Technische Grundlagen der Informatik – Kapitel 5

Prof. Dr. Andreas Koch
Fachbereich Informatik
TU Darmstadt

Kapitel 5 : Themenübersicht

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

Einleitung

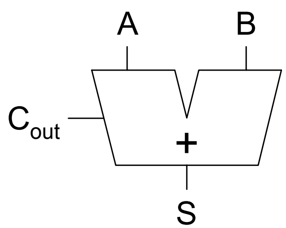
- Grundelemente digitaler Schaltungen:
  - Gatter, Multiplexer, Decoder, Register, Arithmetische Schaltungen, Zähler, Speicher, programmierbare Logikfelder

- 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
  - Kapitel 7

1-Bit Addierer

Halb-addierer

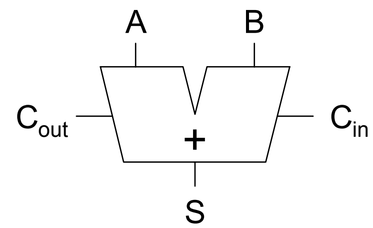


A	B	C_{out}	S
0	0		
0	1		
1	0		
1	1		

$S =$

$C_{out} =$

Voll-addierer



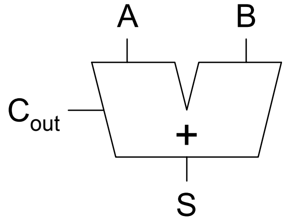
C_{in}	A	B	C_{out}	S
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

$S =$

$C_{out} =$

1-Bit Addierer

Halbaddierer

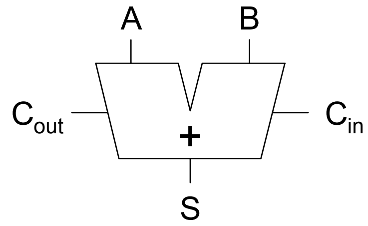


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

S =

C_{out} =

Volladdierer



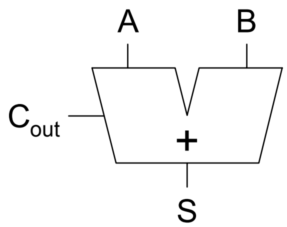
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S =

C_{out} =

1-Bit Addierer

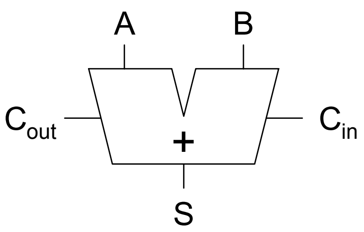
Halb-addierer



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



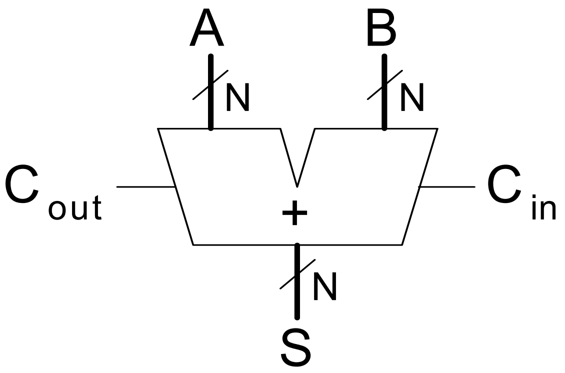
C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$
$$C_{out} = AB + AC_{in} + BC_{in}$$

Mehrbit-Addierer mit Weitergabe von Überträgen

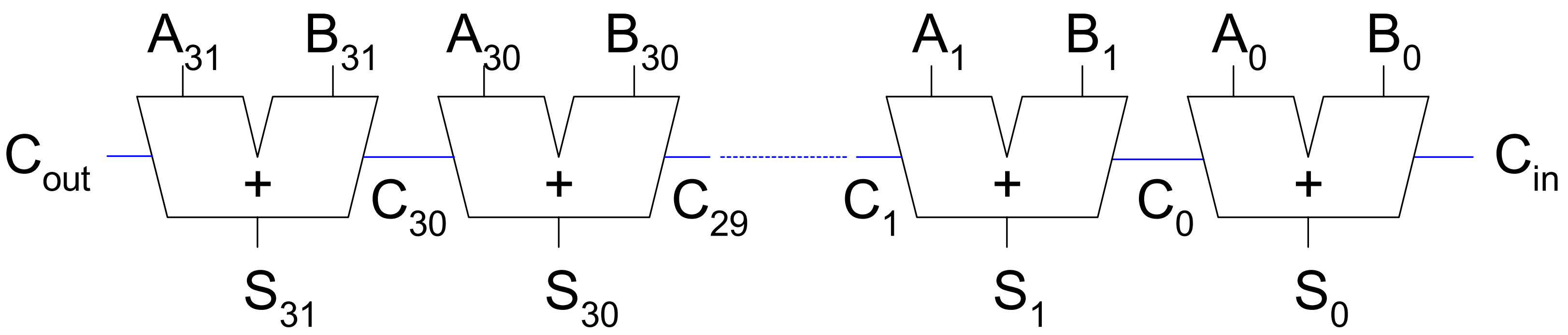
- *Carry-propagate adder (CPA)*
- Verschiedene Typen
  - Ripple-carry-Addierer (langsam)
  - Carry-Lookahead Addierer (schnell)
  - Prefix-Addierer (noch schneller)
- Carry-Lookahead und Prefix-Addierer sind schneller bei breiteren Datenworten
  - Benötigen aber auch mehr Fläche

