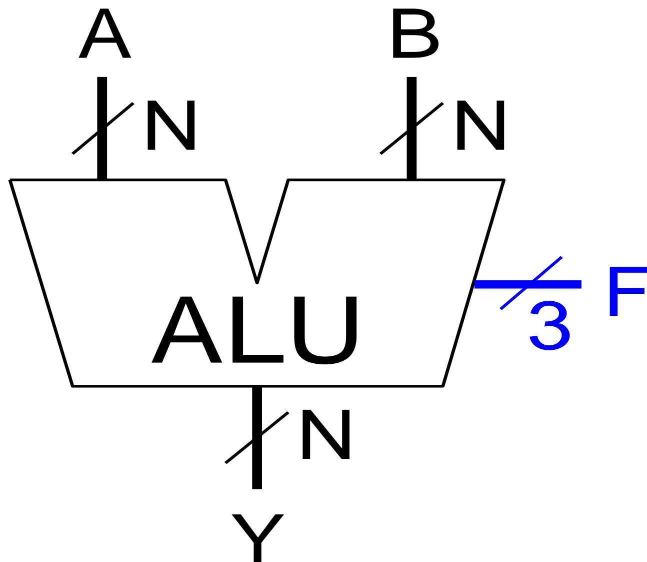
Arithmetisch-logische Einheit (*arithmetic logic unit, ALU*)



$F_{2:0}$	Funktion
000	A & B
001	A | B
010	A + B

Arithmetisch-logische Einheit (*arithmetic logic unit, ALU*)

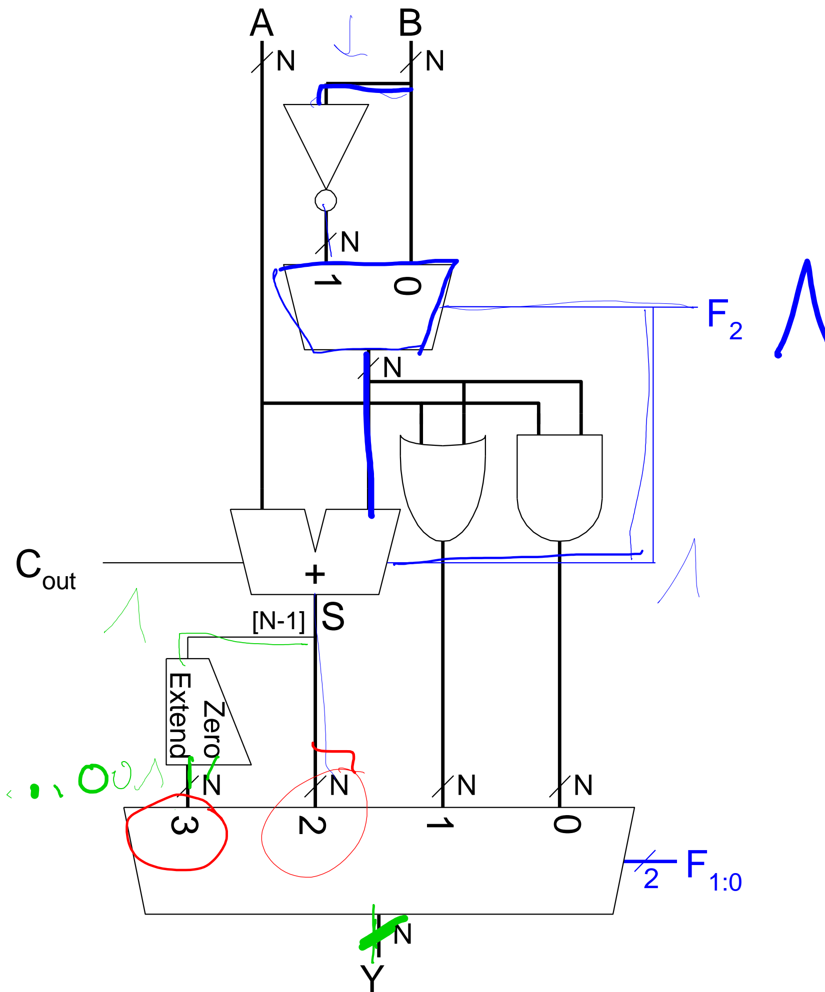
TECHNISCHE UNIVERSITÄT DARMSTADT



$F_{2:0}$	Funktion
000	A & B
001	A | B
010	A + B
011	Nicht verwendet
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Embedded Systems & Applications

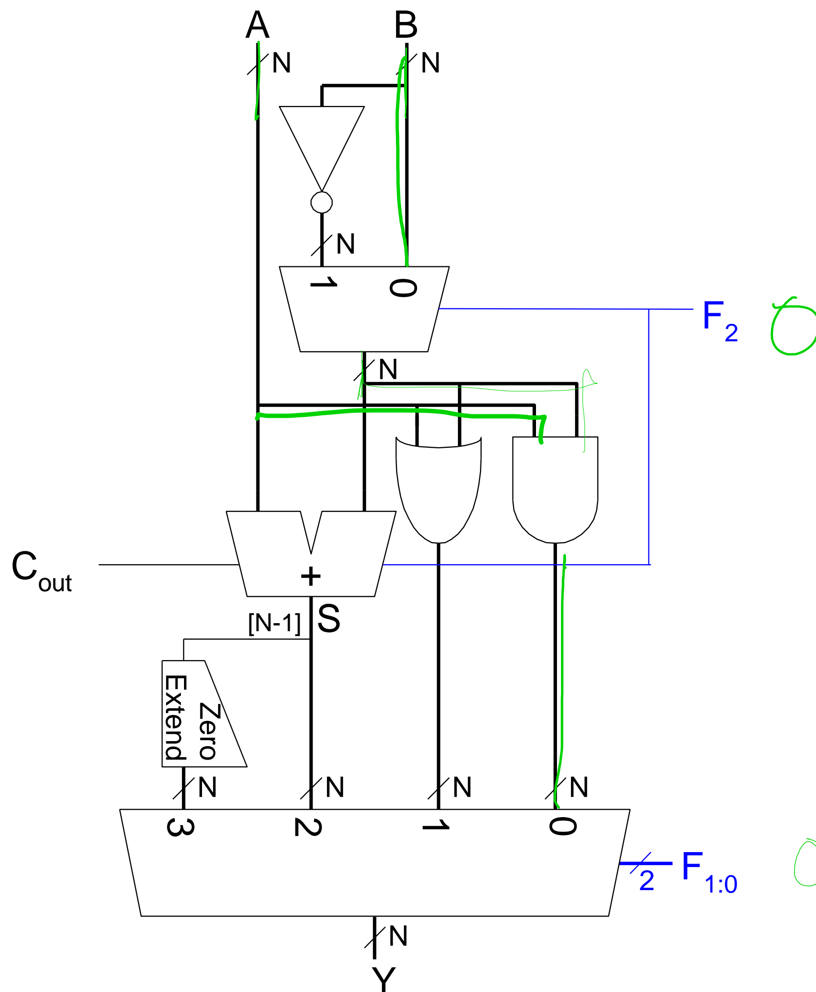
Entwurf einer ALU



$F_{2:0}$	Funktion
000	A & B
001	A | B
010	A + B
011	Nicht verwendet
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Beispiel: AND

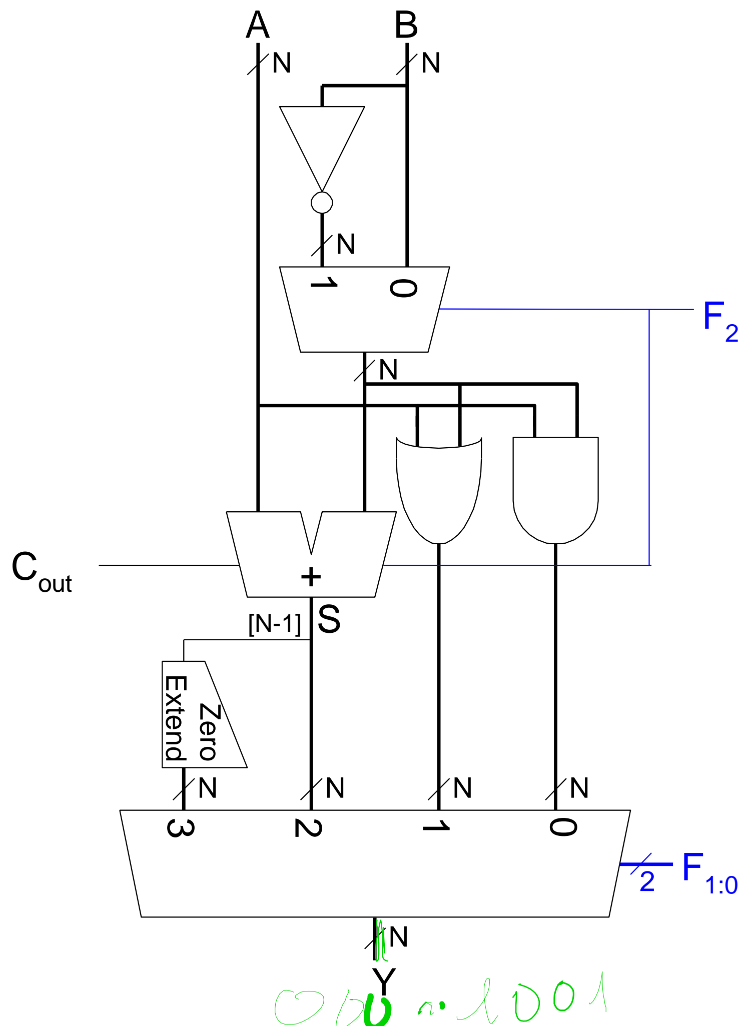
TECHNISCHE UNIVERSITÄT DARMSTADT



- **Konfiguriere 32b ALU für AND-Berechnung**
  - Annahme: A = 32‘b11001, B = 32‘b1101

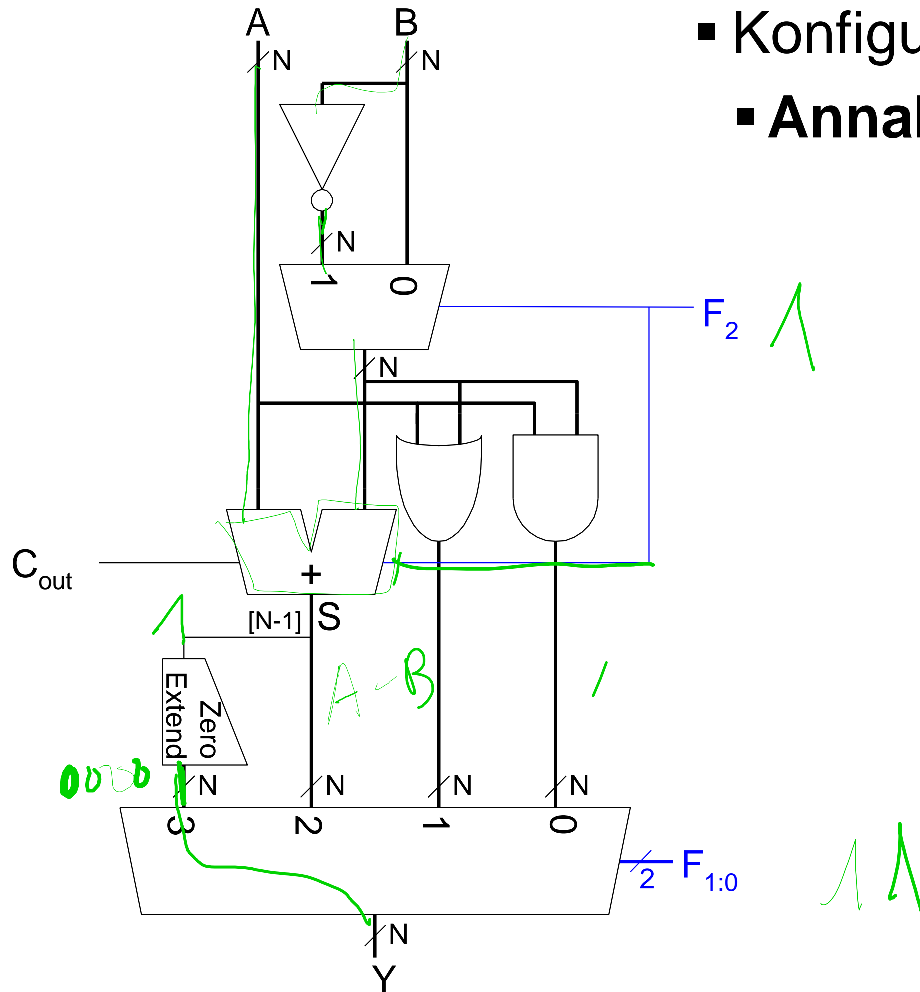
Embedded Systems & Applications

Beispiel: AND



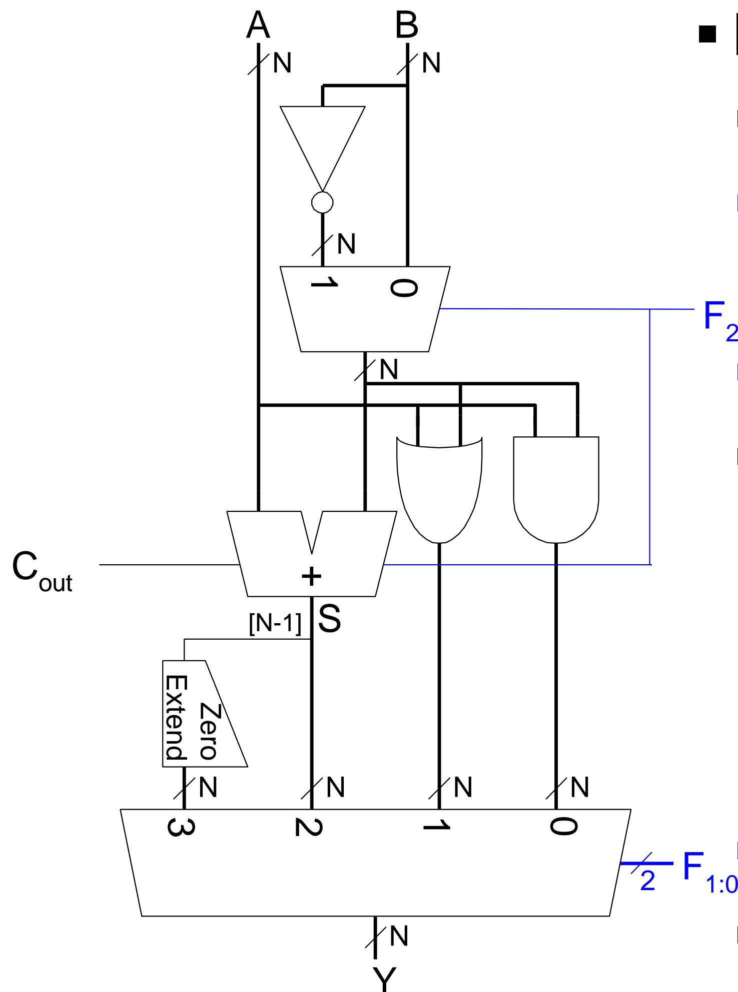
- **Konfiguriere 32b ALU für AND-Berechnung**
  - Annahme: A = 32‘b11001, B = 32‘b1101
  - **Erwartete Ausgabe**
    - Y = A & B = 32’b1001
  - **Steuereingang** für AND:
    - $F_{2:0}$ = 3’b000
  - ***$F_{1:0}$* = 2’b00** wählt *Y* = *UND Gatter Ausgänge*

Beispiel: Set Less Than (SLT)



- Konfiguriere 32b ALU für SLT-Berechnung
  - **Annahme: *A* = 25, *B* = 32**

Beispiel: Set Less Than (SLT)



- Konfiguriere 32b ALU für SLT-Berechnung
  - **Annahme: A = 25, B = 32**
  - **Erwartete Ausgabe:**
    A < B, also Y = 32'b1
  - **Steuereingang für SLT:** $F_{2:0}$ = 3'b111
  - F_2 = 1'b1 konfiguriert Addierer als **Subtrahierer**
    - S = 25 – 32 = -7
    - Im Zweierkomplement
      -7 = 32'h0xfffffff9 ➔msb S_{31} = 1
  - **$F_{1:0}$ = 2'b11** wählt $Y = S_{31}$ als Ausgabe
  - $Y = S_{31}$ (zero extended) = 32'h00000001

Schiebeoperationen (*shifter*)

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Logisches Schieben:** leere Stellen mit 0 aufgefüllt
  - Beispiel: 11001 >> 2 =
  - Beispiel: 11001 << 2 =
- **Arithmetisches Schieben:** wie logisches Schieben. Verwende aber beim Rechtsschieben alten Wert des msb zum Auffüllen leerer Stellen
  - Beispiel: 11001 >>> 2 =
  - Beispiel: 11001 <<< 2 =
- **Rotierer:** rotiert Bits im Kreis, herausgeschobene Bits tauchen am anderen Ende wieder auf
  - Beispiel : 11001 ROR 2 =
  - Beispiel : 11001 ROL 2 =

Embedded Systems & Applications

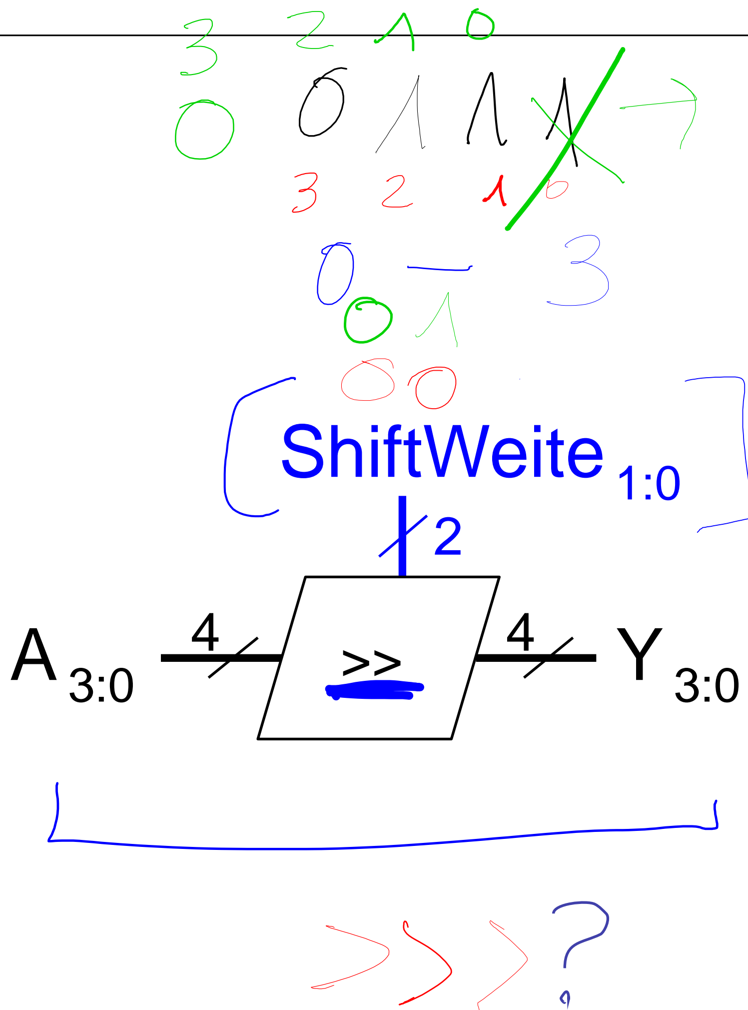
Schiebeoperationen (*shifter*)

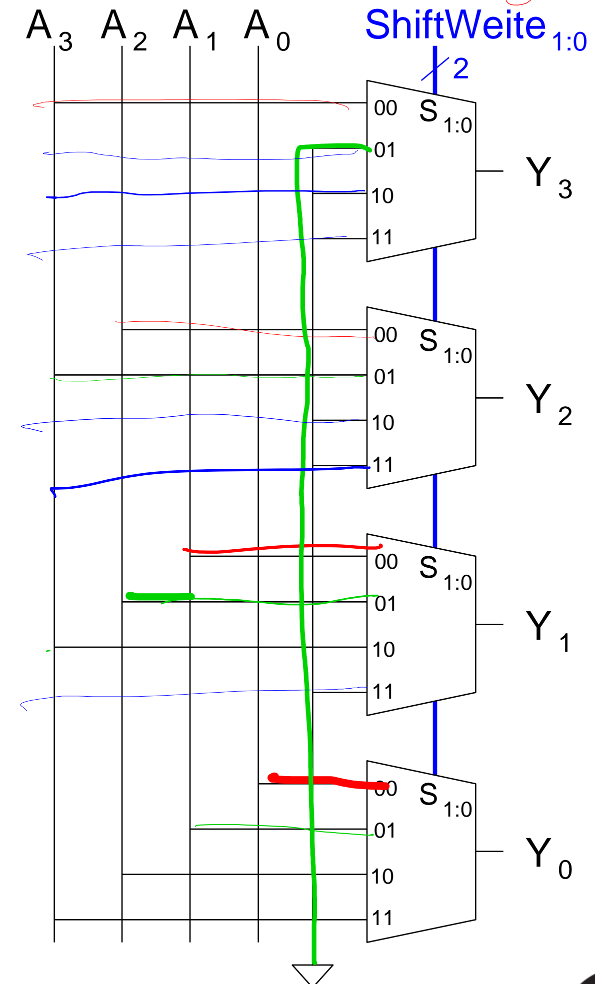
TECHNISCHE UNIVERSITÄT DARMSTADT

- **Logisches Schieben:** leere Stellen mit 0 aufgefüllt
  - Beispiel: 11001 >> 2 = **00**110
  - Beispiel: 11001 << 2 = 001**00**
- **Arithmetisches Schieben:** wie logisches Schieben. Verwende aber beim Rechtsschieben alten Wert des msb zum Auffüllen leerer Stellen
  - Beispiel: 11001 >>> 2 = **11**110
  - Beispiel: 11001 <<< 2 = 001**00**
- **Rotierer:** rotiert Bits im Kreis, herausgeschobene Bits tauchen am anderen Ende wieder auf
  - Beispiel : 11001 ROR 2 = 01110
  - Beispiel : 11001 ROL 2 = 00111