Schaltsymbol



Ripple-Carry-Addierer

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



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

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

- Eine Spalte *i* erzeugt 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

$$P_i = A_i + B_i$$

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

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

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

Beispiel: Carry-Lookahead Addierer

- Bestimme $P_{3:0}$ und $G_{3:0}$ Signale für einen 4b Block

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

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

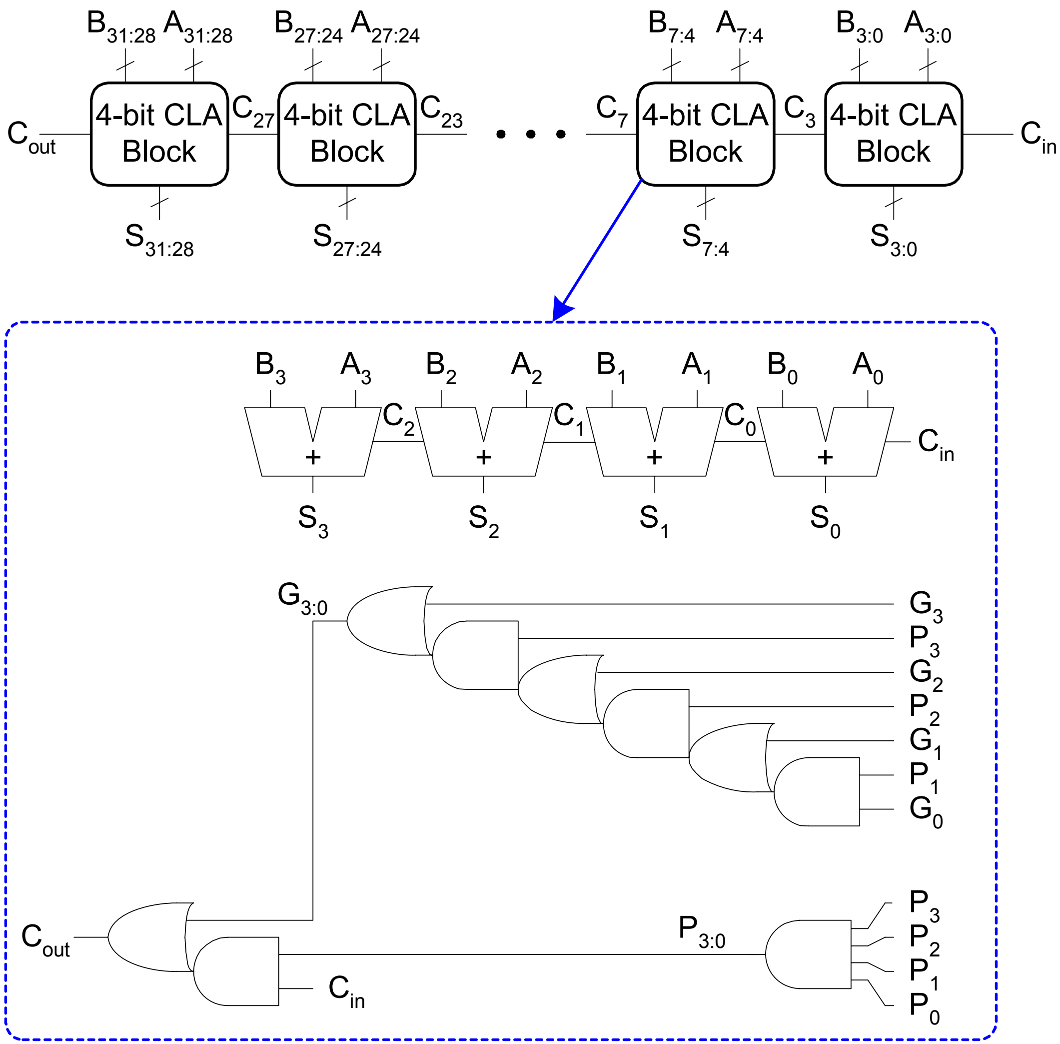
- Überlegung: Der 4b Block leitet einen Übertrag direkt weiter
  - … wenn alle Spalten den Übertrag weiterleiten