Embedded Systems & Applications

Aufbau von Shiftern





Shifter als Multiplizierer und Dividierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- Logisches Schieben um N Stellen nach links **multipliziert** den Zahlenwert mit $\mathbf{2^N}$
  - Beispiel : 00001 << **3** = 01**000** ($1 \times 2^3 = 8$)
  - Beispiel : 11101 << **2** = 101**00** ($-3 \times 2^2 = -12$)

- Arithmetisches Schieben um N Stellen nach rechts **dividiert** den Zahlenwert durch $\mathbf{2^N}$
  - Beispiel : 010000 >>> **4** = **0000**01 ($16 \div 2^4 = 1$)
  - Beispiel : 100000 >>> **2** = **11**1000 ($-32 \div 2^2 = -8$)

Embedded Systems & Applications

4 x 4 Multiplizierer

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Teilprodukte** gebildet vom Multiplizieren einer einzelnen Ziffer des Multiplikators mit dem Multiplikand
- **Verschobene** Teilprodukte danach **addiert**

Decimal		Binary
230	Multiplikand	0101
x 42	Multiplikator	x 0111
460	Teilprodukte	0101
+ 920		0101
		0101
		+ 0000
9660	Ergebnis	0100011

230 x 42 = 9660

5 x 7 = 35

Embedded Systems & Applications

4 x 4 Multiplizierer

$\rightarrow a * b$
$\rightarrow a | b$
$\rightarrow a + b$

$$
\begin{array}{cccccccc}
 & & & & A_3 & A_2 & A_1 & A_0 \\
 & & & \times & B_3 & B_2 & B_1 & B_0 \\
\hline
 & & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 \\
 & & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & \\
 & & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

Multiplikation von *k*-bit Zahlen hat 2*k*-bit breites Produkt

TECHNISCHE UNIVERSITÄT DARMSTADT
Embedded Systems & Applications

Division

- Leidlich einfach, dann aber sehr langsam
- Sehr kompliziert, dann wenigstens etwas schneller
  - Aber immer noch deutlich langsamer als z.B. Multiplikation
- Für Einführungsveranstaltung eher ungeeignet
  - Beschreibung im Buch auch ziemlich schlecht (?)…
- Hier nur aus dem Orbit gestreift
  - Auszug aus

Behrooz Parhami
Computer Arithmetic: Algorithms and Hardware Designs
Oxford U. Press, 2nd ed., 2010, ISBN 978-0-19-532848-6

Division Definitionen

TECHNISCHE UNIVERSITÄT DARMSTADT

Dividiere **vorzeichenlose** Zahlen:

Es gilt: $A/B = Q + R/B$

$A = BQ + R$

	Dividend	A	(k Bit breit)
	Divisor	B	(k Bit breit)
Ergebnis:			
	Quotient	Q	(k Bit breit)
	Rest	R	(k Bit breit)

Alle k-bit breit

Embedded Systems & Applications

Idee: Quotient ziffernweise bestimmen

TECHNISCHE UNIVERSITÄT DARMSTADT

$$A/B \quad = Q + R/B$$

Dezimal Beispiel: 2584/15 = 172 R4

Embedded Systems & Applications

Idee: Quotient ziffernweise bestimmen

$$A/B \quad = Q + R/B$$

Dezimal Beispiel: **2584/15 = 172 R4**

Langschrift:

```
     172 R4
15 | 2584
    -15
     108
    -105
       34
      -30
        4
```

7 x 15

2 x 15

Von **höher- zur niederwertigen** Stellen

1000er, 100er, 10er, 1er

Embedded Systems & Applications

Idee: Quotient ziffernweise bestimmen

A/B = Q + R/B

Dezimal Beispiel: 2584/15 = 172 R4

Langschrift:

```
     172 R4
15 | 2584
    -15
     108
    -105
       34
      -30
        4
```

den Divisor nach rechts schieben

Von **höher- zur niederwertigen** Stellen

1000er, 100er, 10er, 1er

Idee: Quotient ziffernweise bestimmen

TECHNISCHE UNIVERSITÄT DARMSTADT

$$A/B = Q + R/B$$

Dezimal Beispiel: 2584/15 = 172 R4

Langschrift:

```
      172  R4
15 | 2584
     -15
      108
     -105
        34
       -30
         4
```

den Divisor nach rechts schieben

Andersherum:

```
  0002
-   15
   -13
```

den Dividend nach links schieben

Von **höher- zur niederwertigen** Stellen

1000er, 100er, 10er, 1er

Embedded Systems & Applications

Idee: Quotient ziffernweise bestimmen

TECHNISCHE UNIVERSITÄT DARMSTADT

A/B = Q + R/B

den Dividend nach links schieben

Dezimal Beispiel: 2584/15 = 172 R4

Langschrift:

```
     172  R4
15 | 2584
    -15
     108
    -105
       34
      -30
        4
```

Andersherum:

```
 0002
-  15
  -13
```

1. Die **höchstwertige Ziffer** ganz rechts schreiben

Von **höher- zur niederwertigen** Stellen

1000er, 100er, 10er, 1er

Embedded Systems & Applications

Idee: Quotient ziffernweise bestimmen

A/B = Q + R/B

Dezimal Beispiel: 2584/15 = 172 R4

den Dividend nach links schieben

Langschrift:

```
      172 R4
15 | 2584
     -15
      108
     -105
        34
       -30
         4
```

Andersherum:

```
  0002
-   15
   -13
```

1. Die **höchstwertige Ziffer** ganz rechts schreiben
2. B **subtrahieren**

Von **höher- zur niederwertigen** Stellen

1000er, 100er, 10er, 1er

Embedded Systems & Applications

Idee: Quotient ziffernweise bestimmen

A/B = Q + R/B

den Dividend nach links schieben

Dezimal Beispiel: 2584/15 = 172 R4

Langschrift:

```
       172  R4
15 | 2584
     -15
      108
     -105
        34
       -30
         4
```

Andersherum:

```
  0002
 -  15
   -13
```

0 _ _ _ (Stellen 3 2 1 0)

Ergebnis **negativ** (B passt nicht)

Von **höher- zur niederwertigen** Stellen

1000er, 100er, 10er, 1er

Idee: Quotient ziffernweise bestimmen

A/B = Q + R/B

den Dividend nach links schieben

Dezimal Beispiel: 2584/15 = 172 R4

Langschrift:

```
     172 R4
15 | 2584
    -15
     108
    -105
       34
      -30
        4
```

Andersherum:

```
 0002
-  15
  -13

 002
-  15
   10
```

0 _ _ _ (Stellen 3 2 1 0)

eine Ziffer **nach links** schieben

Von **höher- zur niederwertigen** Stellen

1000er, 100er, 10er, 1er

Idee: Quotient ziffernweise bestimmen

TECHNISCHE UNIVERSITÄT DARMSTADT

A/B = Q + R/B

Dezimal Beispiel: 2584/15 = 172 R4

den Dividend nach links schieben

Langschrift:

```
    172  R4
15 |2584
   -15
    108
   -105
      34
     -30
       4
```

Andersherum:

```
 ← 0002
  -  15
    -13        0 _ _ _
               3 2 1 0
   0025
  -  15
     10
```

eine Ziffer **nach links** schieben

und **die nächst höchstwertige Ziffer** dazu schreiben

Von **höher- zur niederwertigen** Stellen

1000er, 100er, 10er, 1er

Embedded Systems & Applications

Idee: Quotient ziffernweise bestimmen

TECHNISCHE UNIVERSITÄT DARMSTADT

A/B = Q + R/B

Dezimal Beispiel: 2584/15 = 172 R4

den Dividend nach links schieben

Langschrift:

```
       172  R4
15 | 2584
    -15
     108
    -105
       34
      -30
        4
```

Andersherum:

```
 0002
-  15
  -13        0 _ _ _
             3 2 1 0

 0025
-  15
   10        0 1 _ _
             3 2 1 0
```

Ergebnis **positiv**
(jetzt B passt)

Von **höher- zur niederwertigen** Stellen

1000er, 100er, 10er, 1er

Embedded Systems & Applications

Idee: Quotient ziffernweise bestimmen

TECHNISCHE UNIVERSITÄT DARMSTADT

A/B = Q + R/B

den Dividend nach links schieben

Dezimal Beispiel: 2584/15 = 172 R4

Langschrift:

```
   172 R4
15|2584
  -15
   108
  -105
     34
    -30
      4
```

Andersherum:

```
 0002
-  15
  -13        0 _ _ _
             3 2 1 0

 0025
-  15
   10        0 1 _ _
             3 2 1 0
 0108
- 105
    3
```

Differenz nach links schieben

und **die nächst höchstwertige Ziffer** dazu schreiben

Von **höher- zur niederwertigen** Stellen

1000er, 100er, 10er, 1er

Embedded Systems & Applications

Idee: Quotient ziffernweise bestimmen

$$A/B = Q + R/B$$

Dezimal Beispiel: **2584/15 = 172 R4**

den Dividend nach links schieben

Langschrift:

```
     172  R4
15 | 2584
    -15
     108
    -105
       34
      -30
        4
```

Andersherum:

```
  0002
-   15        0 _ _ _
 -----        3 2 1 0
   -13

  0025
-   15        0 1 _ _
 -----        3 2 1 0
    10

  0108
-  105        0 1 7 _       7 x 15 = 105
 -----        3 2 1 0
     3
```

Von **höher- zur niederwertigen** Stellen

1000er, 100er, 10er, 1er

Idee: Quotient ziffernweise bestimmen

TECHNISCHE UNIVERSITÄT DARMSTADT

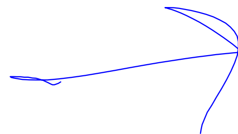
A/B = Q + R/B

den Dividend nach links schieben

Dezimal Beispiel: **2584/15 = 172 R4**

Langschrift:

```
     172 R4
15 | 2584
    -15
     108
    -105
      34
     -30
       4
```



Andersherum:

```
 0002
-  15
  -13        0 _ _ _
             3 2 1 0

 0025
-  15
   10        0 1 _ _
             3 2 1 0

 0108
- 105
    3        0 1 7 _
             3 2 1 0

 0034
-  30
    4        0 1 7 2
             3 2 1 0
```

Von **höher- zur niederwertigen** Stellen
1000er, 100er, 10er, 1er

Embedded Systems & Applications

Dividierer Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

Vorzeichenlose Zahlen

Dezimal: 2584/15=172 R4 **Binär:** 1101/0010 = 0110 R1

```
  0002          0 _ _ _
-   15          3 2 1 0
  -13

  0025          0 1 _ _
-   15          3 2 1 0
   10

  0108          0 1 7 _
-  105          3 2 1 0
     3

  0034          0 1 7 2
-   30          3 2 1 0
     4  Rest
```

```
 0001101        0 _ _ _
-0010           3 2 1 0
 1111

 001101         0 1 _ _
-0010           3 2 1 0
 0001

 00101          0 1 1 _
-0010           3 2 1 0
 0000

 0001           0 1 1 0
-0010           3 2 1 0
 1111
```


Embedded Systems & Applications

Dividierer Algorithmus

Vorzeichenlose Zahlen

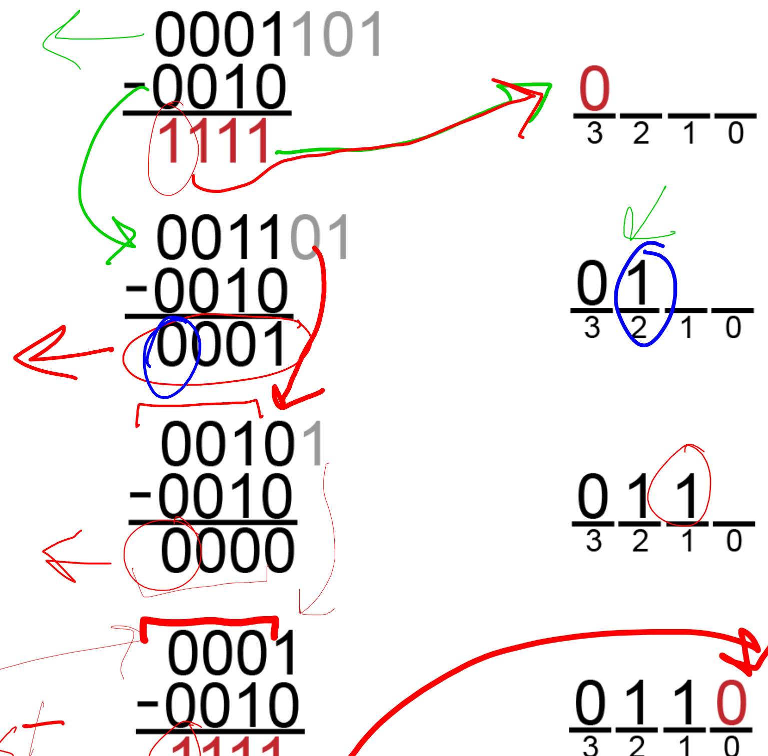
Dezimal: 2584/15=172 R4 **Binär:** 1101/0010 = 0110 R1

```
 0002
-  15
 ----
  -13        0 _ _ _
             3 2 1 0

 0025
-  15
 ----
   10        0 1 _ _
             3 2 1 0

 0108
- 105
 ----
    3        0 1 7 _
             3 2 1 0

 0034
-  30
 ----
    4        0 1 7 2
             3 2 1 0
```



Dividierer Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

A/B = Q + R/B

$R' = 0$
for i = N-1 to 0
  $R = \{R' << 1, A_i\}$
  $D = R - B$
  if $D < 0$, $Q_i = 0$; $R' = R$
  else $Q_i = 1$; $R' = D$
$R = R'$

Binär: 1101/0010 = 0110 R1

```
 0001101
-0010
 1111
```
0 _ _ _ (3 2 1 0)

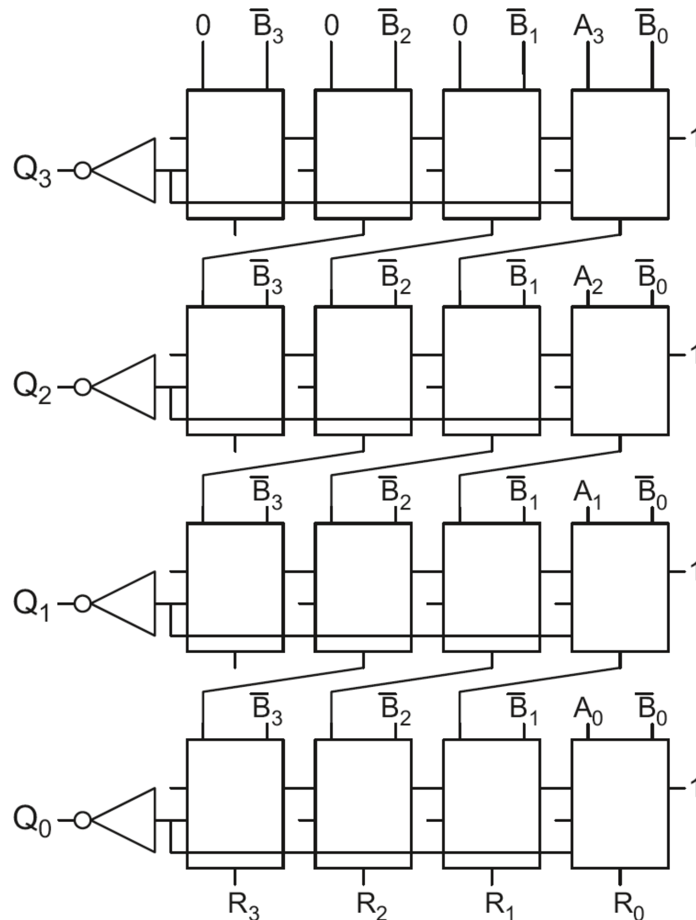
```
 001101
-0010
 0001
```
0 1 _ _ (3 2 1 0)

```
 00101
-0010
 0000
```
0 1 1 _ (3 2 1 0)

```
 0001
-0010
 1111
```
0 1 1 0 (3 2 1 0)

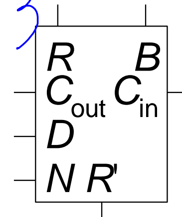
Embedded Systems & Applications

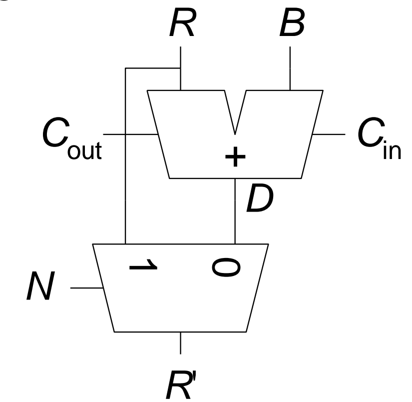
Kombinatorischer 4 x 4 Array-Dividierer: A / B



Jede Zeile rechnet eine Iteration des Algorithmus

Legend





Division: A/B = Q + R/B

R′ = 0

for i = N-1 to 0

R = {R′ << 1, A_i}

D = R - B

if D < 0, Q_i=0, R′ =R

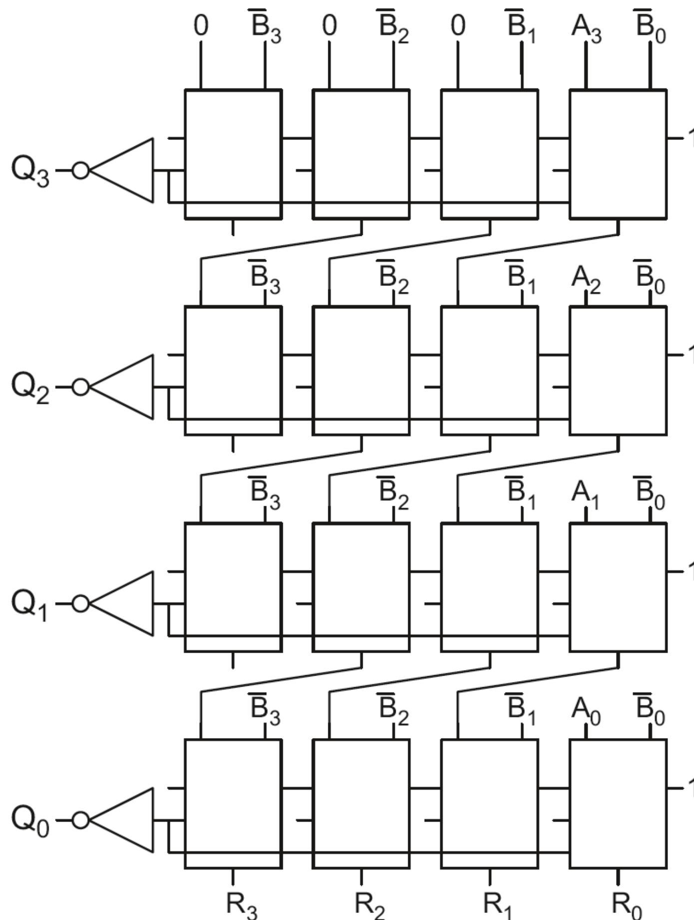
else Q_i=1, R′ =D

R=R′

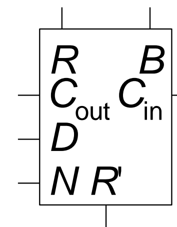
Embedded Systems & Applications

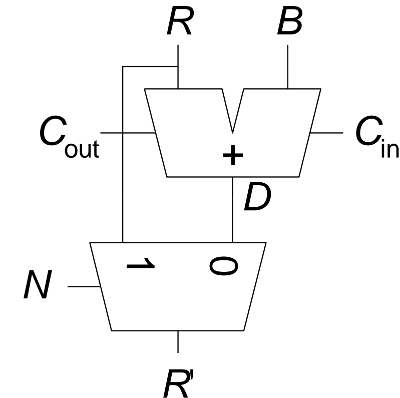
Kombinatorischer 4 x 4 Array-Dividierer: A / B

TECHNISCHE UNIVERSITÄT DARMSTADT



Legend





1101/0010 = 0110 R1

```
 0001101
-0010
 1111        0 _ _ _
             3 2 1 0

 001101
-0010
 0001        0 1 _ _
             3 2 1 0

 00101
-0010
 0000        0 1 1 _
             3 2 1 0

 0001
-0010
 1111        0 1 1 0
             3 2 1 0
```

Jede Zeile rechnet eine Iteration des Algorithmus

Embedded Systems & Applications

Organisatorische

TECHNISCHE UNIVERSITÄT DARMSTADT

- Evaluationen Feedback
- Wiederholung: Nächste Woche (10.02) werden wir das Material des Semesters wiederholen
  - Ab Freitag (05.02) werden Probefragen bei Moodle hochgeladen
  - Die selber versuchen vor der Vorlesung
- Klausur:
  - 01.03.2016 (Dienstag)
  - 11:00 Uhr – 12:30 Uhr (noch immer unter Vorbehalt)
  - die Räume werde wir durch Moodle bekannt machen

Division

TECHNISCHE UNIVERSITÄT DARMSTADT

A/B = Q + R/B

Dezimal Beispiel:

2584/15 = 172 R4

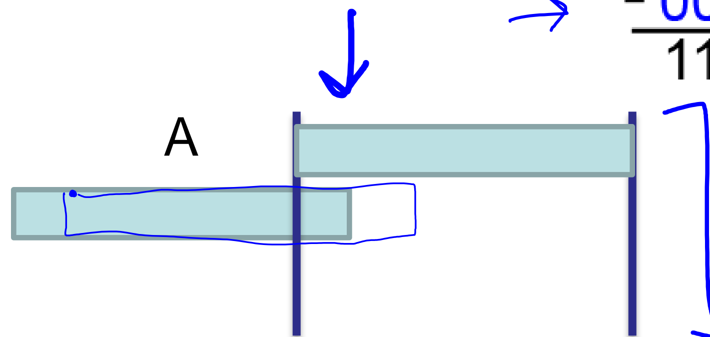
```
     172  R4
15 | 2584
    -15
     108
    -105
      34
     -30
       4
```

Binär Beispiel:

1101/0010 = 0110 R 1

```
         0110
0010 | 1101
      -0010
       00010
       -0010
        00001
        -0010
          1111
```

den Divisor nach rechts geschoben



Embedded Systems & Applications

Division

TECHNISCHE UNIVERSITÄT DARMSTADT

A/B = Q + R/B

Dezimal Beispiel:

2584/15 = 172 R4

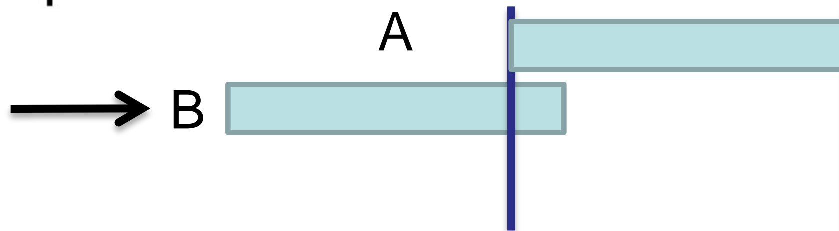
```
      172  R4
 15 | 2584
     -15
      108
     -105
        34
       -30
         4
```

Binär Beispiel:

1101/0010 = 0110 R 1

```
          0110
 0010 | 1101
      -0010
       1111
```

den Divisor nach rechts geschoben



Division

TECHNISCHE UNIVERSITÄT DARMSTADT

A/B = Q + R/B

Dezimal Beispiel:

2584/15 = 172 R4

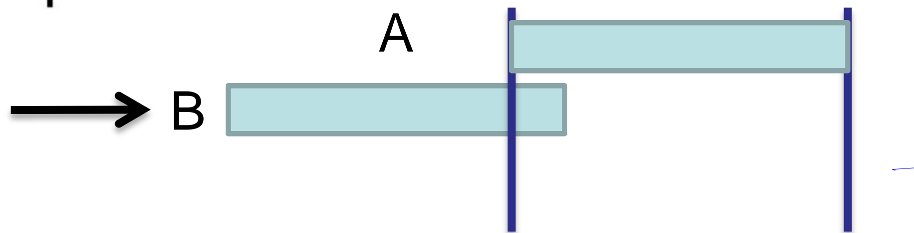
```
      172  R4
  15 |2584
     -15
      108
     -105
        34
       -30
         4
```

Binär Beispiel:

1101/0010 = 0110 R 1

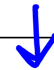
```
          0110
  0010 |1101
       -0010
        00010
        -0010
         00001
         -0010
          1111
```

den Divisor nach rechts geschoben



Embedded Systems & Applications

Division



A/B = Q + R/B

1101/0010 = 0110 R 1

```
  0001101
 -0010
  1111
```

0 _ _ _ (3 2 1 0)

```
  001101
 -0010
  0001
```

0 1 _ _ (3 2 1 0)

```
  00101
 -0010
  0000
```

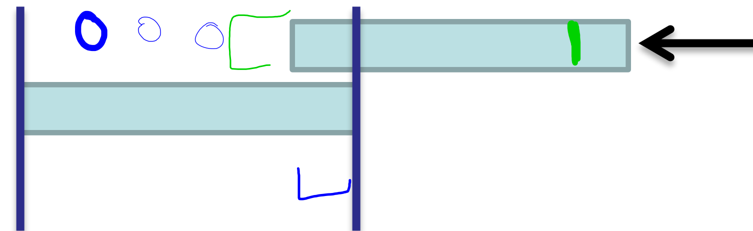
0 1 1 _ (3 2 1 0)

```
  0001
 -0010
  1111
```

0 1 1 0 (3 2 1 0)

```
        0110
 0010 | 1101
       -0010
        00010
        -0010
         00001
         -0010
          1111
```

Besser: den Dividend / partiellen Rest nach links schieben



Embedded Systems & Applications

Dividierer Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

Binär: 1101/0010 = 0110 R1

A/B = Q + R/B

$R' = 0$

for i = N-1 to 0

$R = \{R' << 1, A_i\}$

$D = R - B$

if $D < 0$, $Q_i = 0$; $R' = R$

else $Q_i = 1$; $R' = D$

$R = R'$

i = 3

```
 0001101
-0010
 1111
```

$\underset{3}{0}\ \underset{2}{_}\ \underset{1}{_}\ \underset{0}{_}$

i = 2

```
 001101
-0010
 0001
```

$\underset{3}{0}\ \underset{2}{1}\ \underset{1}{_}\ \underset{0}{_}$

i = 1

```
 00101
-0010
 0000
```

$\underset{3}{0}\ \underset{2}{1}\ \underset{1}{1}\ \underset{0}{_}$

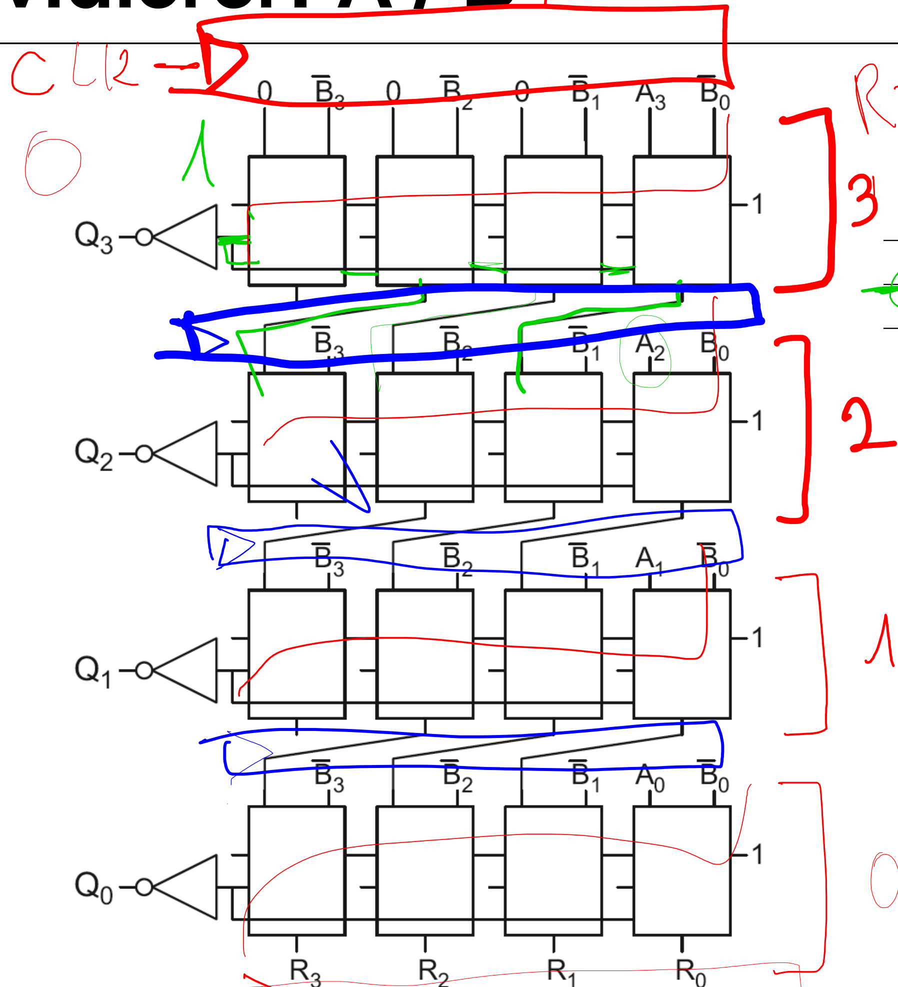
i = 0

```
 0001
-0010
 1111
```

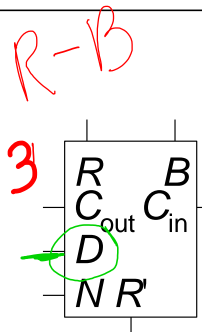
$\underset{3}{0}\ \underset{2}{1}\ \underset{1}{1}\ \underset{0}{0}$

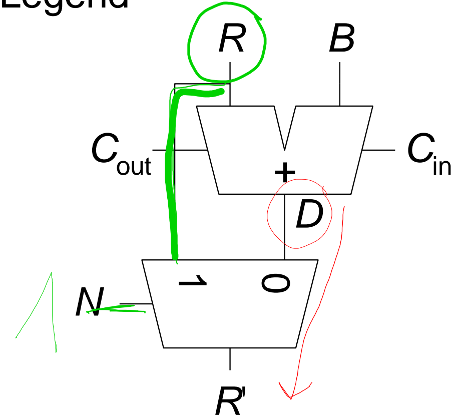
Embedded Systems & Applications

Kombinatorischer 4 x 4 Array-Dividierer: A / B



Legend





Division: $A/B = Q + R/B$

$R' = 0$

for i = N-1 to 0

  $R = \{R' << 1, A_i\}$

  $D = R - B$

  if $D < 0$, $Q_i = 0$, $R' = R$

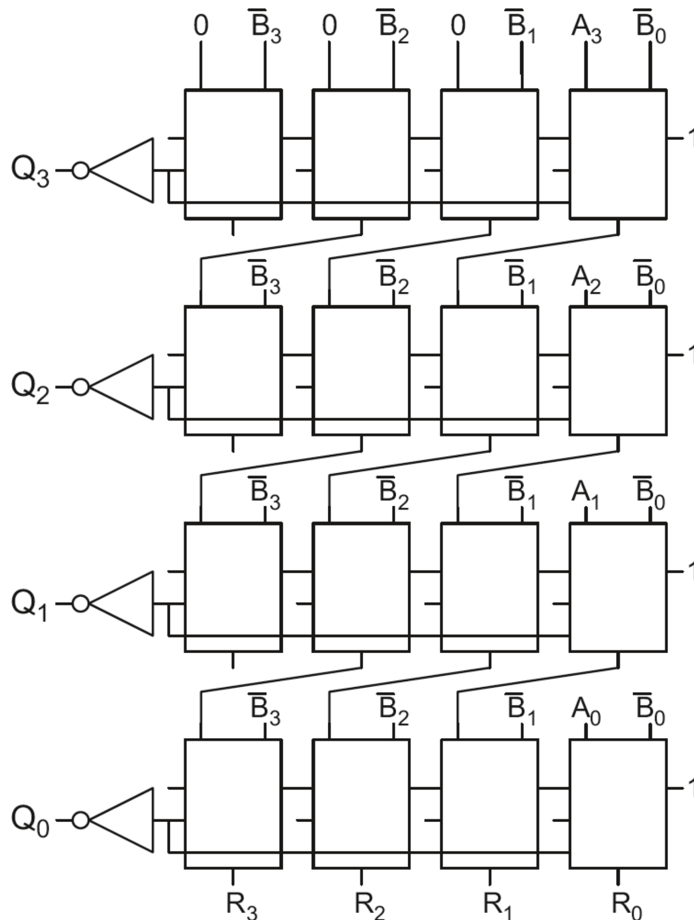
  else $Q_i = 1$, $R' = D$

$R = R'$

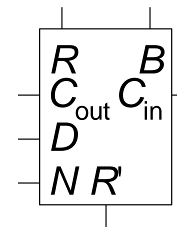
Jede Zeile rechnet eine Iteration des Algorithmus

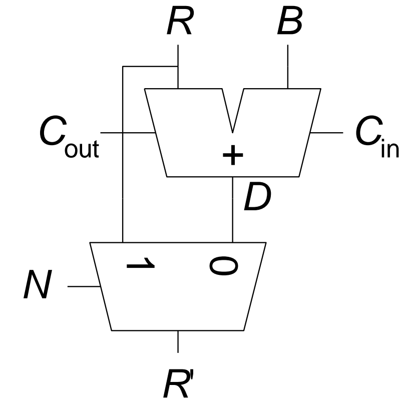
TECHNISCHE UNIVERSITÄT DARMSTADT
Embedded Systems & Applications

Kombinatorischer 4 x 4 Array-Dividierer: A / B



Legend





1101/0010 = 0110 R1

```
 0001101
-0010
 1111
```

0 _ _ _ (3 2 1 0)

```
 001101
-0010
 0001
```

0 1 _ _ (3 2 1 0)

```
 00101
-0010
 0000
```

0 1 1 _ (3 2 1 0)

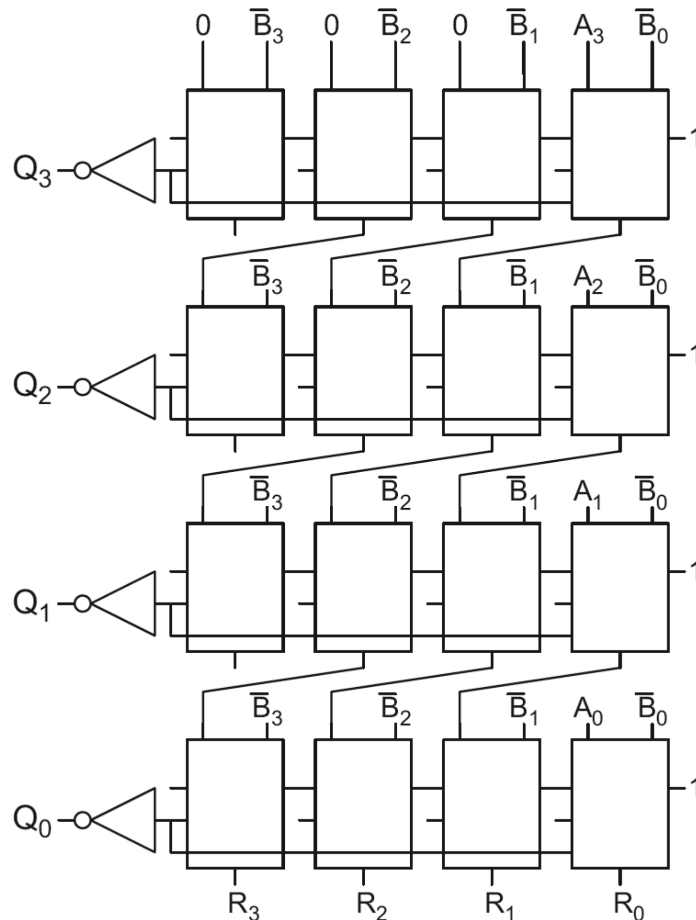
```
 0001
-0010
 1111
```

0 1 1 0 (3 2 1 0)

Jede Zeile rechnet eine Iteration des Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT
Embedded Systems & Applications

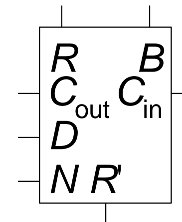
Kombinatorischer 4 x 4 Array-Dividierer: A / B

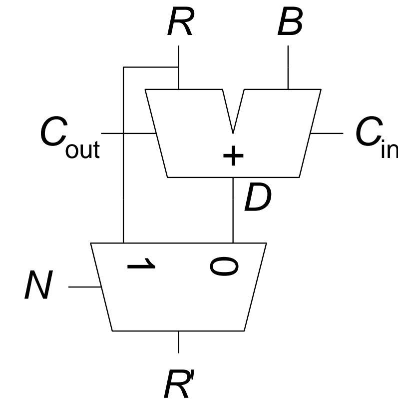


Jede Zeile rechnet eine Iteration des Algorithmus

Verzögerung proportional zu N^2

Legend





Division: $A/B = Q + R/B$

$R' = 0$

for i = N-1 to 0

  $R = \{R' << 1, A_i\}$

  $D = R - B$

  if $D < 0$, $Q_i=0$, $R'=R$

  else $Q_i=1$, $R'=D$

$R=R'$

Division: Unterschiedliche Bitbreite

TECHNISCHE UNIVERSITÄT DARMSTADT

$$A/B = Q + R/B$$

Aber...

- wenn A 2x so breit ist wie B, Q, und R
- denn müssen wir auf **Überlauf** (overflow) aufpassen

Embedded Systems & Applications

Division mit unterschiedlichen Bitbreiten

A	Dividend	$A_{2k-1}A_{2k-2}\ .\ .\ .\ A_3A_2A_1A_0$
B	Divisor	$B_{k-1}B_{k-2}\ .\ .\ .B_1B_0$
Q	Quotient	$Q_{k-1}Q_{k-2}.\ .\ .Q_1Q_0$
R	Rest, $A-(B*Q)$	$R_{k-1}R_{k-2}.\ .\ .R_1R_0$

Dividiere **vorzeichenlose** Zahlen:

Es gilt: $\boldsymbol{A/B = Q + R/B}$ ***nach wie vor***

$\boldsymbol{A = BQ + R}$

Dividend	***A***	(**2k** Bit breit)
Divisor	***B***	(k Bit breit)

Ergebnis:

Quotient	***Q***	(k Bit breit)
Rest	***R***	(k Bit breit)

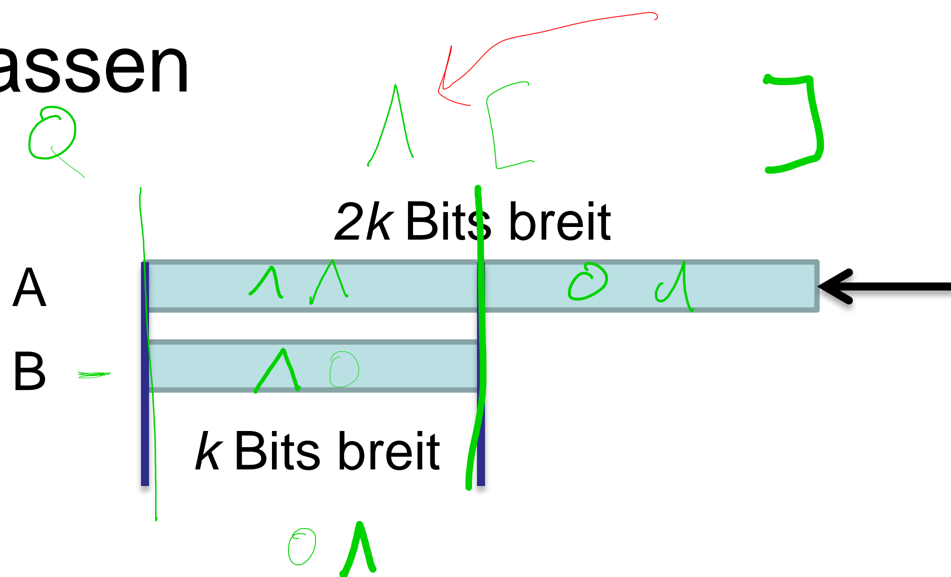
TECHNISCHE UNIVERSITÄT DARMSTADT
Embedded Systems & Applications

Division: Unterschiedliche Bitbreite

$$A/B = Q + R/B$$

Aber...