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

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

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

32-bit CLA mit 4b Blöcken

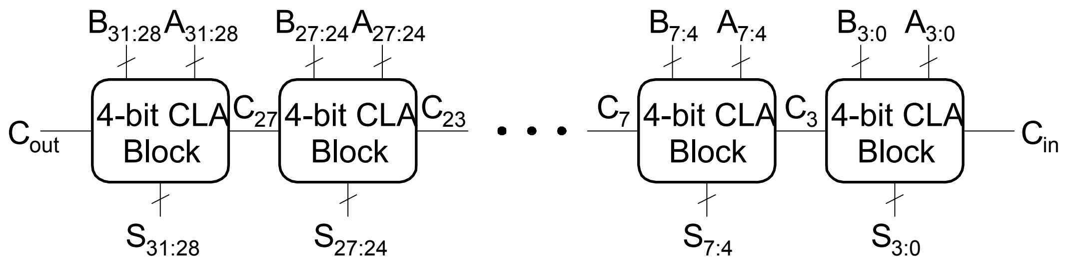


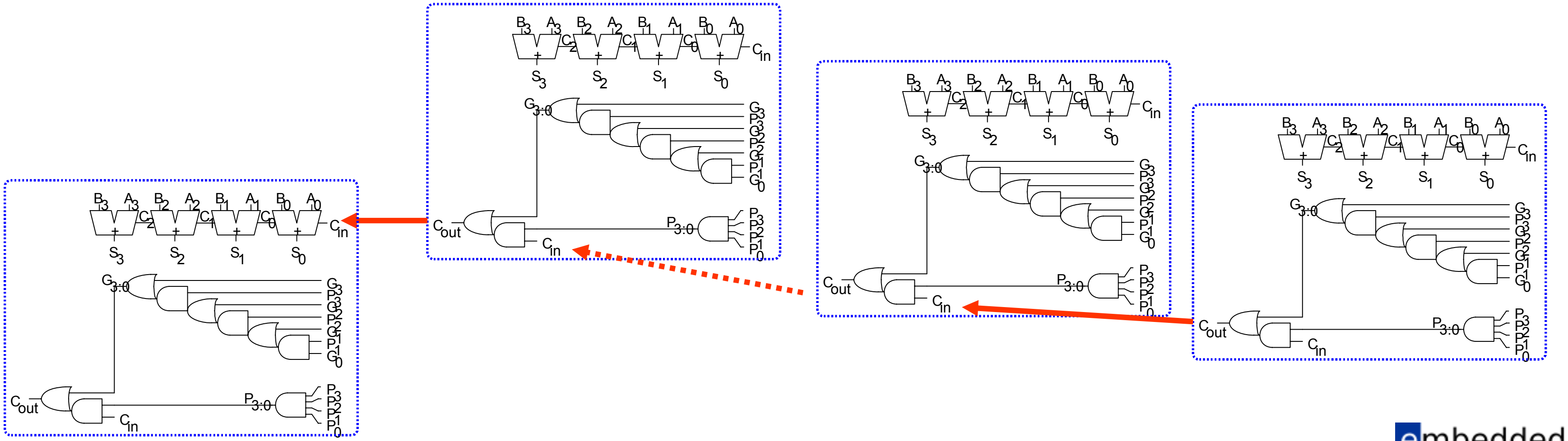
$Gi = Ai\ Bi$

$Pi = Ai + Bi$

embedded systems + applications

32-bit CLA mit 4b Blöcken





Carry-Lookahead Addierer

- 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

- 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

- 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!

Präfix-Addierer

- 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

$$G_{-1} = C_{in}, P_{-1} = 0$$

- 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*

- Interpretation: Ein Ausgangsübertrag aus Spalte *i*-1 entsteht
  - … wenn der Block *i*-1:-1 einen Übertrag generiert

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

embedded systems + applications

Präfix-Addierer

- 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*

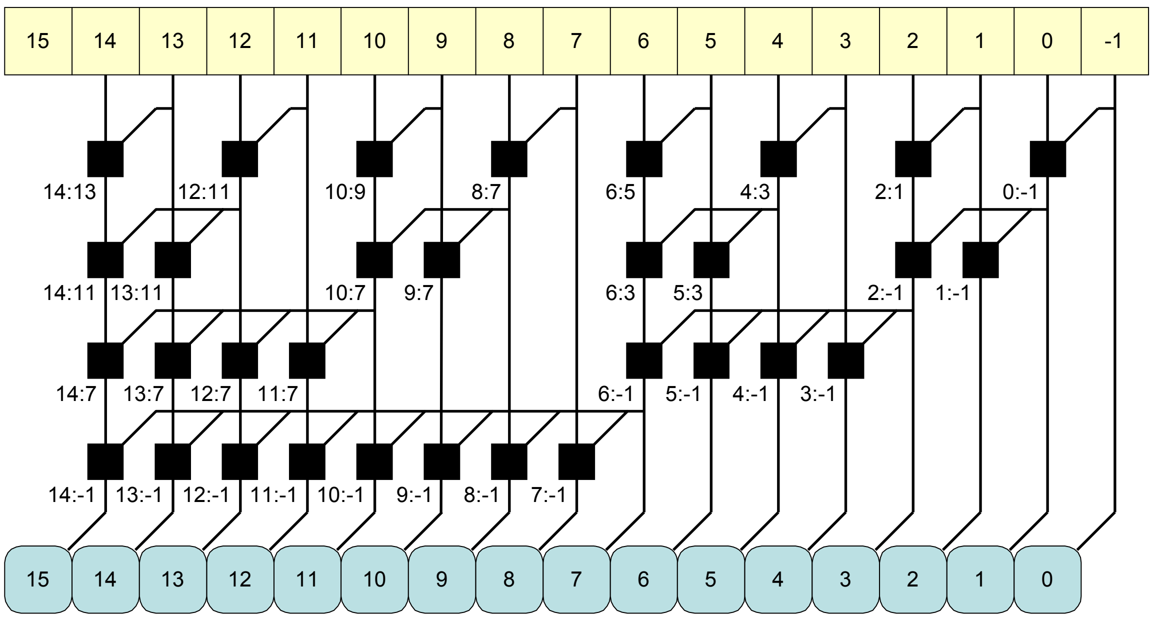
$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$
$$P_{i:j} = P_{i:k} P_{k-1:j}$$