- wenn A 2x so breit ist wie B, Q, und R
- denn müssen wir auf **Überlauf** (overflow) aufpassen



fangen wir hier an, aber den Dividend immer noch nach links schieben

Auftreten von Überläufen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beispiel k=8:16b Dividend (A), 8b Divisor (B), 8b Quotient (Q), 8b Rest (R)

- **Problem:** Damit nicht alle Ergebniswerte repräsentierbar
  - Operanden: A = 1482, B = 3
  - Ergebnis: Q = 494 **nicht mehr in 8b darstellbar (nicht<2^8 = 256), Überlauf!**
    R = 0

- **Vorgehensweise:** Vorher auf darstellbares Ergebnis prüfen
  - Vermeidet Überlauf
  - Fängt auch Division durch Null ab

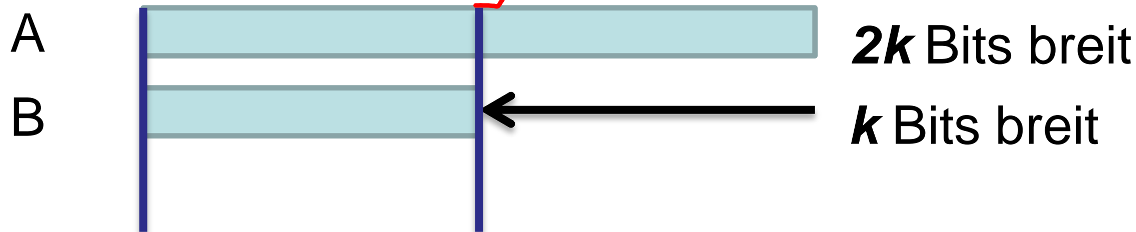
Embedded Systems & Applications

Abfangen von Überläufen

- **Beispiel:**

  k: 8 bit
  Operanden: A (16b) = 1482, B(8b) = 3
  Ergebnis: Q (8b) = 494
  R (8b) = 0

```
A:   1482:  0000 0101 1100 1010
B:      3:  0000 0011
```



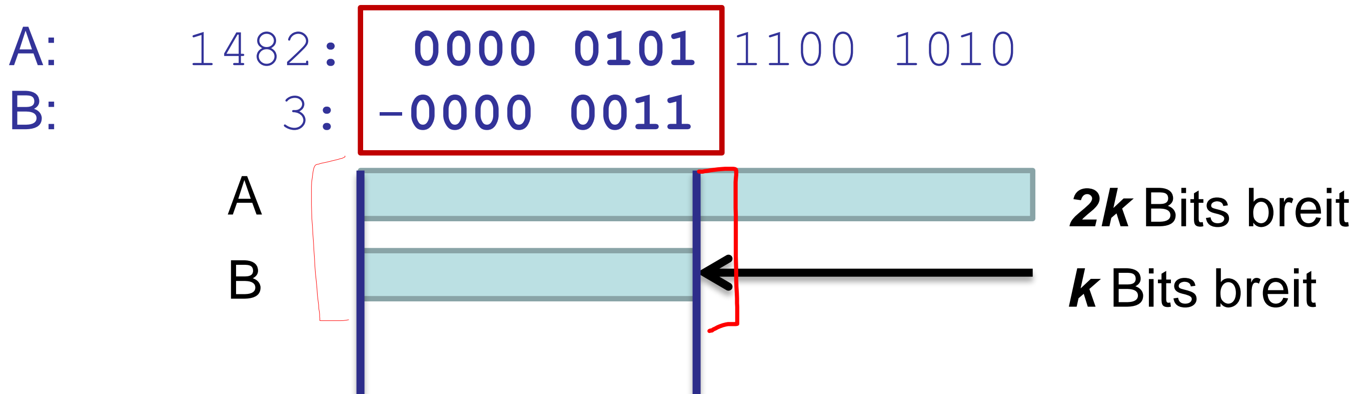
1. Vergleiche:
   A[2k-1:k] mit *B[k-1:0]*
2. Wenn B passt, Überlauf
   Wenn nicht, kein Überlauf

TECHNISCHE UNIVERSITÄT DARMSTADT
Embedded Systems & Applications

Abfangen von Überläufen

- **Beispiel:**

  k: 8 bit
  Operanden: A (16b) = 1482, B(8b) = 3
  Ergebnis: Q (8b) = 494 **Überlauf!**
  R (8b) = 0



1. Vergleiche:
   A[2k-1:k] mit *B[k-1:0]*
2. **Wenn B passt, Überlauf!**
   Wenn nicht, kein Überlauf

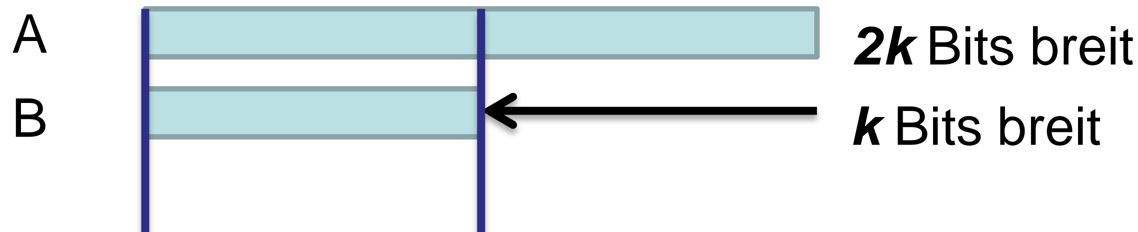
Abfangen von Überläufen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Beispiel 2:**

k: 8 bit
Operanden: A (16b) = 1482, B(8b) = 6
Ergebnis: Q (8b) = 247 **kein Überlauf**
R (8b) = 0

```
A:   1482:  0000 0101 1100 1010
B:      6: -0000 0110
```



1. Vergleiche:
   A[2k-1:k] mit *B[k-1:0]*
2. Wenn B passt, Überlauf
   Wenn nicht, kein Überlauf

Embedded Systems & Applications

Dividierer Algorithmus mit unterschiedlichen Bitbreiten

A/B = Q + R
A: 8-bit breit
B,Q,R: 4-bit breit

Binär: 01011100/1010=1001 R0010
(92/10 = 9 R 2)

```
R' = A_{2k-1:k}
D = R' – B
if (D < 0) {
  for i = k-1 to 0 {
    R = {R' << 1, A_i}
    D = R - B
    if D < 0, Q_i= 0;  R'= R
    else       Q_i= 1;  R'= D
  }
R=R'
}
else Überlauf
```

```
 01011100
-1010
 1011
```
Negativ (kein Überlauf)

```
 1011100
-1010
 0001
```
Passt: $Q_3 = 1$

```
 001100
-1010
 1001
```
Passt nicht: $Q_2 = 0$

```
 01100
-1010
 1100
```
Passt nicht: $Q_1 = 0$

```
 1100
-1010
 0010
```
der Rest → 0010

Passt: $Q_2 = 1$

TECHNISCHE UNIVERSITÄT DARMSTADT

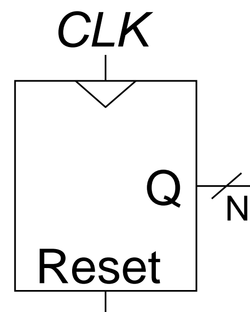
Embedded Systems & Applications

Zähler

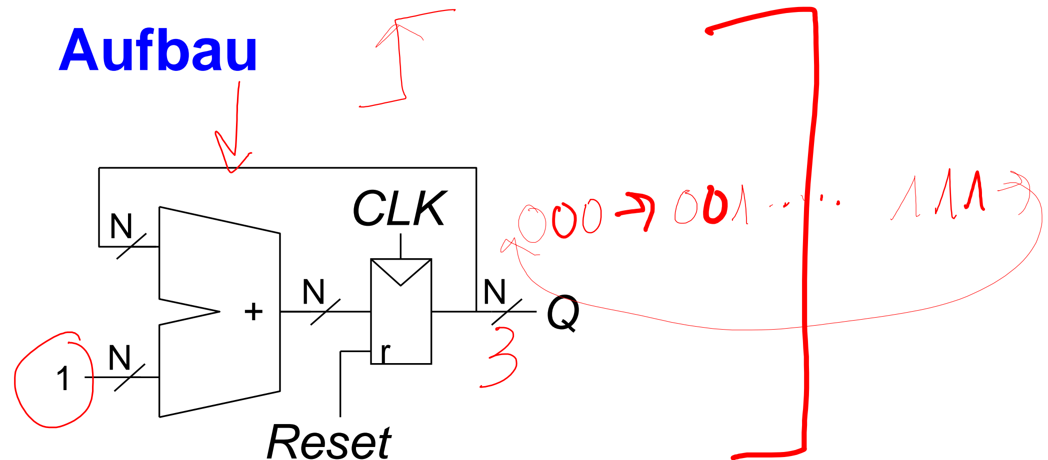
TECHNISCHE UNIVERSITÄT DARMSTADT

- Einfachster Fall: Inkrementieren zu jeder positiven Taktflanke
- Zählen durch einen Zyklus von Werten, Beispiel für 3b Breite
  - 000, 001, 010, 011, 100, 101, 110, 111, 000, 001…
- Beispielanwendungen
  - Digitaluhren
  - Programmzähler: Zeigt auf nächste auszuführende Instruktion

Symbol



Aufbau

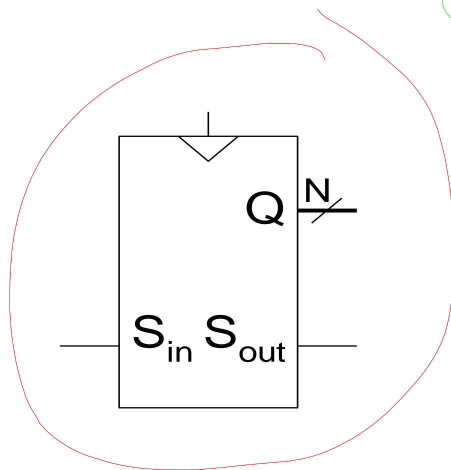


Embedded Systems & Applications

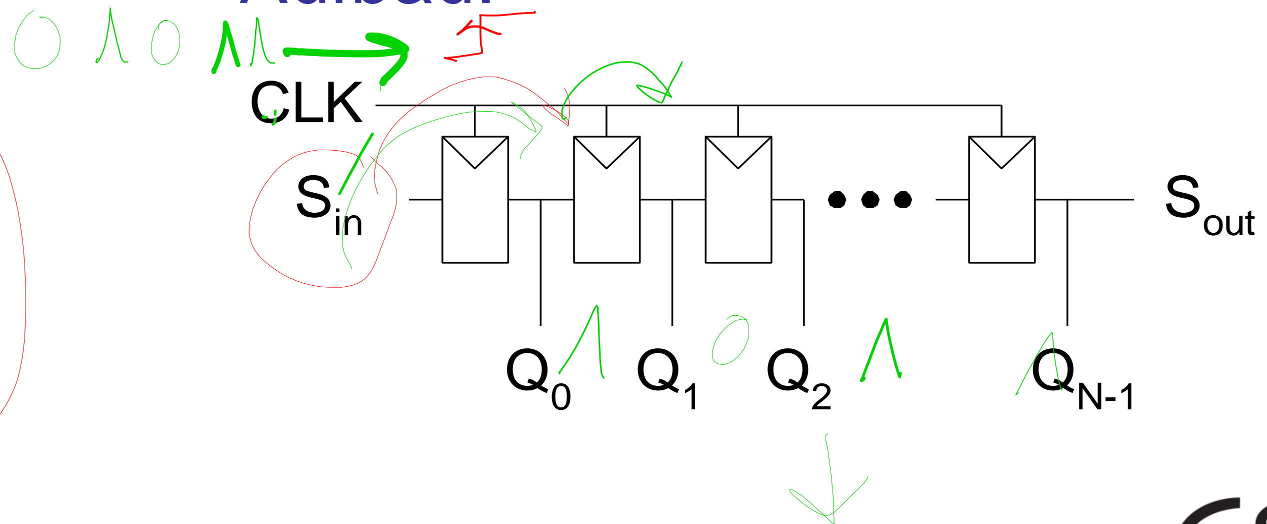
Schieberegister

- Auch: FIFO (*first-in first-out*)
- Schiebe einen neuen Wert jeden Takt ein
- Schiebe einen alten Wert jeden Takt aus
- Kann auch agieren als Seriell-nach-Parallel-Konverter
  - Konvertiert serielle Eingabe (S_{in}) in parallele Ausgabe ($Q_{0:N-1}$)

Symbol:



Aufbau:



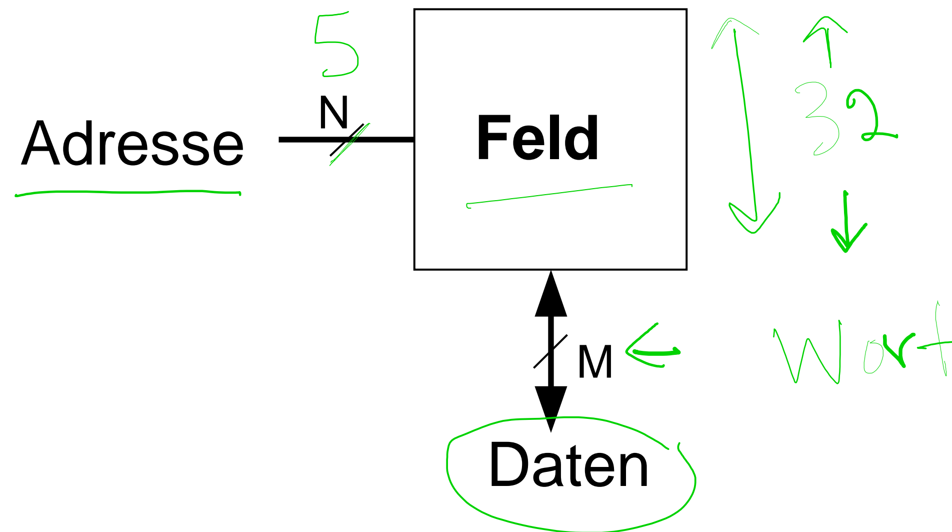
Embedded Systems & Applications

Schieberegister mit parallelem Laden

- Bei *Load* = 1: Agiert als normales *N*-bit Register
- Bei *Load* = 0: Agiert als Schieberegister
- Verwendbar als
  - Seriell-nach-Parallelkonverter (S_{in} nach $Q_{0:N-1}$)
  - Parallel-nach-Seriellkonverter ($D_{0:N-1}$ nach S_{out})

Speicherfelder

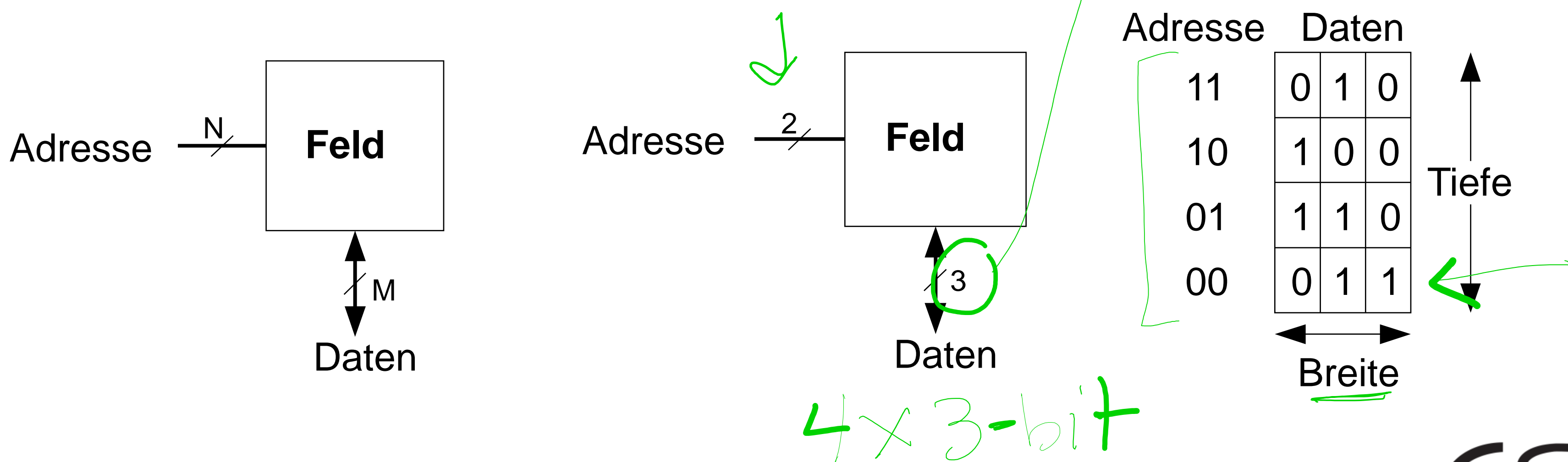
- Können effizient größere Datenmengen speichern
- An jede *N*-bit Adresse kann ein *M*-bit breites Datum geschrieben werden



Embedded Systems & Applications

Speicherfelder

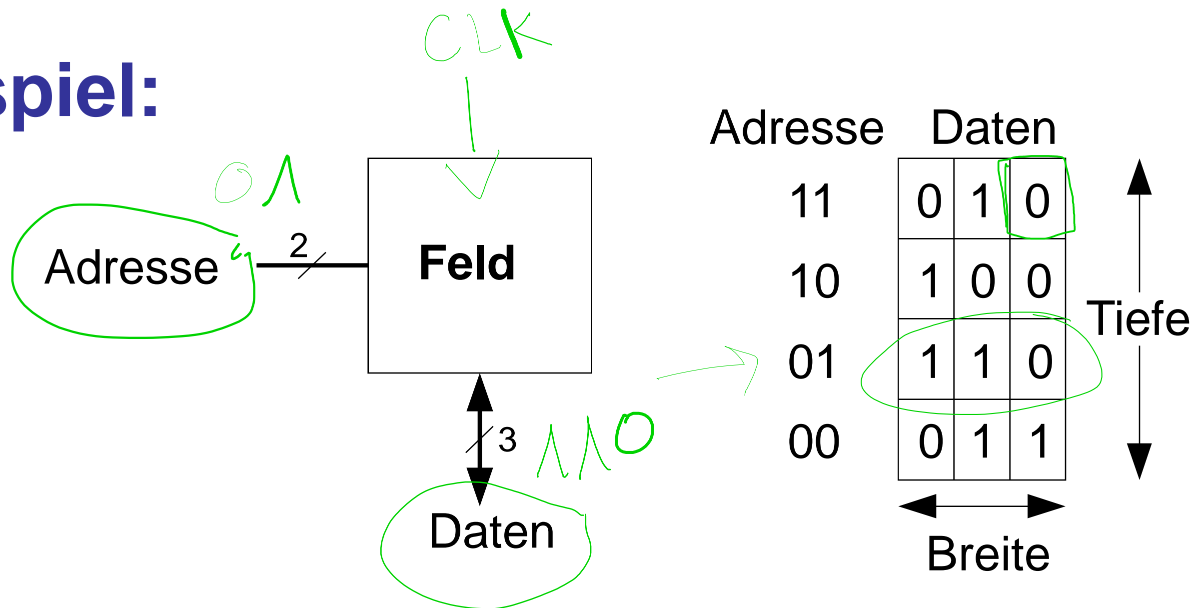
- Zweidimensionales Feld von Bit-Zellen
- Jede Bit-Zelle speichert ein Bit
- Feld mit *N* Adressbits und *M* Datenbits:
  - 2^N Zeilen und *M* Spalten
  - **Tiefe:** Anzahl von Zeilen (Anzahl von Worten)
  - **Breite:** Anzahl von Spalten (Bitbreite eines Wortes)
  - **Feldgröße:** Tiefe × Breite = $2^N \times M$



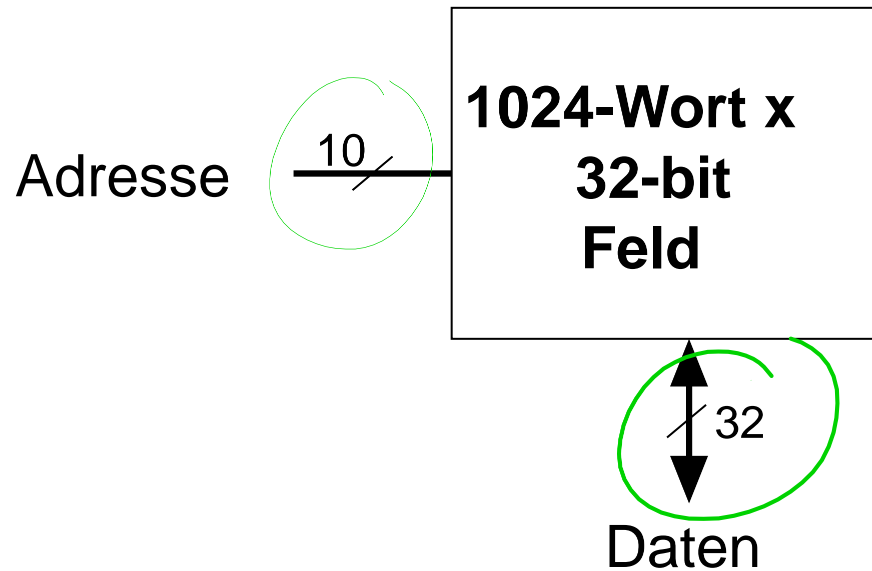
Beispiel: Speicherfeld

- $2^2 \times$ 3-Bit Feld
- Anzahl Worte: 4
- Wortbreite: 3-Bit
- Beispiel: 3-Bit gespeichert an Adresse 2'b10 ist 3'b100