- 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

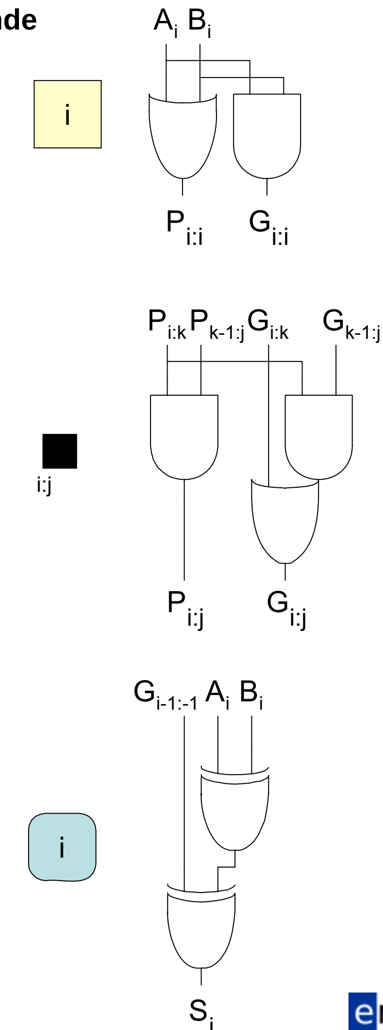
Aufbau eines Präfix-Addierers

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

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



Legende



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

Verzögerung durch Präfix-Addierer

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

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

wobei

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

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} = 300ps
  - Zwei-Eingangs Gatter $t_{\mathrm{AND}} = t_{\mathrm{OR}} = t_{\mathrm{XOR}}$ = 100ps

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

$$=$$

$$t_{CLA} = t_{pg} + t_{pg\_\mathrm{block}} + (N / k - 1)\, t_{\mathrm{AND\_OR}} + k\, t_{FA}$$

$$=$$

$$=$$

$$t_{PA} = t_{pg} + (\log_2 \mathrm{N})\, t_{\mathrm{pg\_prefix}} + t_{\mathrm{XOR}}$$

$$=$$

$$=$$

embedded systems + applications

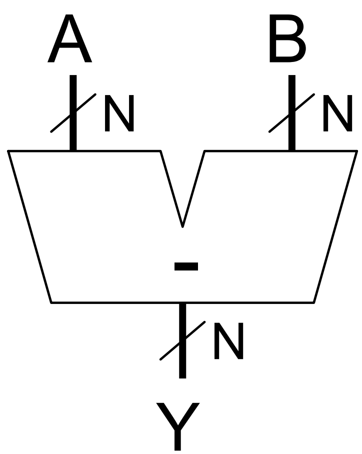
Vergleich von Addiererverzögerungen

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

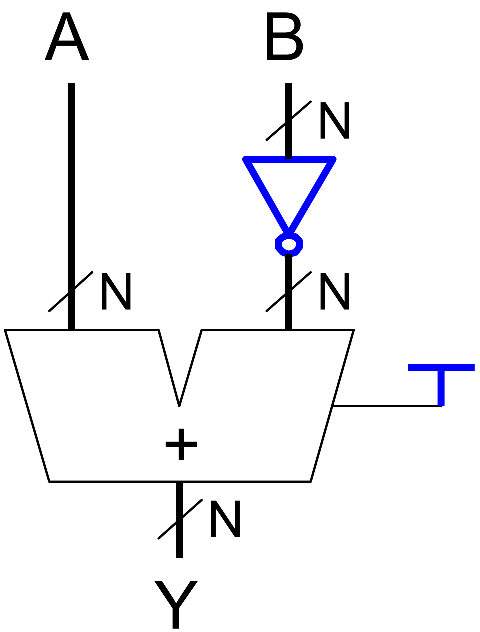
$$
\begin{aligned}
t_{\mathrm{ripple}} &= N\, t_{FA} = 32\ (300\ \text{ps}) \\
&= 9{,}6\ \text{ns} \\
t_{CLA} &= t_{pg} + t_{pg\_\mathrm{block}} + (N / k - 1)\, t_{\mathrm{AND\_OR}} + k\, t_{FA} \\
&= [100 + 600 + (7)\ 200 + 4\ (300)]\ \text{ps} \\
&= 3{,}3\ \text{ns} \\
t_{PA} &= t_{pg} + (\log_2 \mathrm{N})\ \mathrm{t}_{\mathrm{pg\_prefix}} + t_{\mathrm{XOR}} \\
&= [100 + (\log_2 32)\ 200 + 100]\ \text{ps} \\
&= 1{,}2\ \text{ns}
\end{aligned}
$$