Beispiel:

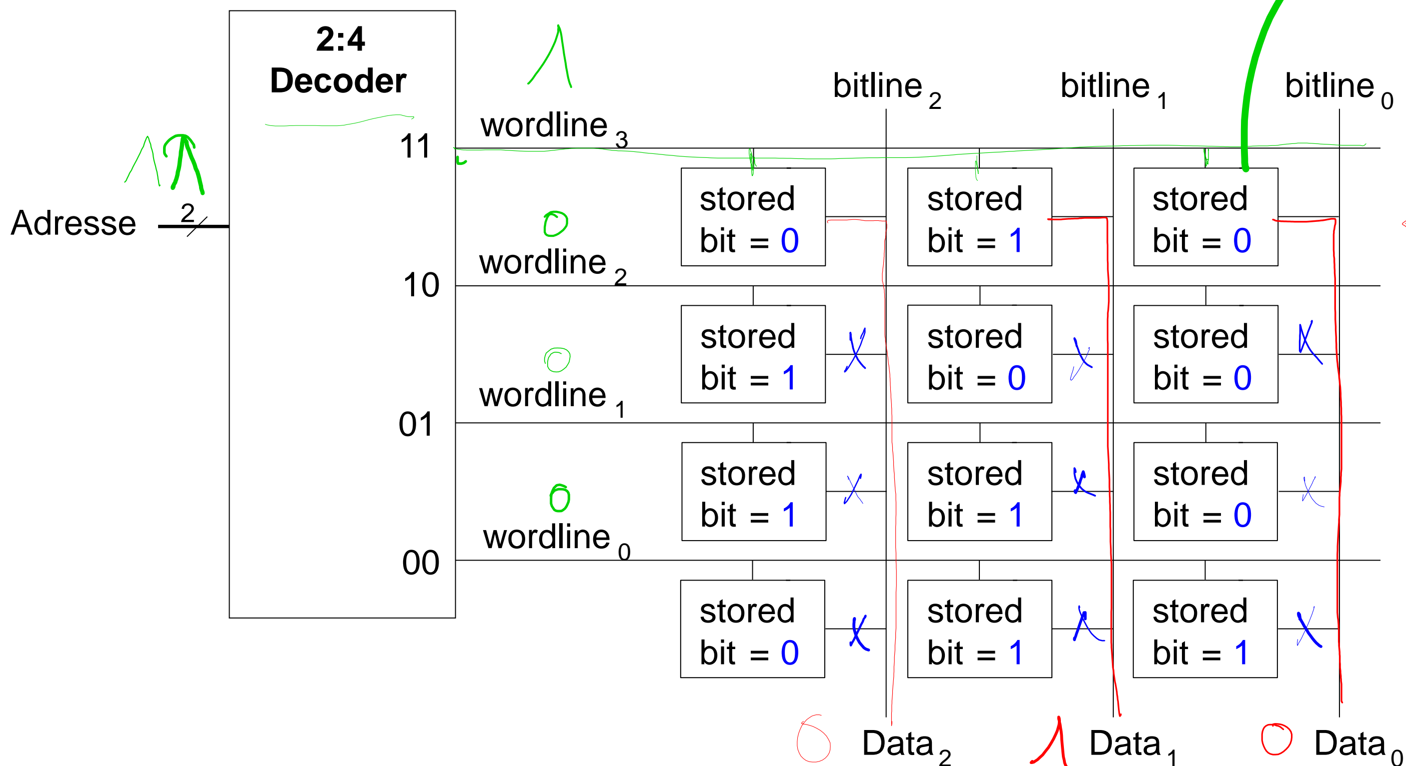


Speicherfelder

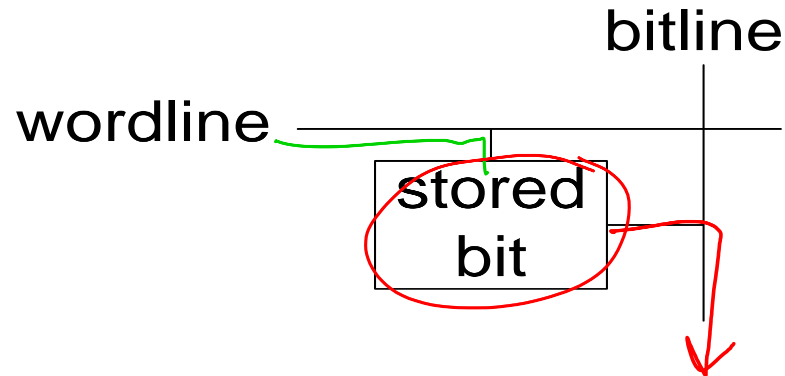


Aufbau von Speicherfeldern

- **Wordline:**
  - Vergleichbar mit Enable-Signal
  - Erlaubt Zugriff auf eine Zeile des Speichers zum Lesen oder Schreiben
  - Entspricht genau einer eindeutigen Adresse
  - Maximal eine Wordline ist zu jedem Zeitpunkt HIGH

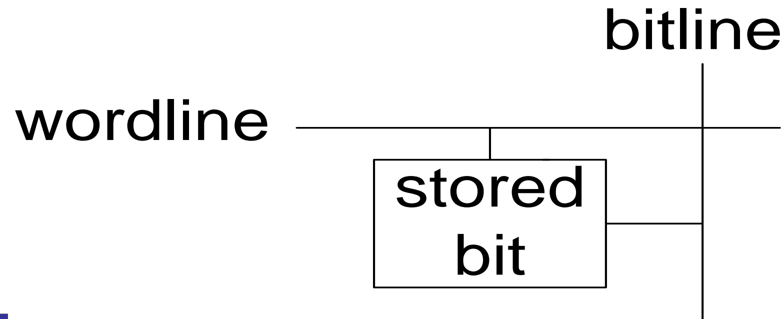


Bit-Zellen für Speicherfelder

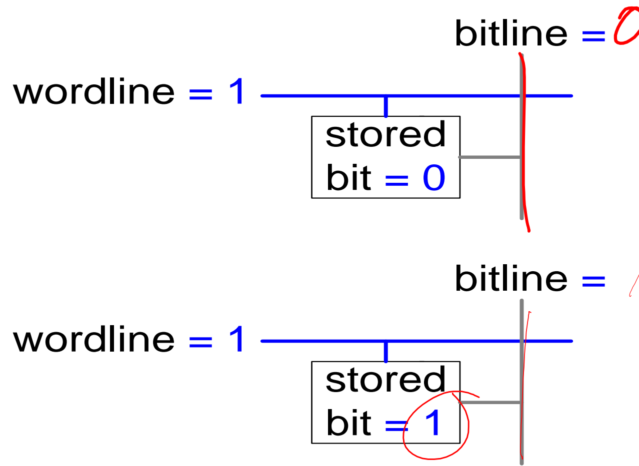


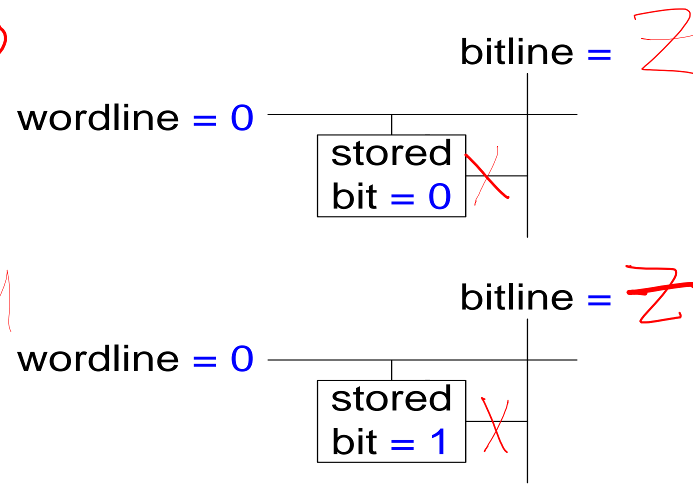
Bit-Zellen für Speicherfelder

TECHNISCHE UNIVERSITÄT DARMSTADT



Beispiel:

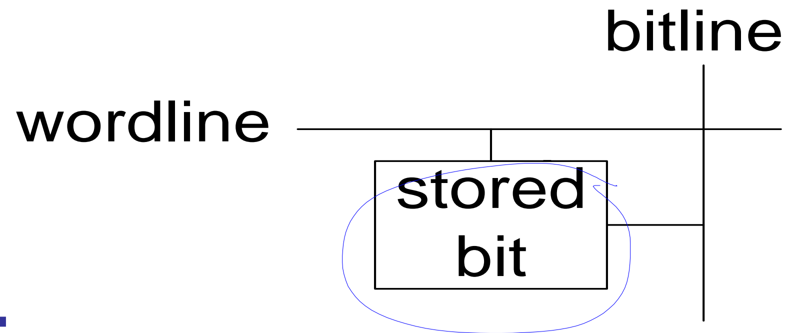




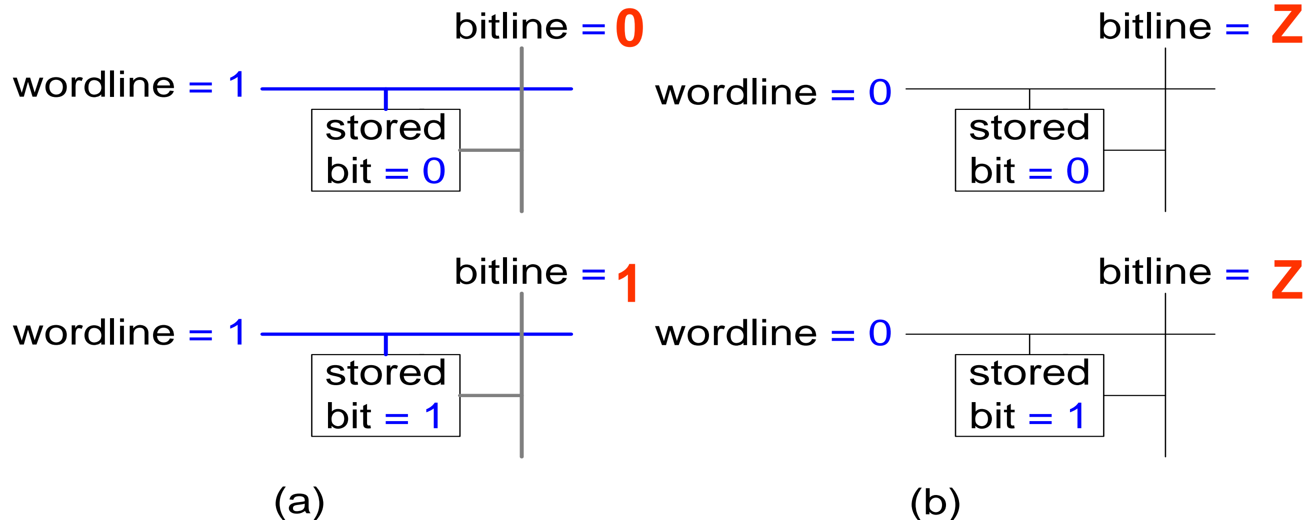
Embedded Systems & Applications

Bit-Zellen für Speicherfelder

TECHNISCHE UNIVERSITÄT DARMSTADT



Beispiel:



Embedded Systems & Applications

Arten von Speicher: Historische Sicht

- Speicher mit wahlfreiem Zugriff (RAM)
- Nur-Lese Speicher (ROM)

Arten von Speicher: Historische Sicht

TECHNISCHE UNIVERSITÄT DARMSTADT

- Speicher mit wahlfreiem Zugriff (RAM) **jetzt → flüchtig**
- Nur-Lese Speicher (ROM) **jetzt → nicht flüchtig**

Embedded Systems & Applications

RAM: Random-Access Memory

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Flüchtig:** Speicherinhalte gehen bei Verlust der Betriebsspannung verloren
- Kann i.d.R. gleich schnell gelesen und geschrieben werden
- Zugriff auf beliebige Adressen mit ähnlicher Verzögerung möglich
- **Hauptspeicher moderner Computer ist dynamisches RAM (DRAM)**
  - Aktuell & genauer: DDR3-SDRAM
  - *Double Data Rate 3 - Synchronous Dynamic Random Access Memory*
- Name „RAM" ist historisch gewachsen
  - Früher unterschiedliche Zugriffszeiten auf unterschiedliche Adressen
    - Bandspeicher, Trommelspeicher, Ultraschall-Laufzeitspeicher, …

Embedded Systems & Applications

ROM: Read-Only Memory

- **Nicht-flüchtig:** Erhält Speicherinhalt auch ohne Betriebsspannung
- Schnell lesbar
- Schreibbar nur sehr langsam (wenn überhaupt)
- **Flash-Speicher ist in diesem Sinne ein ROM**
  - Kameras
  - Handys
  - MP3-Player
- Auch hier Nomenklatur „ROM“ historisch
  - Auch aus ROMs kann von beliebigen Adressen gelesen werden
  - Es gibt auch schreibbare Arten von ROMs
    - PROMs, EPROMs, EEPROMs, Flash

Embedded Systems & Applications

Arten von RAM

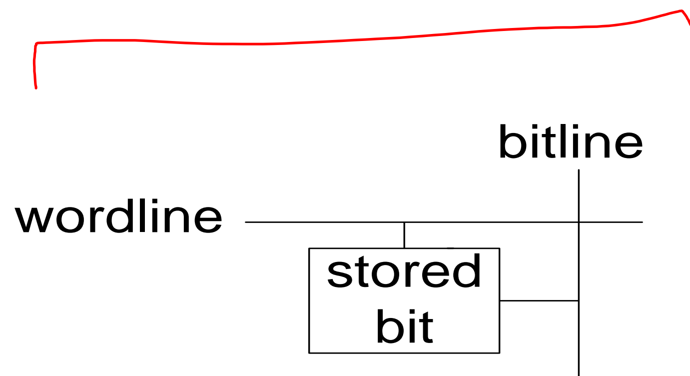
- Zwei wesentliche Typen:
  - Dynamisches RAM (**DRAM**)
  - Statisches RAM (**SRAM**)
- Verwenden unterschiedliche Speichertechniken in den Bit-Zellen:
  - DRAM: Kondensator
  - SRAM: Kreuzgekoppelte Inverter

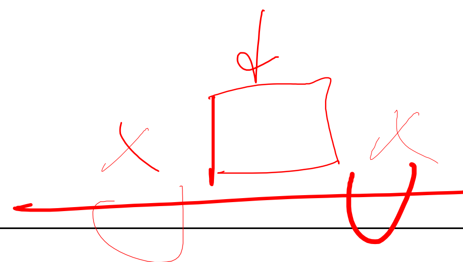
Robert Dennard, 1932 -

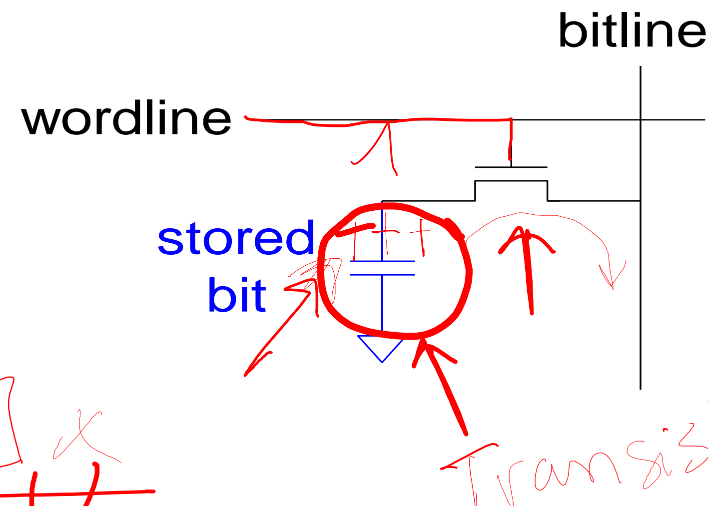
- Erfand 1966 bei IBM das DRAM
- Anfangs große Skepsis, ob Technik praktikabel
- Seit Mitte der 1970er Jahre ist DRAM die am weitesten verbreitete Speichertechnik in Computern

DRAM Bit-Zelle

- Datenbit wird als Ladezustand eines Kondensators gespeichert
- Dynamisch: Der Speicherwert muss periodisch neu geschrieben werden
  - Auffrischung alle paar Millisekunden erforderlich (üblich: 64ms)
  - Kondensator verliert Ladung durch Leckströme
  - … und beim Auslesen







DRAM Bit-Zelle

TECHNISCHE UNIVERSITÄT DARMSTADT

3V ≙ 1

bitline

wordline

gespeicherter Wert = 1

bitline

wordline

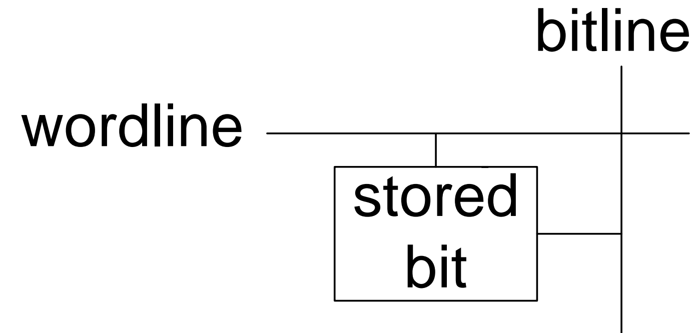
gespeicherter Wert = 0

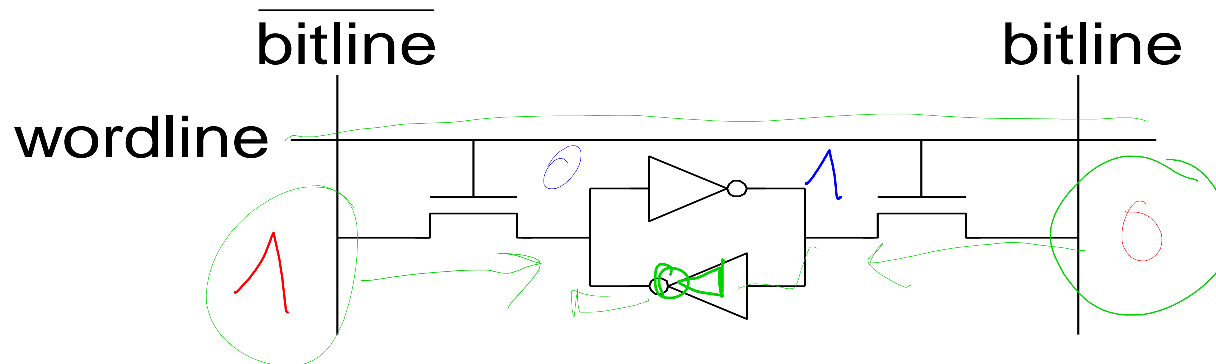
Embedded Systems & Applications

SRAM Bit-Zelle

TECHNISCHE UNIVERSITÄT DARMSTADT

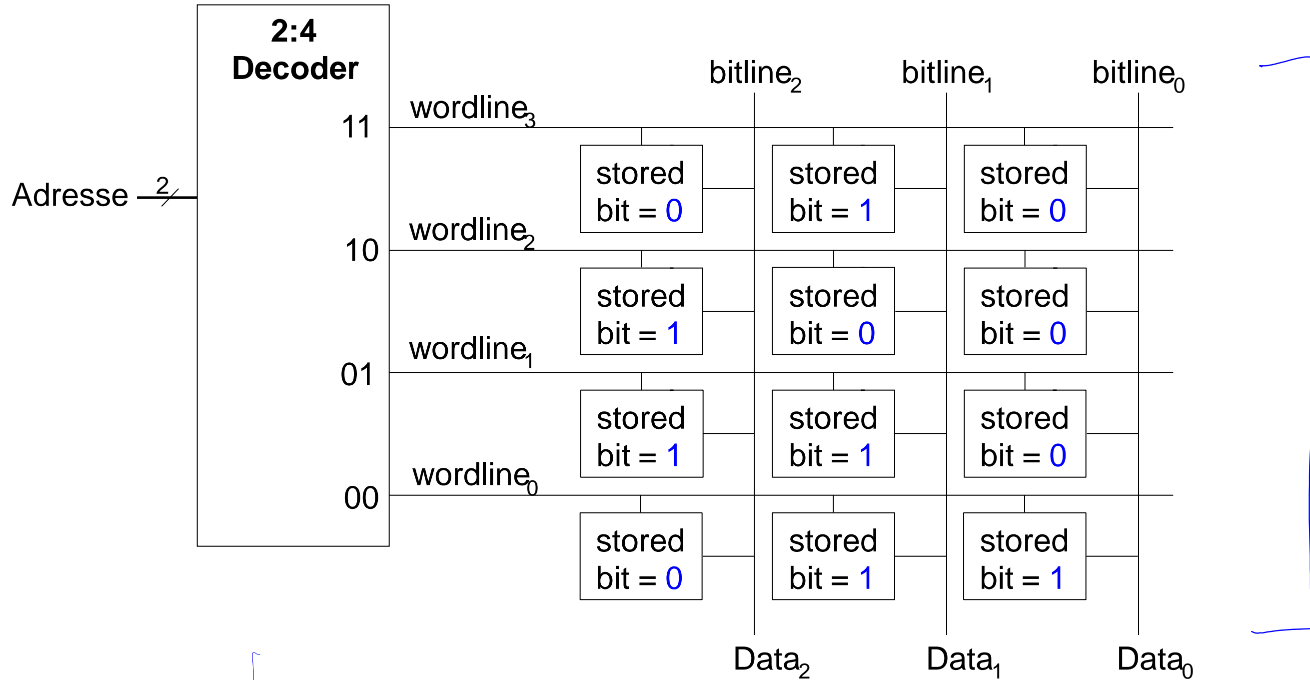
- Datenbit wird als Zustand von rückgekoppelten Invertern gespeichert
- Statisch: Keine Auffrischung erforderlich
  - Inverter treiben Werte auf gültige Logikpegel



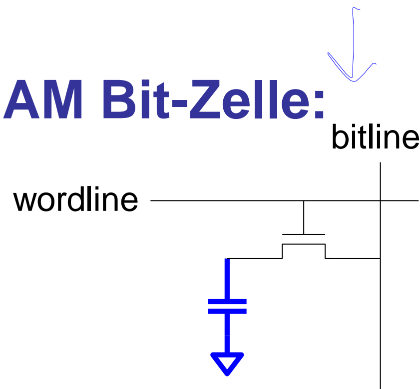


Embedded Systems & Applications

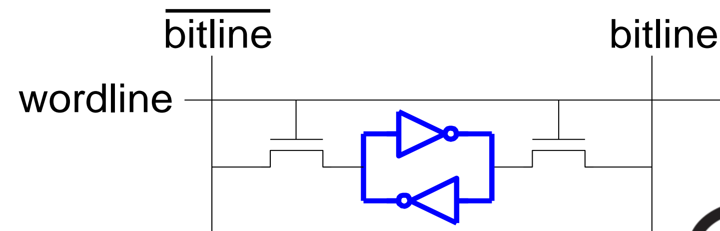
Speicherfelder



DRAM Bit-Zelle:



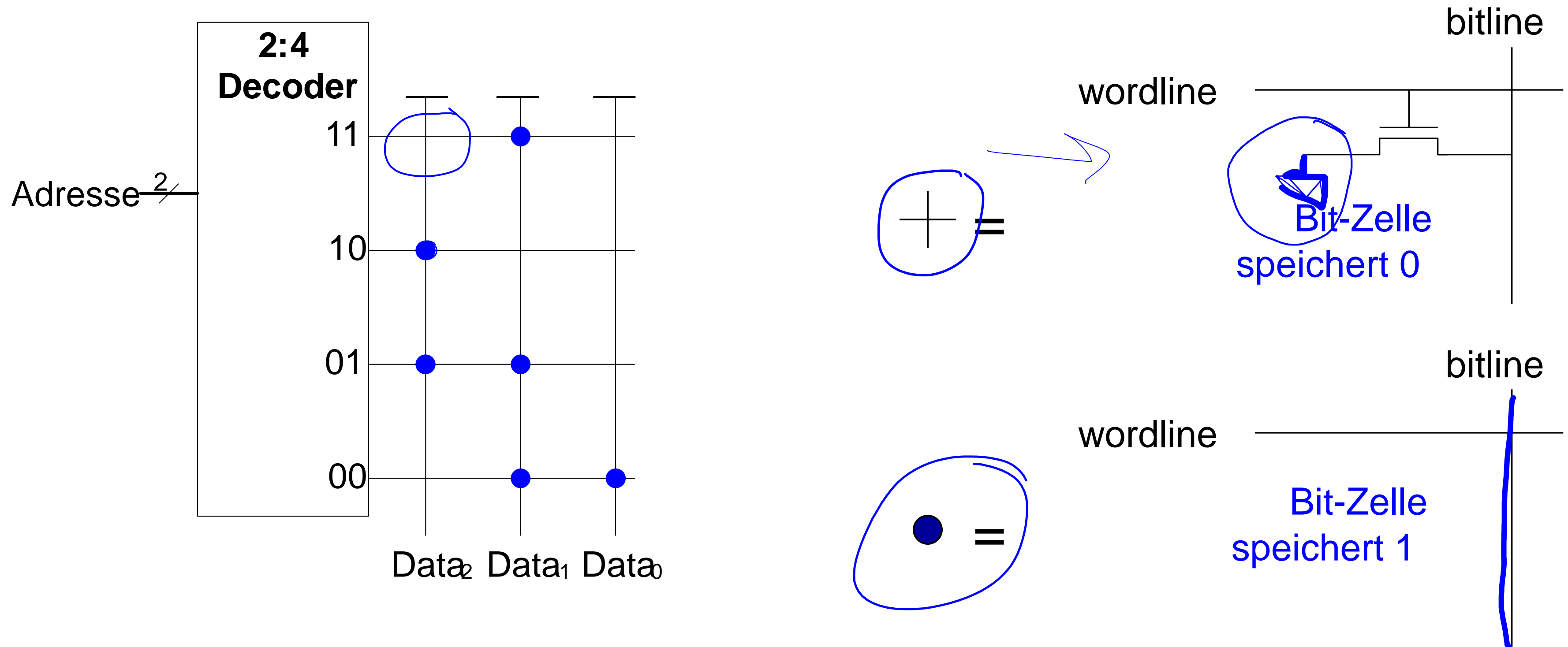
SRAM Bit-Zelle:



Embedded Systems & Applications

ROMs: Aufbau der Bit-Zellen

TECHNISCHE UNIVERSITÄT DARMSTADT



Bitlines sind schwach auf HIGH getrieben

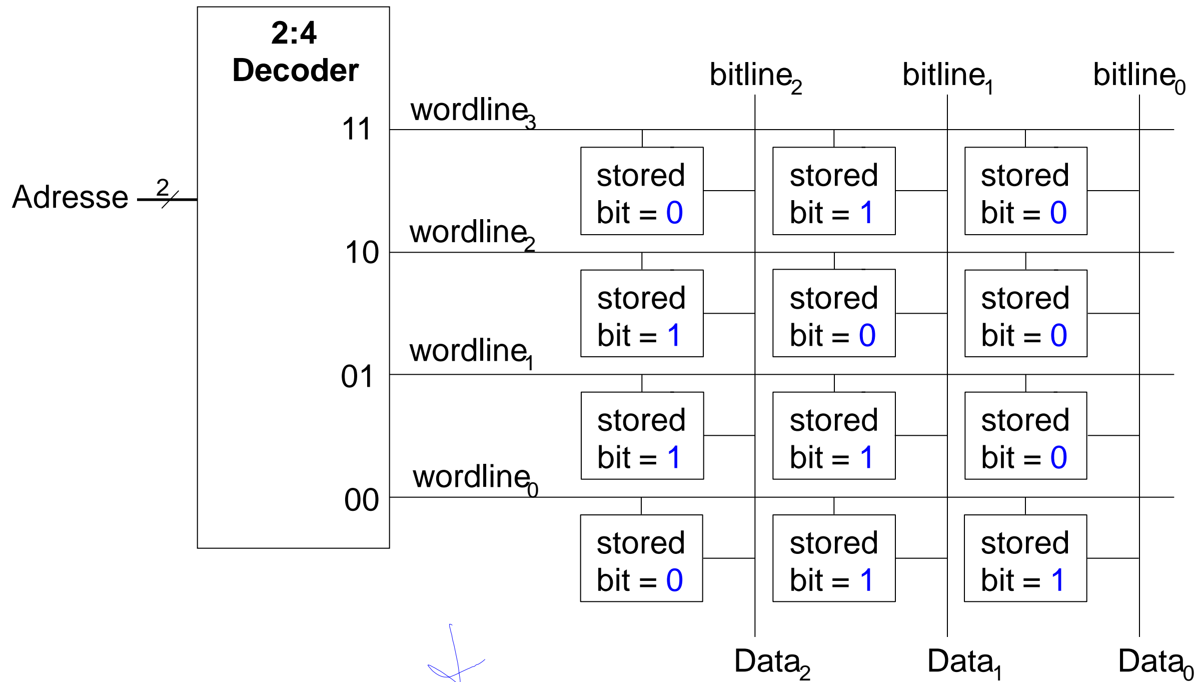
Embedded Systems & Applications

Fujio Masuoka, 1944-

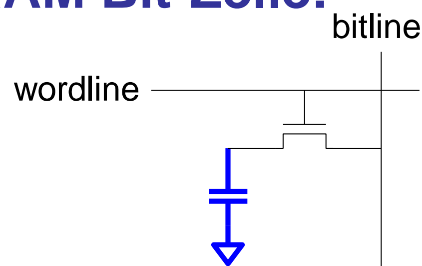
- Entwickelte Speicher und schnelle Schaltungen bei Toshiba von 1971-1994
- Erfand Flash-Speicher als eigenes ungenehmigtes Projekt in den späten 1970ern
  - An Wochenenden und Abends
- Löschvorgang erinnerte ihn an Kamerablitz
  - Deshalb Flash-Speicher
- Toshiba kommerzialisierte Technik nur zögerlich
- Erste kommerzielle Chips von Intel in 1988
- Flash-Produkte haben großen Erfolg
  - Derzeit USD 25 Milliarden Umsatz / Jahr

Embedded Systems & Applications

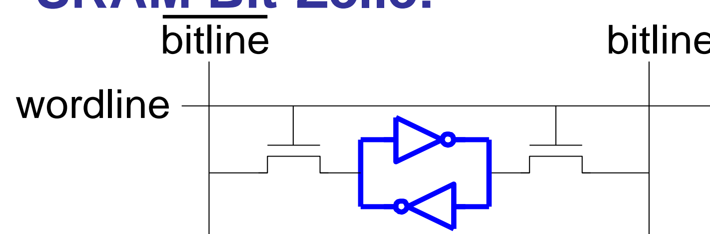
Speicherfelder



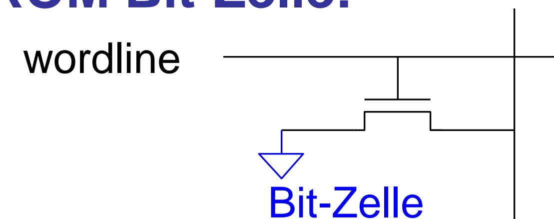
DRAM Bit-Zelle:



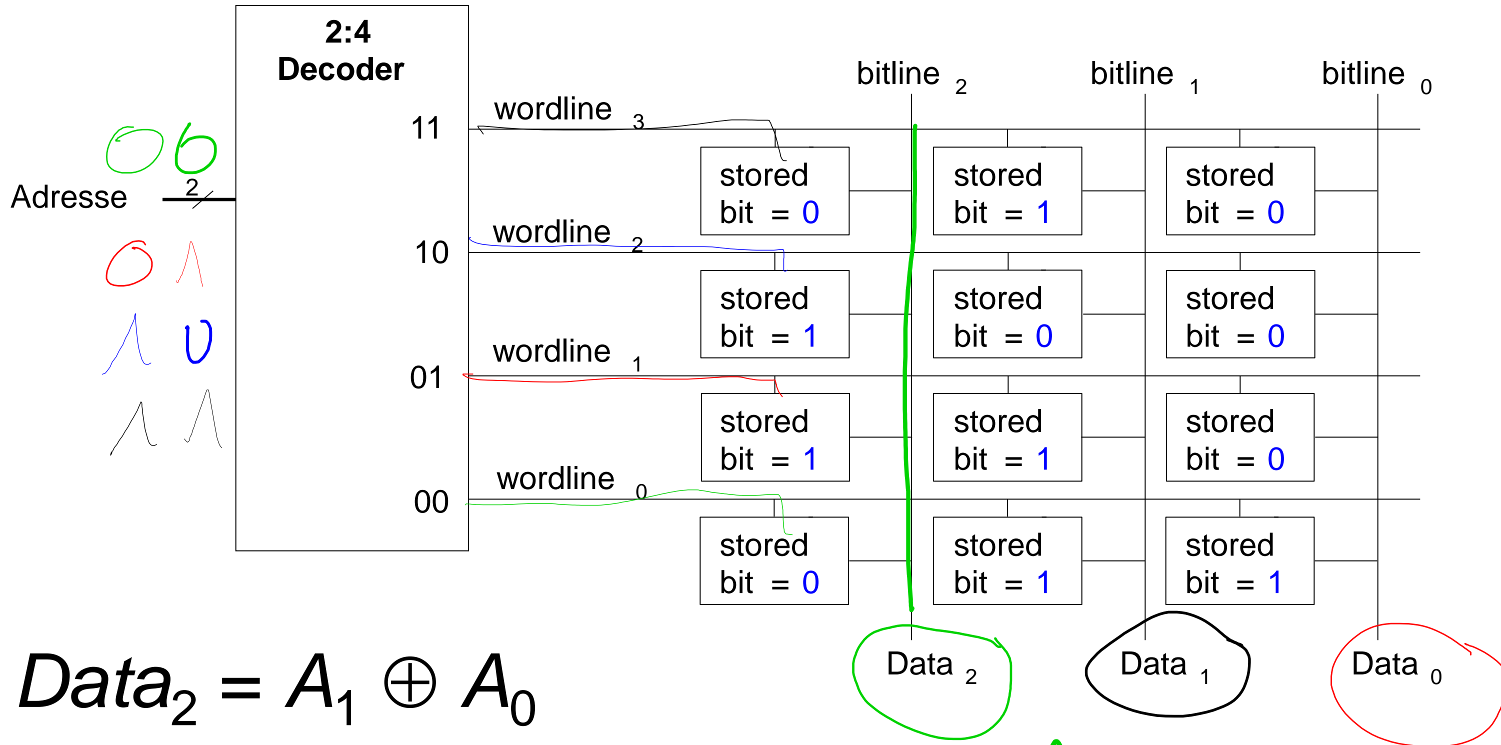
SRAM Bit-Zelle:



ROM Bit-Zelle:



Logik aus beliebigem Speicherfeld



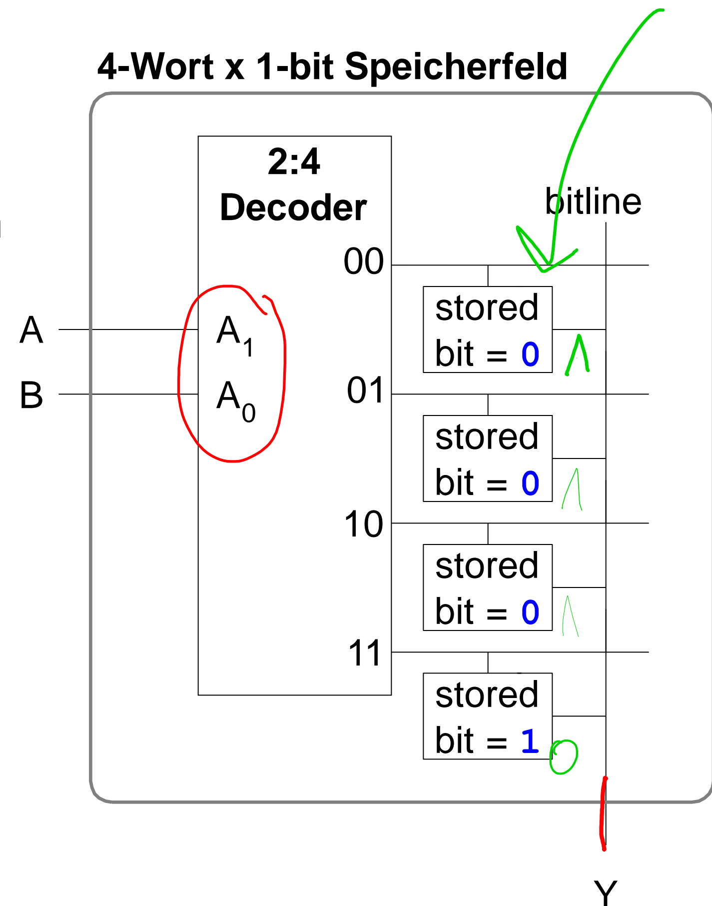
$$Data_2 = A_1 \oplus A_0$$

$$Data_1 = \overline{A_1} + A_0$$

$$Data_0 = \overline{A_1}\,\overline{A_0}$$

Logik aus beliebigen Speicherfeldern

- Speicherfelder speichern Wertetabellen
  - Lookup-Tables (LUTs)
- Wort aus Eingangsvariablen bildet Adresse
- Für jede Kombination von Eingangsvariablen ist Funktionsergebnis abgespeichert



Werte-tabelle

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

Logikfelder (*logic arrays*)

Programmable Logic Arrays (PLAs)

- AND Feld gefolgt von OR Feld
- Kann nur kombinatorische Logik realisieren
- Feste interne Verbindungen, spezialisiert für DNF (SoP-Form)

Field Programmable Gate Arrays (FPGAs)

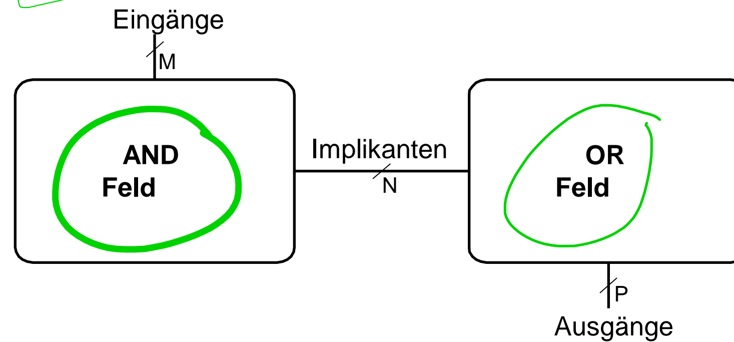
- Feld von konfigurierbaren Logikblöcken (CLBs)
- Können kombinatorische und sequentielle Logik realisieren
- Programmierbare Verbindungsknoten zwischen Schaltungselementen