Subtrahierer

Symbol

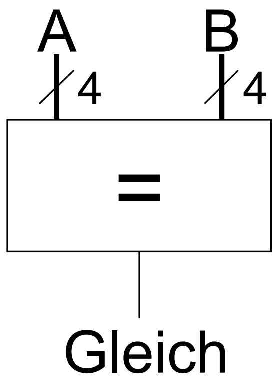


Implementierung

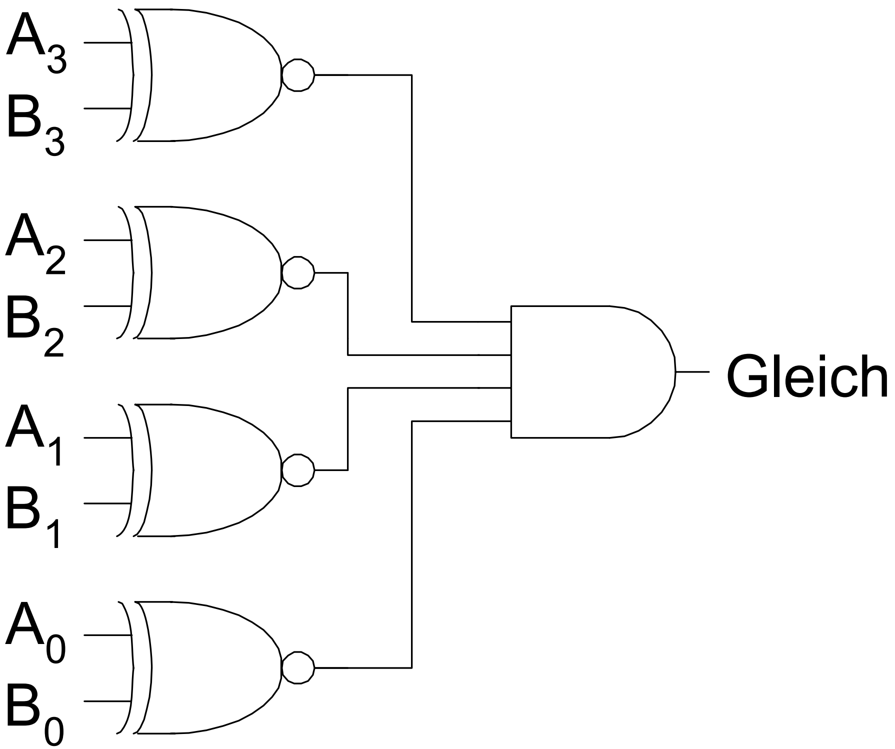


Vergleicher: Gleichheit

Symbol

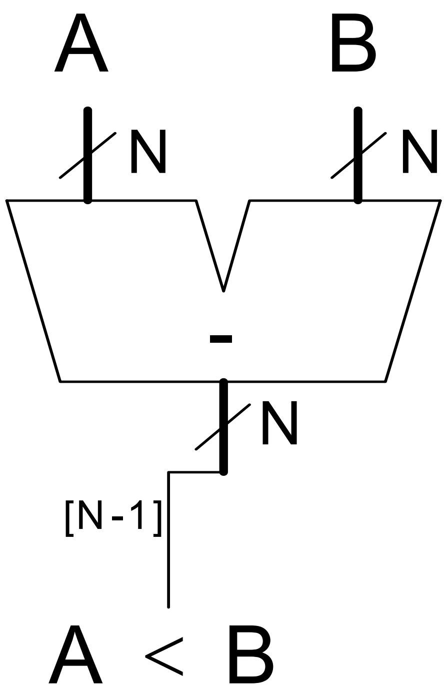


Implementierung

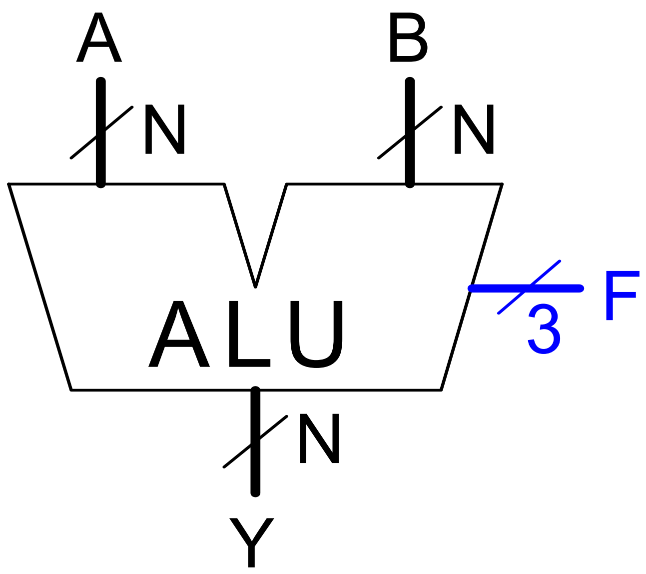


Vergleicher: Kleiner-Als

- Für vorzeichenlose Zahlen

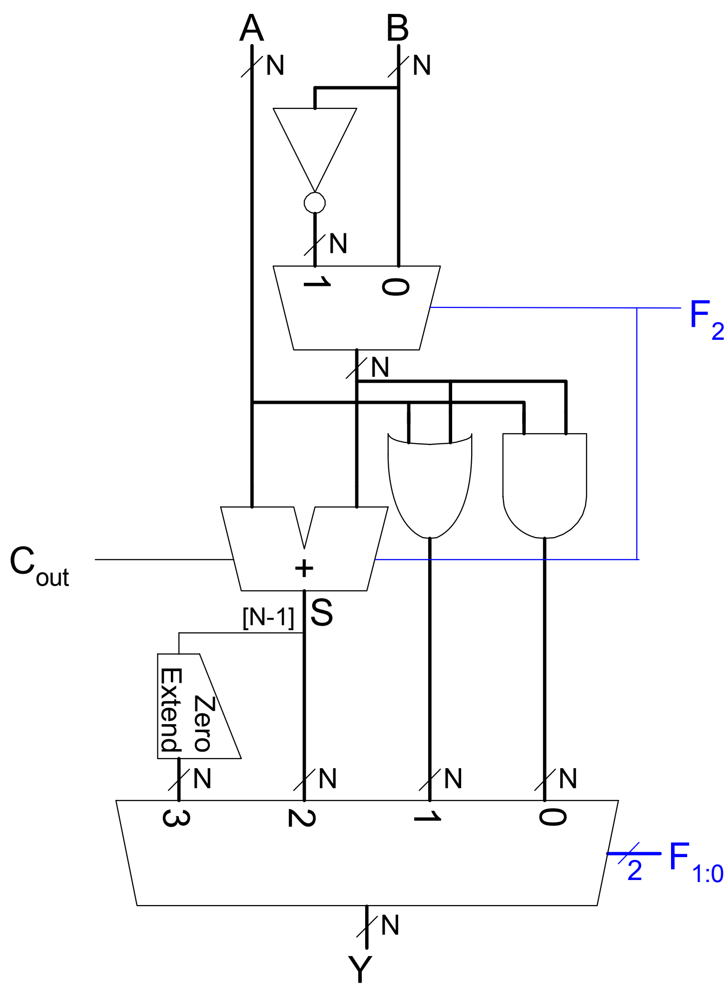


Arithmetisch-logische Einheit
(*arithmetic logic unit, ALU*)