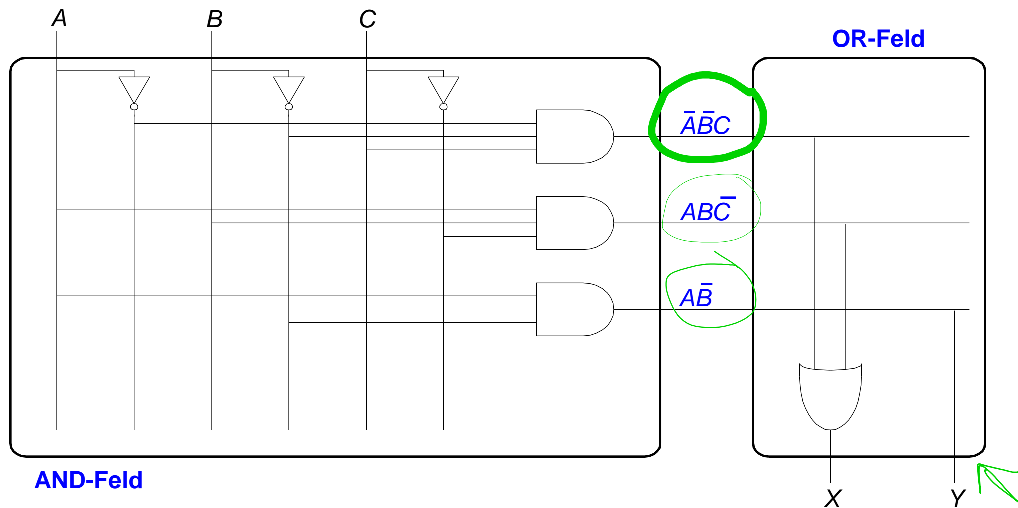
Embedded Systems & Applications

Boole‘sche Funktionen mit PLAs: Idee

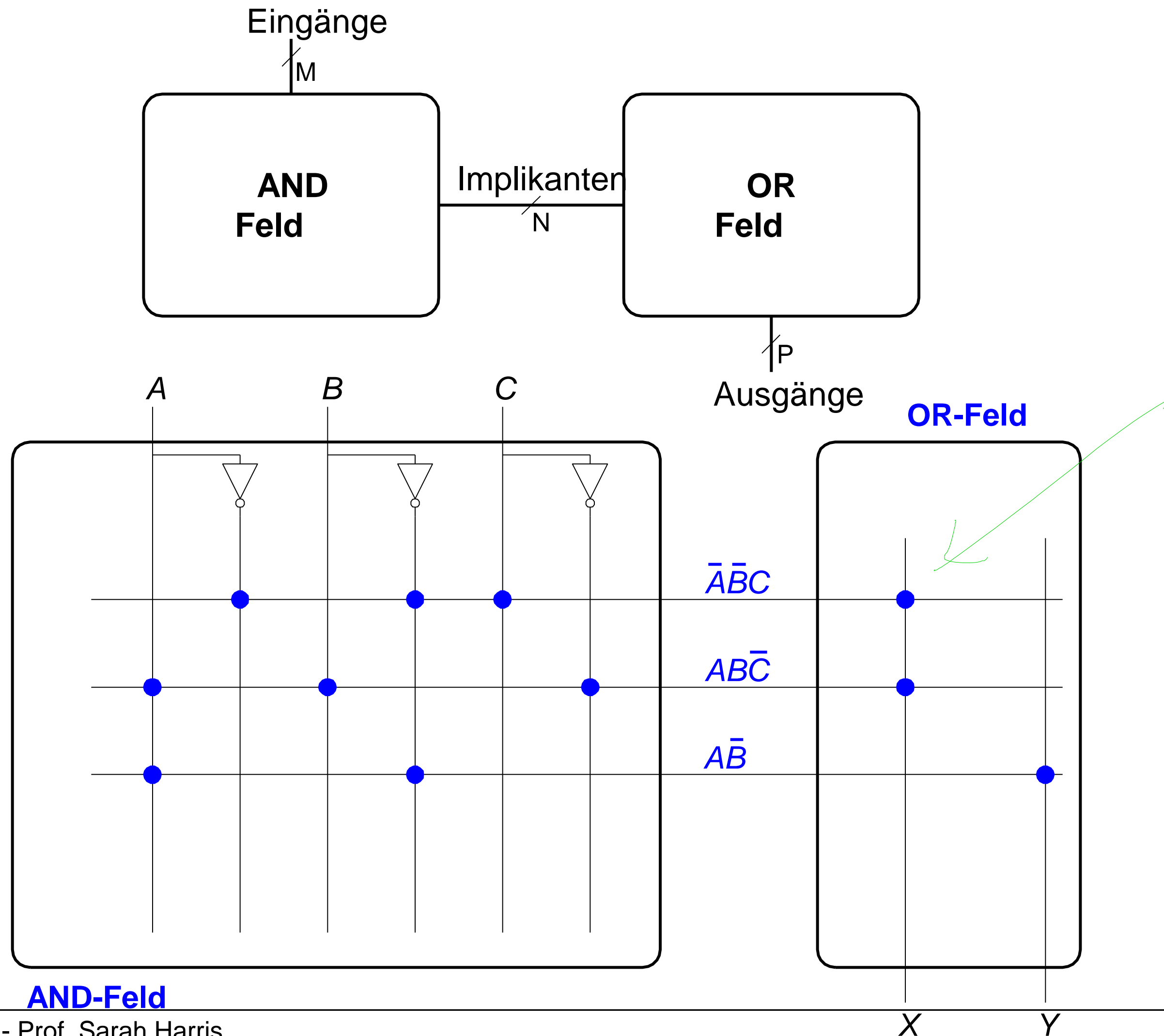
- $X = \bar{A}\bar{B}C + AB\bar{C}$
- $Y = A\bar{B}$

DNF (SOP)





PLAs: Vereinfachte Schreibweise



FPGAs: Field Programmable Gate Arrays

TECHNISCHE UNIVERSITÄT DARMSTADT

Bestehen grundsätzlich aus:

- **CLBs (Configurable Logic Blocks):** Realisieren kombinatorische und sequentielle Logik
  - Konfigurierbare Logikblöcke
- **IOBs (Input/Output Blocks):** Schnittstelle vom Chip zur Außenwelt
  - Ein-/Ausgabeblöcke
- **Programmierbares Verbindungsnetz:** verbindet CLBs und IOBs
  - Kann flexibel Verbindungen je nach Bedarf der aktuellen Schaltung herstellen

Embedded Systems & Applications

FPGAs: Field Programmable Gate Arrays

TECHNISCHE UNIVERSITÄT DARMSTADT

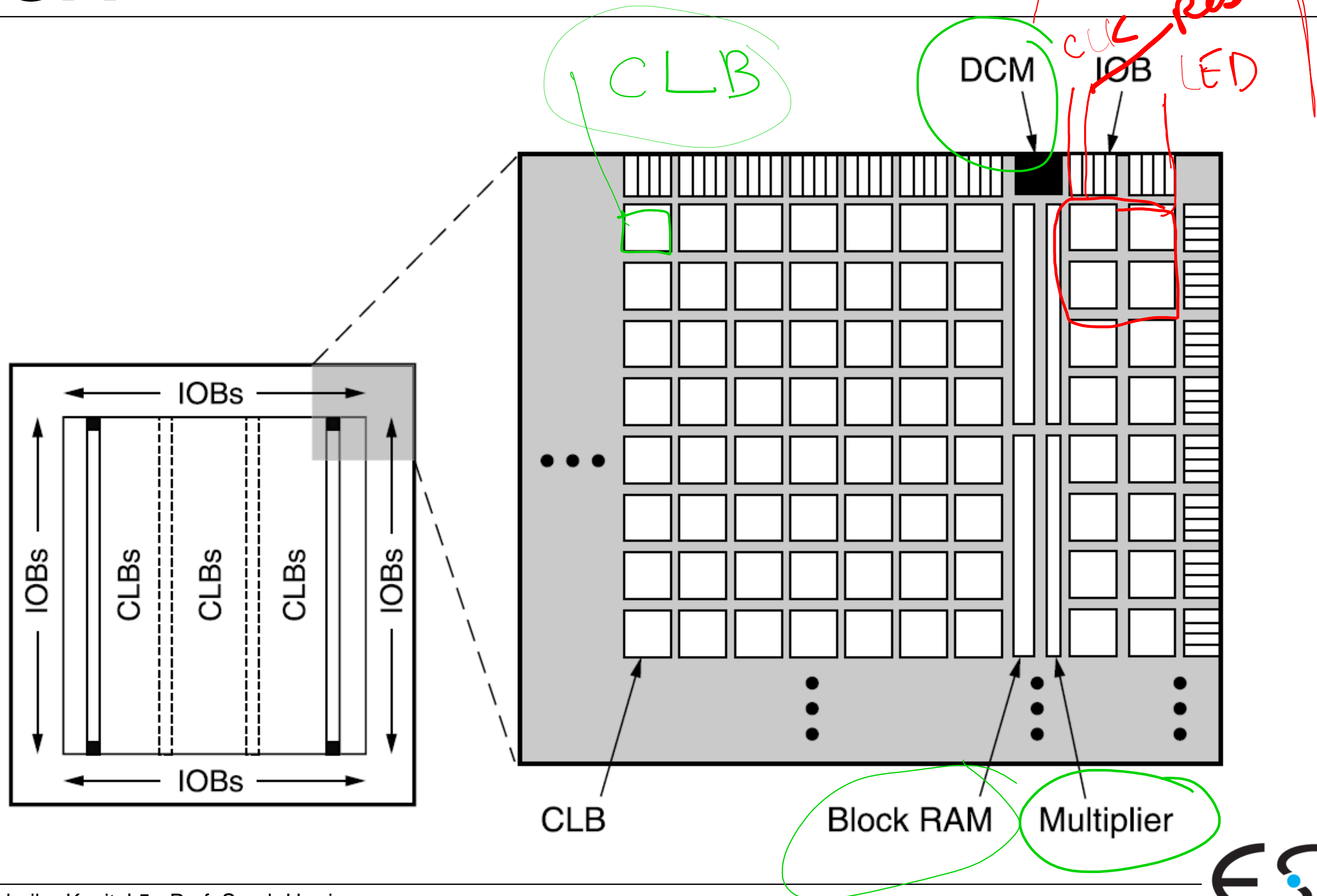
Reale FPGAs enthalten oftmals noch weitere Arten von Blöcken

- RAM
- Multiplizierer
- Manipulation von Taktsignalen (DCM)
- Sehr schnelle serielle Verbindungen (11 Gb/s)
- Komplette Mikroprozessoren
- …

Embedded Systems & Applications

Struktur eines Xilinx Spartan 3 FPGA

TECHNISCHE
UNIVERSITÄT
DARMSTADT



Embedded Systems & Applications

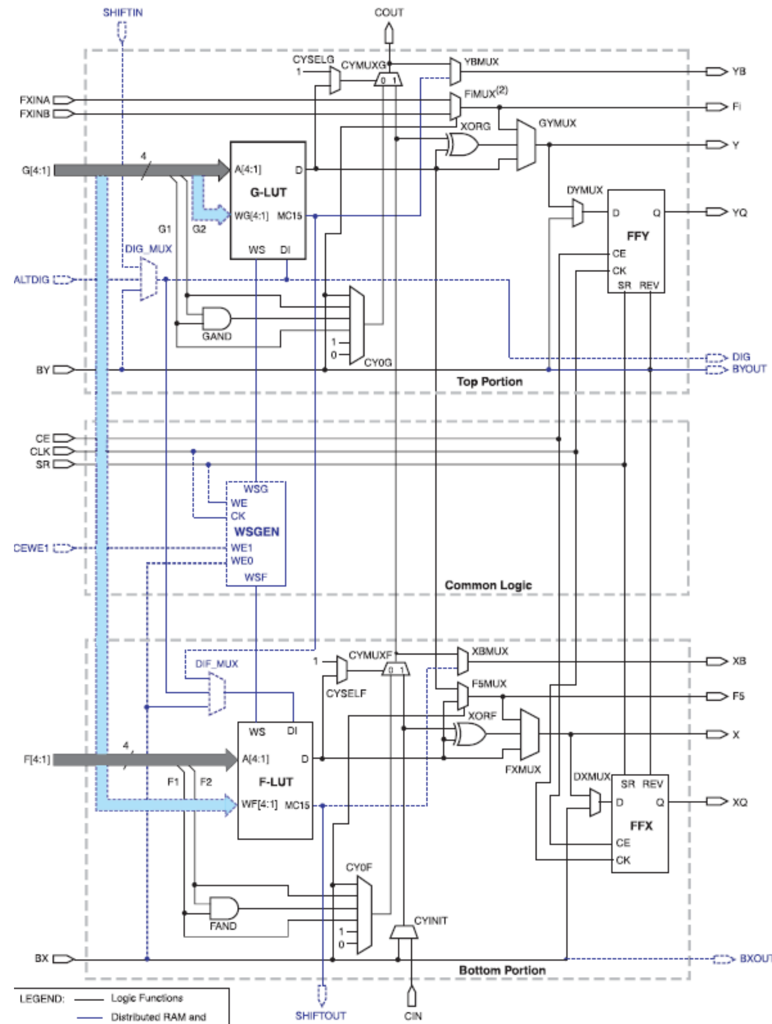
Konfigurierbare Logikblöcke (CLBs)

Bestehen im wesentlichen aus:

- **LUTs** (lookup tables): realisieren kombinatorische Funktionen
- **Flip-Flops**: realisieren sequentielle Funktionen
- **Multiplexern**: Verbinden LUTs und Flip-Flops

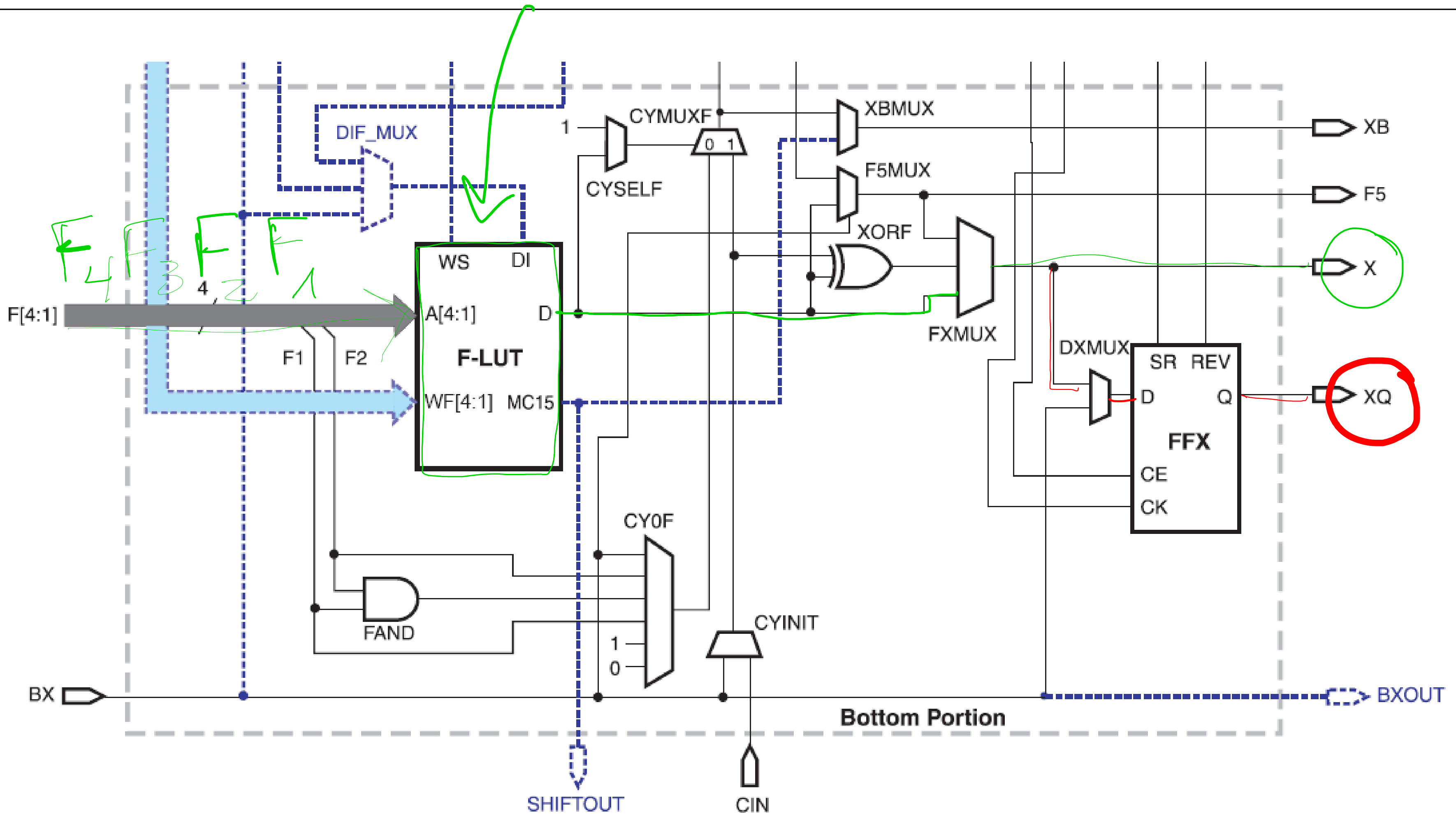
Embedded Systems & Applications

Xilinx Spartan3 CLB



Xilinx Spartan3 CLB

TECHNISCHE UNIVERSITÄT DARMSTADT



Embedded Systems & Applications

Xilinx Spartan3 CLB

Ein Spartan3 CLB enthält:

- **2 LUTs:**
  - F-LUT (2^4 x 1-bit LUT)
  - G-LUT (2^4 x 1-bit LUT)
- **2 sequentielle Ausgänge:**
  - *XQ*
  - *YQ*
- **2 kombinatorische Ausgänge:**
  - *X*
  - *Y*

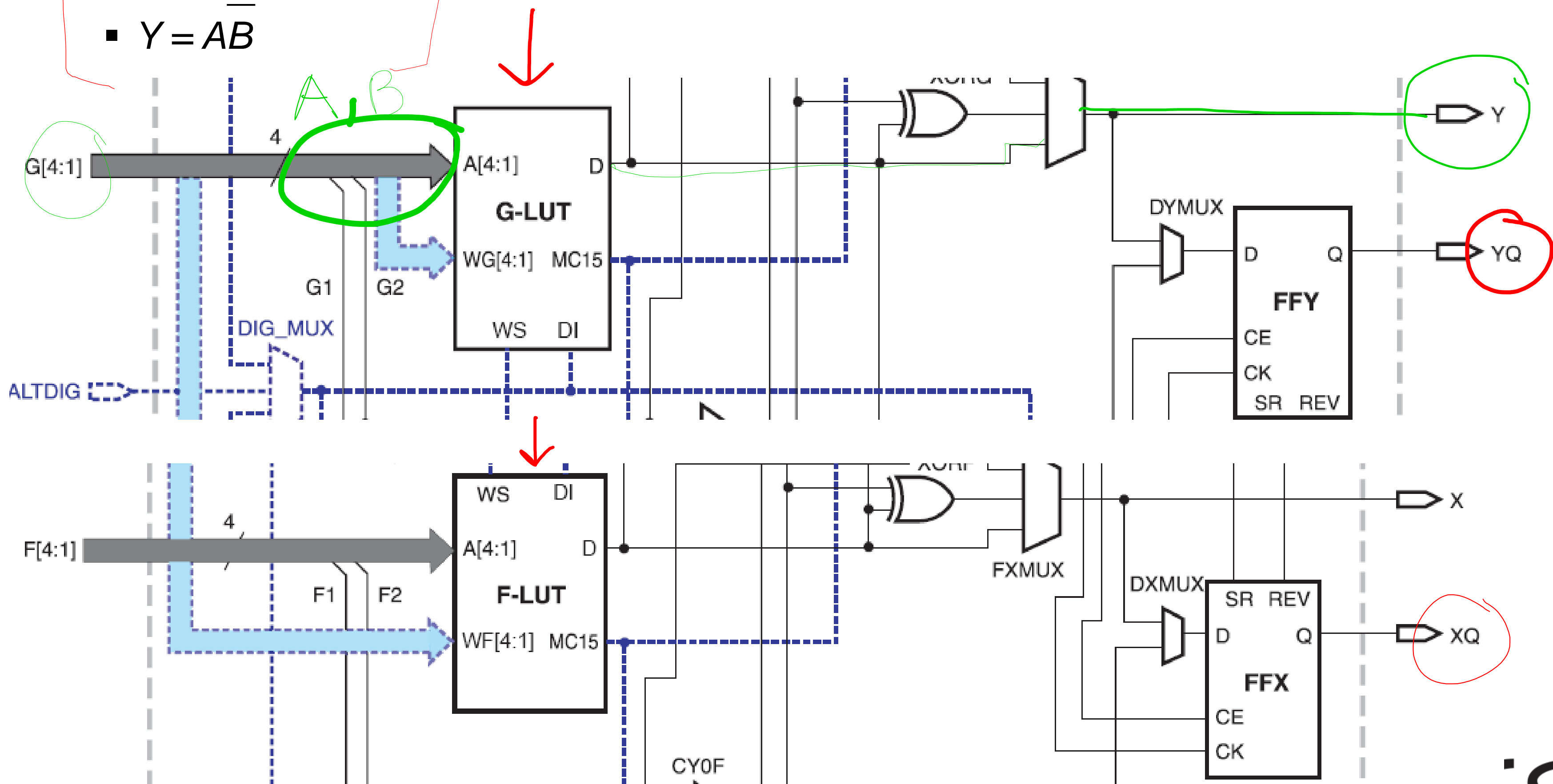
Embedded Systems & Applications

Beispiel: Kombinatorische Logik mit CLBs

TECHNISCHE UNIVERSITÄT DARMSTADT

Berechnung der folgenden Funktionen mit dem Spartan3 CLB

- $X = \overline{A}\,\overline{B}C + AB\overline{C}$
- $Y = A\overline{B}$



Embedded Systems & Applications

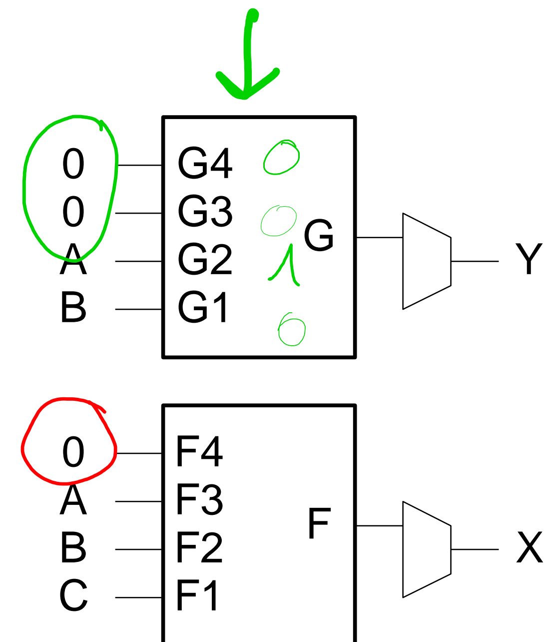
Beispiel: Kombinatorische Logik mit CLBs