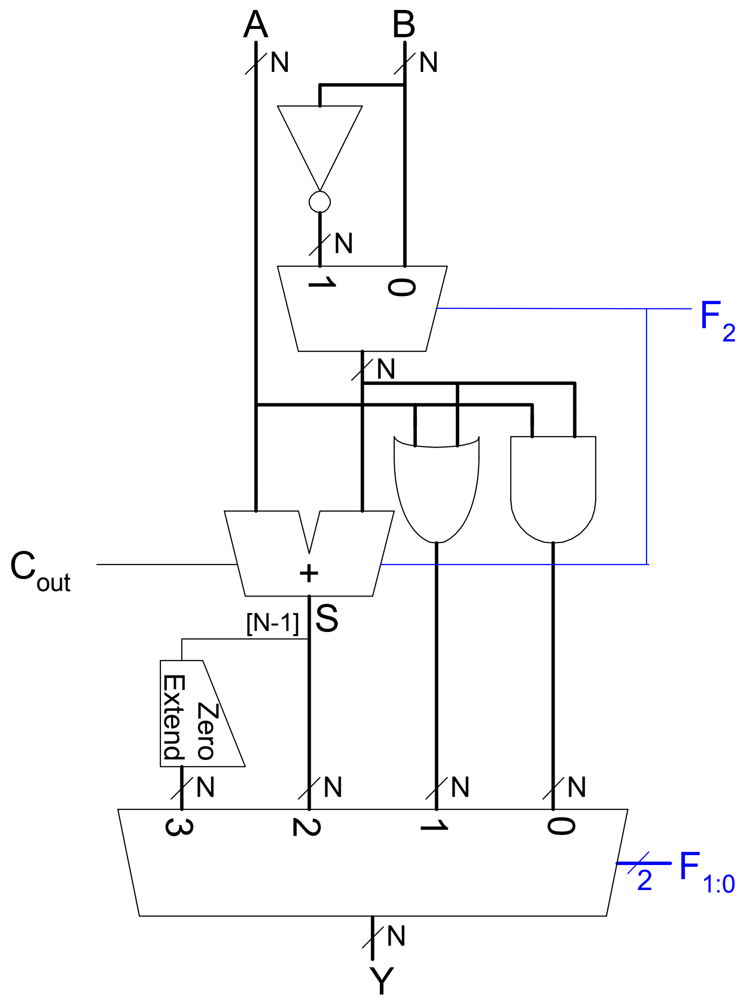
$F_{2:0}$	Funktion
000	A & B
001	A | B
010	A + B
011	Nicht verwendet
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Entwurf einer ALU



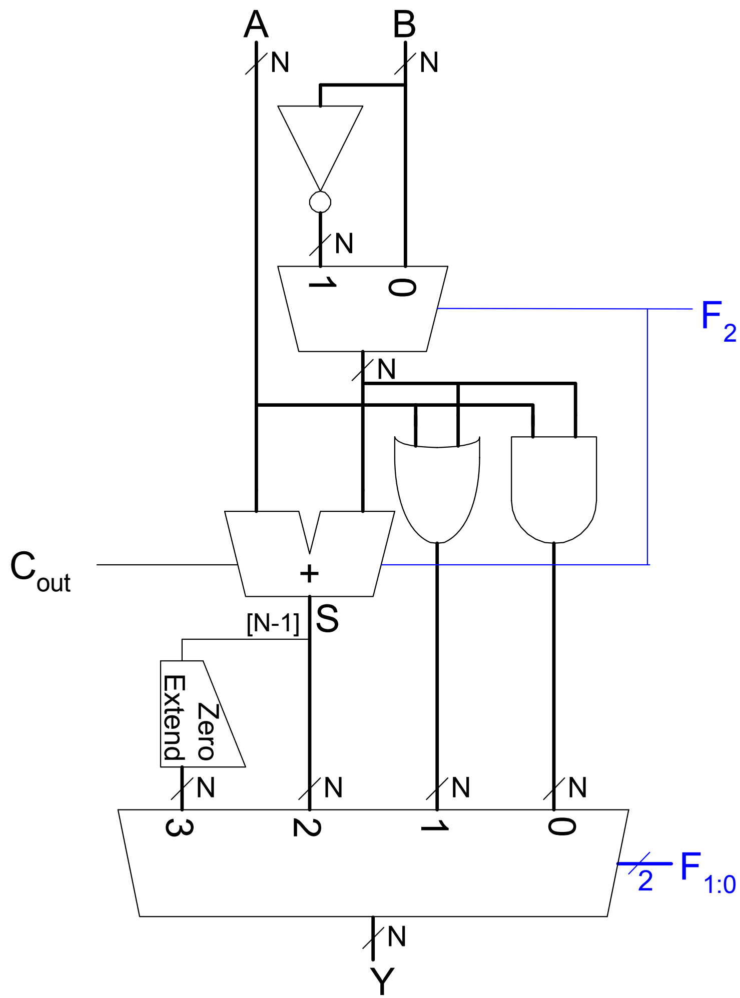
$F_{2:0}$	Funktion
000	A & B
001	A | B
010	A + B
011	Nicht verwendet
100	A & ~B
101	A | ~B
110	A - B
111	SLT

Beispiel: 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.

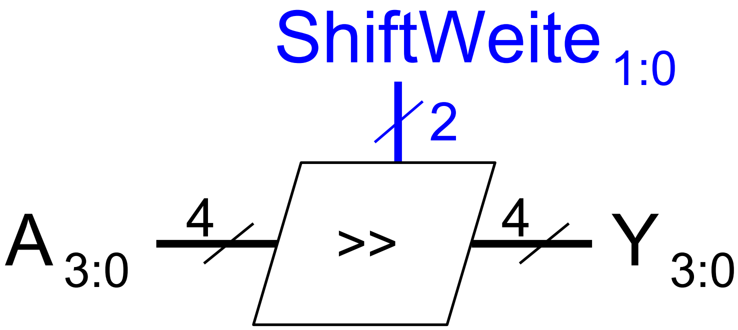
Schiebeoperationen (*shifter*)

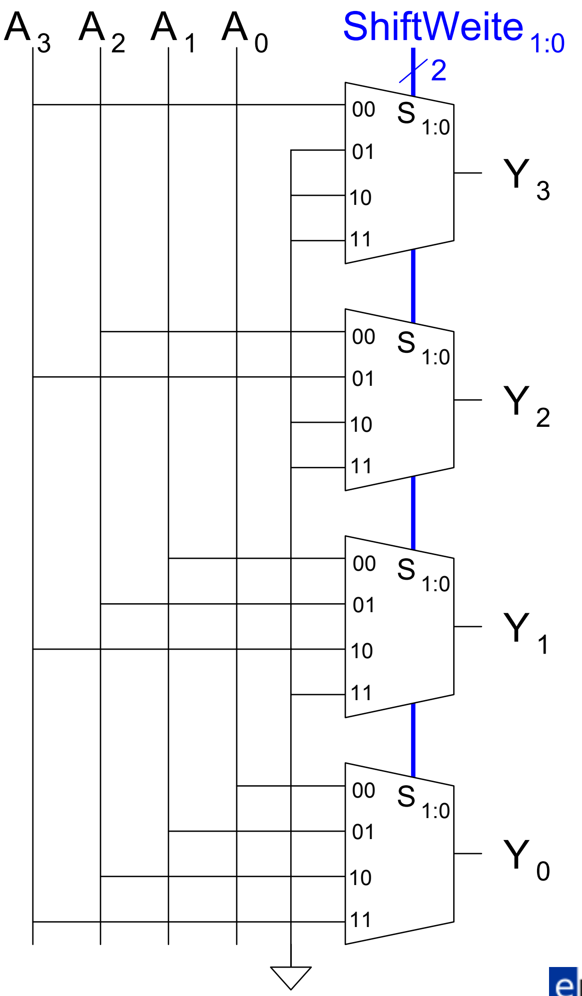
- **Logisches Schieben:** Wert wird eine Bitposition verschoben, leere Stellen mit 0 aufgefüllt
  - Beispiel: 11001 >> 2 =
  - Beispiel: 11001 << 2 =
- **Arithmetisches Schieben:** wie logisches Schieben. Verwende aber beim Rechtsschieben alten Wert des msb zum Auffüllen leerer Stellen
  - Beispiel: 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 =

Schiebeoperationen (*shifter*)

- **Logisches Schieben:** Wert wird eine Bitposition verschoben, leere Stellen mit 0 aufgefüllt
  - Beispiel: 11001 >> 2 = 00110
  - Beispiel: 11001 << 2 = 00100
- **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 = 00100
- **Rotierer:** rotiert Bits im Kreis, herausgeschobene Bits tauchen am anderen Ende wieder auf
  - Beispiel : 11001 ROR 2 = 01110
  - Beispiel : 11001 ROL 2 = 00111

Aufbau von Shiftern





Shifter als Multiplizierer und Dividierer

- Logisches Schieben um *N* Stellen nach links multipliziert den Zahlenwert mit 2^N
  - Beispiel : 00001 << 2 = 00100 ($1 \times 2^2 = 4$)
  - Beispiel : 11101 << 2 = 10100 ($-3 \times 2^2 = -12$)

- Arithmetisches Schieben um *N* Stellen nach rechts dividiert den Zahlenwert durch 2^N
  - Beispiel : 01000 >>> 2 = 00010 ($8 \div 2^2 = 2$)
  - Beispiel : 10000 >>> 2 = 11100 ($-16 \div 2^2 = -4$)

Multiplizierer

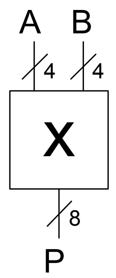
TECHNISCHE UNIVERSITÄT DARMSTADT

- Schrittweise Multiplikation in Dezimal- und Binärdarstellung:
  - Multiplizieren des Multiplikanden mit einzelner Stelle des Multiplikators
    - Berechnet ein Teilprodukt (auch partielles Produkt genannt)
  - Entsprechend der Wertigkeit der aktuellen Multiplikatorstelle nach links verschobene partielle Produkte werden aufaddiert