Berechnung der folgenden Funktionen mit dem Spartan 3 CLB

- $X = \overline{A}\,\overline{B}C + AB\overline{C}$
- $Y = A\overline{B}$

F4	(A) F3	(B) F2	(C) F1	(X) F
X	0	0	0	0
X	0	0	1	1
X	0	1	0	0
X	0	1	1	0
X	1	0	0	0
X	1	0	1	0
X	1	1	0	1
x	1	1	1	0

G4	G3	(A) G2	(B) G1	(Y) G
X	X	0	0	0
X	X	0	1	0
X	X	1	0	1
X	X	1	1	0



Entwurfsfluß für FPGAs

- Wird in der Regel durch **Entwurfswerkzeuge** unterstützt
  - Beispiel: Vivado (LiveDemo an 10. Dezember 2015)
- Ist in der Regel ein **iterativer Prozess**
  - Planen
  - Implementieren (SystemVerilog)
  - Simulieren
  - Wiederhole …

Embedded Systems & Applications

Entwurfsfluß für FPGAs

TECHNISCHE UNIVERSITÄT DARMSTADT

Ist in der Regel ein iterativer Prozess

- Entwickler:
  - denkt nach
  - gibt Entwurf als Schaltplan oder **HDL-Beschreibung** ein
  - wertet **Simulationergebnisse** aus
- Wenn Simulation zufriedenstellend: **Synthetisiere** Entwurf in Netzliste
- Bilde Netzliste auf **FPGA-Konfiguration** ab (CLBs, IOBs, Verbindungsnetz)
- Lade Konfigurationsdaten (*bitstream*) **auf FPGA**
- **Teste** Schaltung nun in realer Hardware

Embedded Systems & Applications

Zusätzliche Folien für Dividierer

Beispiele und Beweise für Division

Embedded Systems & Applications

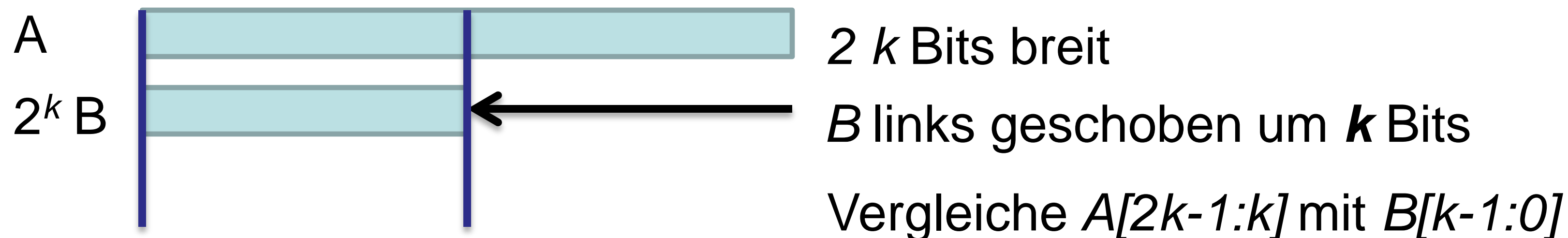
Abfangen von Überläufen

- **Maximalwert für Dividenden**

  A/B = Q + R/B

  A = QB + R

- **Maximalwert für *Q* (*k* Bit breit): $2^k - 1$**
- **Maximalwert für R (*k* Bit breit): *B* - 1**
  - A_{max} = ***QB + R*** = $[(2^k-1)\ B] + [B - 1] = 2^k\ B - 1$
  - $A \leq A_{max}$ ➔ $A < A_{max} + 1$ ➔ Wenn $A < 2^k$ B, dann **kein** Überlauf



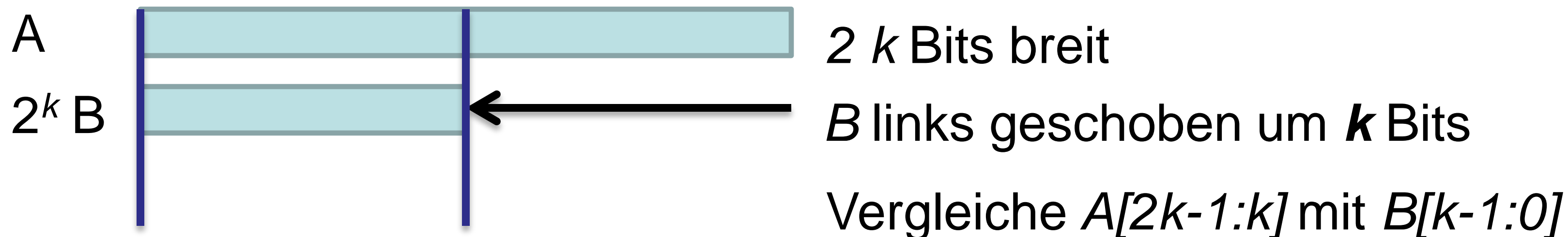
Abfangen von Überläufen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Beispiel: wir müssen herausfinden: $A < 2^k B$?**

k:	8 bit
Operanden:	A (16b) = 1482, B(8b) = 3
Ergebnis:	Q (8b) = 494 **Nein, Überlauf!**
	R (8b) = 0

Manuell: $A < 2^k B = 2^8 * 3 = 768$? **Nein, Überlauf!**



Embedded Systems & Applications

Abfangen von Überläufen

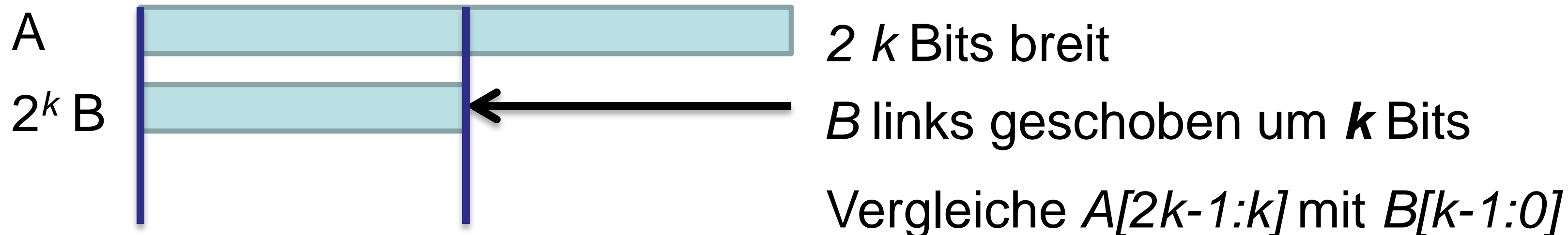
TECHNISCHE UNIVERSITÄT DARMSTADT

- **Beispiel: wir müssen herausfinden: A < 2^k B?**

k: 8 bit
Operanden: A (16b) = 1482, B(8b) = 3
Ergebnis: Q (8b) = 494 **Nein, Überlauf!**
R (8b) = 0

Manuell: $A < 2^k B = 2^8 * 3 = 768$? **Nein, Überlauf!**
beim Schieben: $B << k = B * 2^k = 3 * 2^k = 768$

```
A:     1482:  0000 0101 1100 1010
B<<8:  3 << 8:  0000 0011 0000 0000
```



Embedded Systems & Applications

Abfangen von Überläufen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Beispiel: wir müssen herausfinden: $A < 2^k B$?**

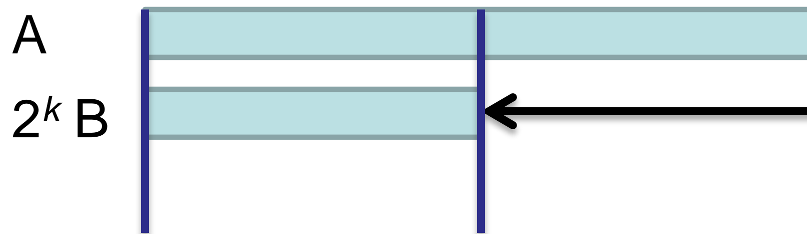
k:	8 bit
Operanden:	A (16b) = 1482, B(8b) = 3
Ergebnis:	Q (8b) = 494 **Nein, Überlauf!**
	R (8b) = 0

Manuell: $A < 2^k B = 2^8 * 3 = 768$? **Nein, Überlauf!**

beim Schieben: $B << k = B * 2^k = 3 * 2^k = 768$

```
A:      1482:   0000 0101 1100 1010
B<<8:   3 << 8: -0000 0011 0000 0000
```

wenn **$A < 2^k B$**, Differenz muss **negativ** sein



$2k$ Bits breit

B links geschoben um ***k*** Bits

Vergleiche *A[2k-1:k]* mit *B[k-1:0]*

Embedded Systems & Applications

Abfangen von Überläufen

TECHNISCHE UNIVERSITÄT DARMSTADT

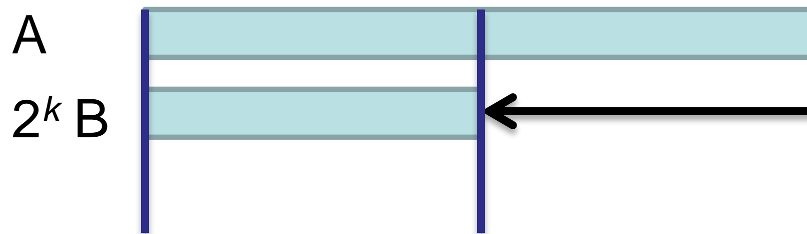
- **Beispiel 2: wir müssen herausfinden: $A < 2^k B$?**

k: 8 bit
Operanden: A (16b) = 1482, B(8b) = 5
Ergebnis: Q (8b) = 296 **Nein, Überlauf!**
R (8b) = 2

Manuell: $A < 2^k B = 2^8 * 5 = 1280$? **Nein, Überlauf!**
beim Schieben: $B << k = B * 2^k = 5 * 2^k = 1280$

```
A:      1482:  0000 0101 1100 1010
B<<8:  5 << 8: -0000 0101 0000 0000
```

wenn $A < 2^k B$, Differenz muss **negativ** sein



2 *k* Bits breit

B links geschoben um ***k*** Bits

Vergleiche *A[2k-1:k]* mit *B[k-1:0]*

Embedded Systems & Applications

Abfangen von Überläufen

TECHNISCHE UNIVERSITÄT DARMSTADT

- **Beispiel 3: wir müssen herausfinden: $A < 2^k B$?**

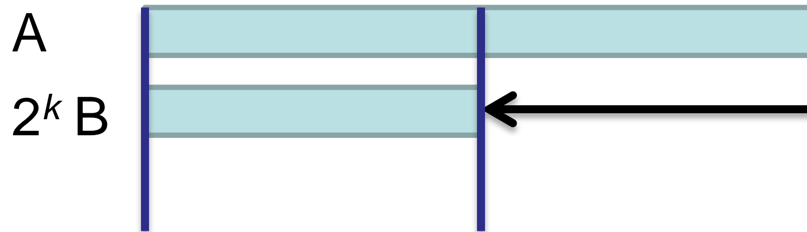
k:	8 bit
Operanden:	A (16b) = 1482, B(8b) = 6
Ergebnis:	Q (8b) = 247 **kein Überlauf**
	R (8b) = 0

Manuell: $A < 2^k B = 2^8 * 6 = 1536$? **kein Überlauf**

beim Schieben: $B << k = B * 2^k = 6 * 2^k = 1536$

```
A:     1482:   0000 0101 1100 1010
B<<8:  6 << 8: -0000 0110 0000 0000
```

$A < 2^k B$: Differenz ist **negativ**



2 *k* Bits breit

B links geschoben um ***k*** Bits

Vergleiche *A[2k-1:k]* mit *B[k-1:0]*

Embedded Systems & Applications

Idee: Quotient ziffernweise bestimmen

A/B = Q + R/B

Binär Beispiel: **0111/10 = 11 R01 (k=2)**

Schritt 1.

```
10|0111
  -10
```

erstmals, für Überlauf prüfen
wenn es passt – Überlauf!
Hier: kein Überlauf

Schritt 2.

```
10|0111
    10
```

Divisor rechts schieben

Schritt 3.

```
     1
10|0111
   -10
    01
```

Idee: Quotient ziffernweise bestimmen

TECHNISCHE UNIVERSITÄT DARMSTADT

$$A/B = Q + R/B$$

Binär Beispiel: 0111/10 = 11 R01 (k=2)

Schritt 4.

$$\begin{array}{r} 1 \\ 10 \overline{)0111} \\ -10\downarrow \\ \hline 11 \end{array}$$

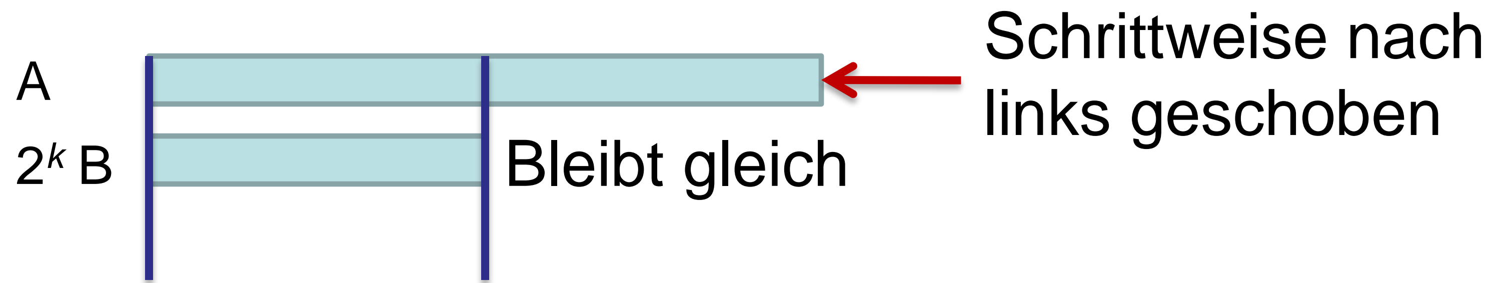
Schritt 5.

$$\begin{array}{r} 11 \\ 10 \overline{)0111} \\ -10 \\ \hline 11 \\ -10 \\ \hline 1 \end{array}$$

Embedded Systems & Applications

Optimierung

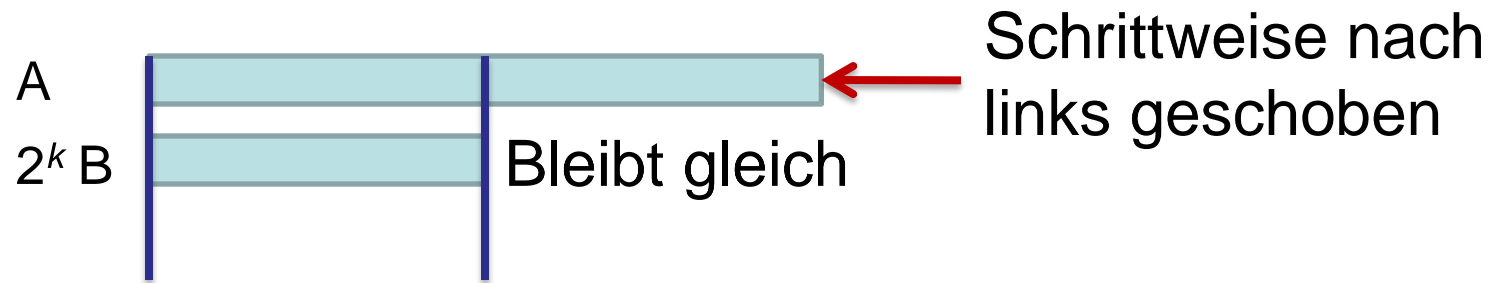
- Schiebe nicht Divisor nach rechts
- … sondern Dividend/partiellen Rest **nach links**



Optimierung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Schiebe nicht Divisor nach rechts
- … sondern Dividend/partiellen Rest **nach links**



Embedded Systems & Applications

Dividierer Algorithmus von vorher

Binär: 1101/0010 = 0110 R0001

$$A/B = Q + R/B$$

$R' = 0$

for i = N-1 to 0

$R = \{R' << 1, A_i\}$

$D = R - B$

if $D < 0$, $Q_i = 0$; $R' = R$

else $Q_i = 1$; $R' = D$

$R' = R$

```
  0001
- 0010
  1111      0 _ _ _
            3 2 1 0

  0011
- 0010
  0001      0 1 _ _
            3 2 1 0

  0010
- 0010
  0000      0 1 1 _
            3 2 1 0

  0001
- 0010
  1111      0 1 1 0  R1
            3 2 1 0
```

Dividierer Algorithmus mit unterschiedlichen Bitbreiten

A/B = Q + R

A: 8-bit breit

B,Q,R: 4-bit breit

$R' = A_{2k-1:k}$
$D = R' - B$
if $(D < 0)$ {
  for i = k-1 to 0 {
    $R = \{R' << 1, A_i\}$
    $D = R - B$
    if $D < 0$, $Q_i = 0$; $R' = R$
    else $Q_i = 1$; $R' = D$
  }
  $R' = R$
}
else Überlauf

Binär: 01011100/1010=1001 R0010

(92/10 = 9 R 2)

```
 01011100
-1010
 1011

 1011100
-1010
 0001

 001100
-1010
 1001

 01100
-1010
 1100

 1100
-1010
 0010
```


Embedded Systems & Applications

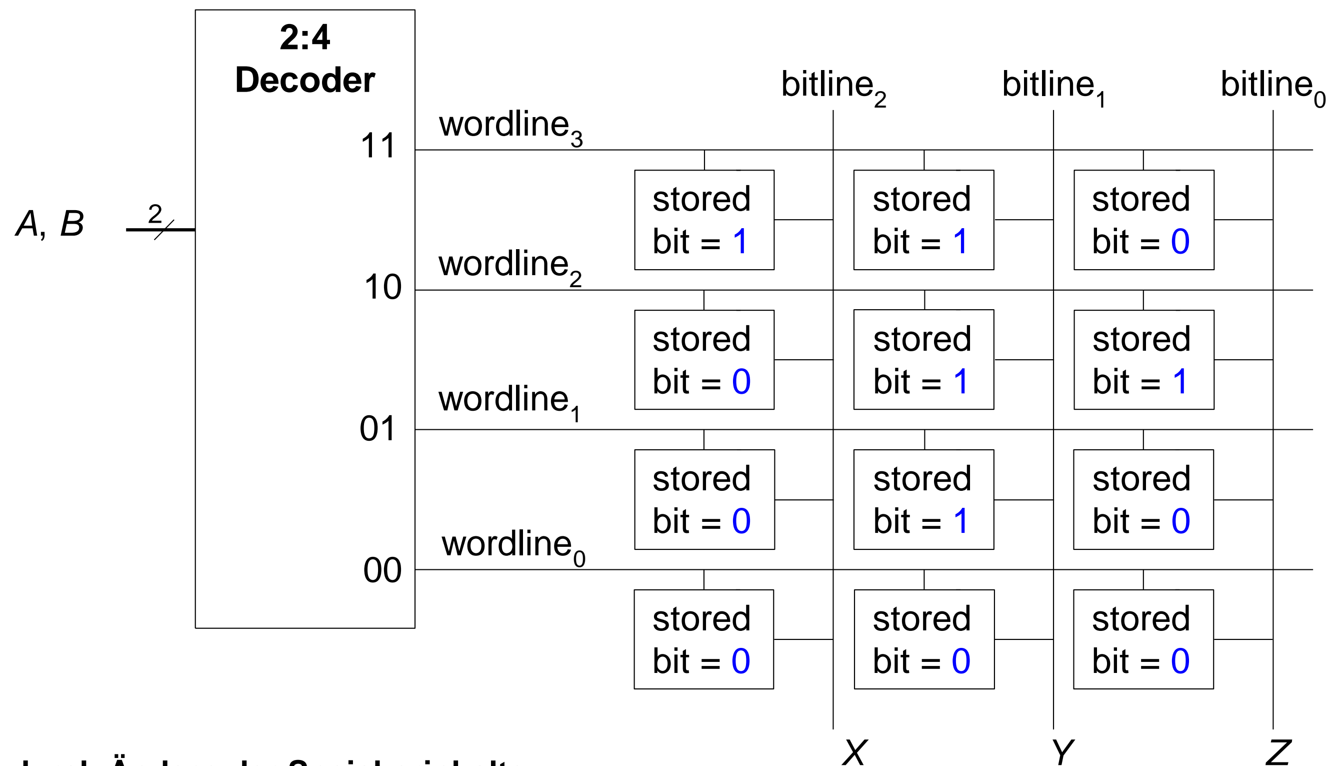
Zusätzliche Speicherfelder
Beispiele

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Embedded Systems & Applications

Zusätzliches Beispiel: Logik aus beliebigem Speicherfeld

- Implementierung der folgenden logischen Funktionen durch $2^2 \times$ 3-bit RAM:
  - $X = AB$
  - $Y = A + B$
  - $Z = A\overline{B}$



Andere Funktion nur durch Ändern der Speicherinhalte

Speicherfeld in SystemVerilog

```
// 256 x 3b Speicher mit einem Schreib/Lese-Port
module dmem(input  logic        clk, we,
            input  logic [7:0]  a,
            input  logic [2:0]  wd,
            output logic [2:0]  rd);

  logic  [2:0] RAM[255:0];

  assign rd = RAM[a];

  always @(posedge clk)
    if (we)
      RAM[a] <= wd;
endmodule
```