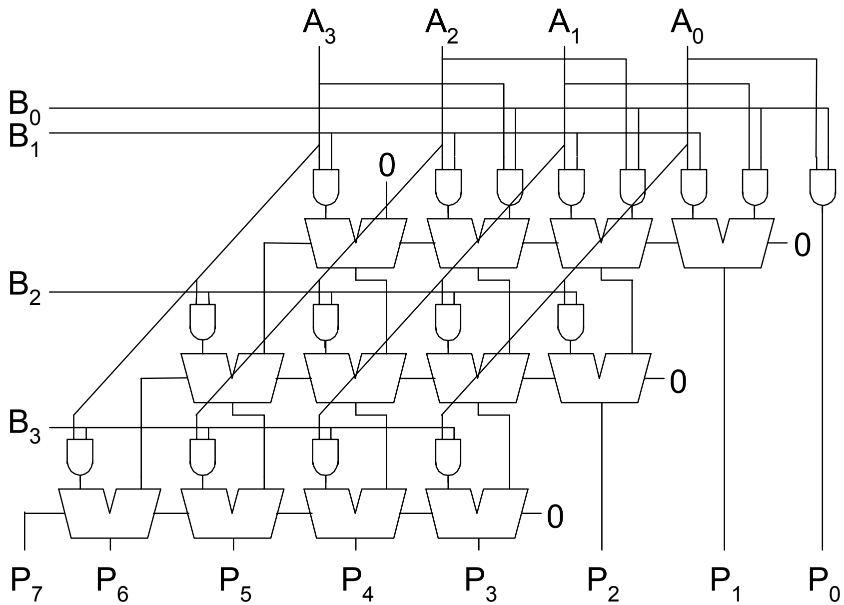
Dezimal		Binär
230	Multiplikand	0101
x 42	Multiplikator	x 0111
460	partielle Produkte	0101
+ 920		0101
9660		0101
		+ 0000
	Ergebnis	0100011
230 x 42 = 9660		5 x 7 = 35

embedded systems + applications

4 x 4 Multiplizierer



					A_3	A_2	A_1	A_0
				x	B_3	B_2	B_1	B_0
					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			
	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0



Zahlensysteme

- Bisher kennengelernt
  - Positive Zahlen
    - Vorzeichenlose Binärdarstellung
  - Negative Zahlen
    - Zweierkomplement
    - Darstellung als Vorzeichen/Betrag
- Wo bleiben Brüche?
  - Rationale Zahlen?
- Reelle Zahlen?

Zahlen mit Bruchanteilen

- Zwei gängige Darstellungen:
  - Festkomma (*fixed-point*):

    Position des Kommas bleibt konstant

    Beispiel: Dezimalsystem, 2 Vorkomma-, 3 Nachkommastellen

    2,000 99,999 0,000 -2,718   nicht: 3,1415 365,250

  - Gleitkomma (*floating-point*)

    Position des Kommas kann wandern, ist stets rechts der höchstwertigen Stelle <> 0. Angabe der Position des Kommas in Exponentenschreibweise

    Beispiel: Dezimalsystem, insgesamt 5 Stellen

    $2*10^0$ $9,9999*10^1$ $0*10^0$ $-2,718*10^0$ $3,1415*10^0$ $3,6525*10^2$ $5*10^6$

    nicht: $3,14159*10^0$

    Auch: Obergrenze für Exponenten, keine beliebig großen Zahlen darstellbar

Binäre Festkommazahlen

- Darstellung von 6,75 mit 4b für ganzen Anteil und 4b für Binärbruch:

$$01101100$$

$$0110\,,1100$$

$$2^{2} + 2^{1} + 2^{-1} + 2^{-2} = 6{,}75$$

- Binärkomma wird nicht explizit dargestellt
  - Position wird durch Format impliziert (hier: 4,4)
- Alle Leser und Schreiber von Festkommadaten müssen dasselbe Format verwenden

Binäre Festkommazahlen

- Beispiel: Stelle 7.5_{10} in 8b im 4,4-Festkommaformat dar

Binäre Festkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Beispiel: Stelle 7.5_{10} in 8b im 4,4-Festkommaformat dar

01111000

embedded systems + applications

Vorzeichenbehaftete Festkommazahlen

- Wie bei ganzen Zahlen: Zwei Darstellungen möglich
  - Vorzeichen/Betrag
  - Zweierkomplement

- Stelle -7.5_{10} in 8b als 4,4-Festkommazahl dar

  - Vorzeichen/Betrag:

  - Zweierkomplement:

Vorzeichenbehaftete Festkommazahlen

- Wie bei ganzen Zahlen: Zwei Darstellungen möglich
  - Vorzeichen/Betrag
  - Zweierkomplement

- Stelle -7.5_{10} in 8b als 4,4-Festkommazahl dar

  - Vorzeichen/Betrag:

    11111000

  - Zweierkomplement:

Vorzeichenbehaftete Festkommazahlen

- Wie bei ganzen Zahlen: Zwei Darstellungen möglich
  - Vorzeichen/Betrag
  - Zweierkomplement

- Stelle -7.5_{10} in 8b als 4,4-Festkommazahl dar
  - Vorzeichen/Betrag:

    11111000

  - Zweierkomplement:

1. +7.5:	01111000
2. Invertieren:	10000111
3. Addiere 1 zu lsb:	+ 1
	10001000

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

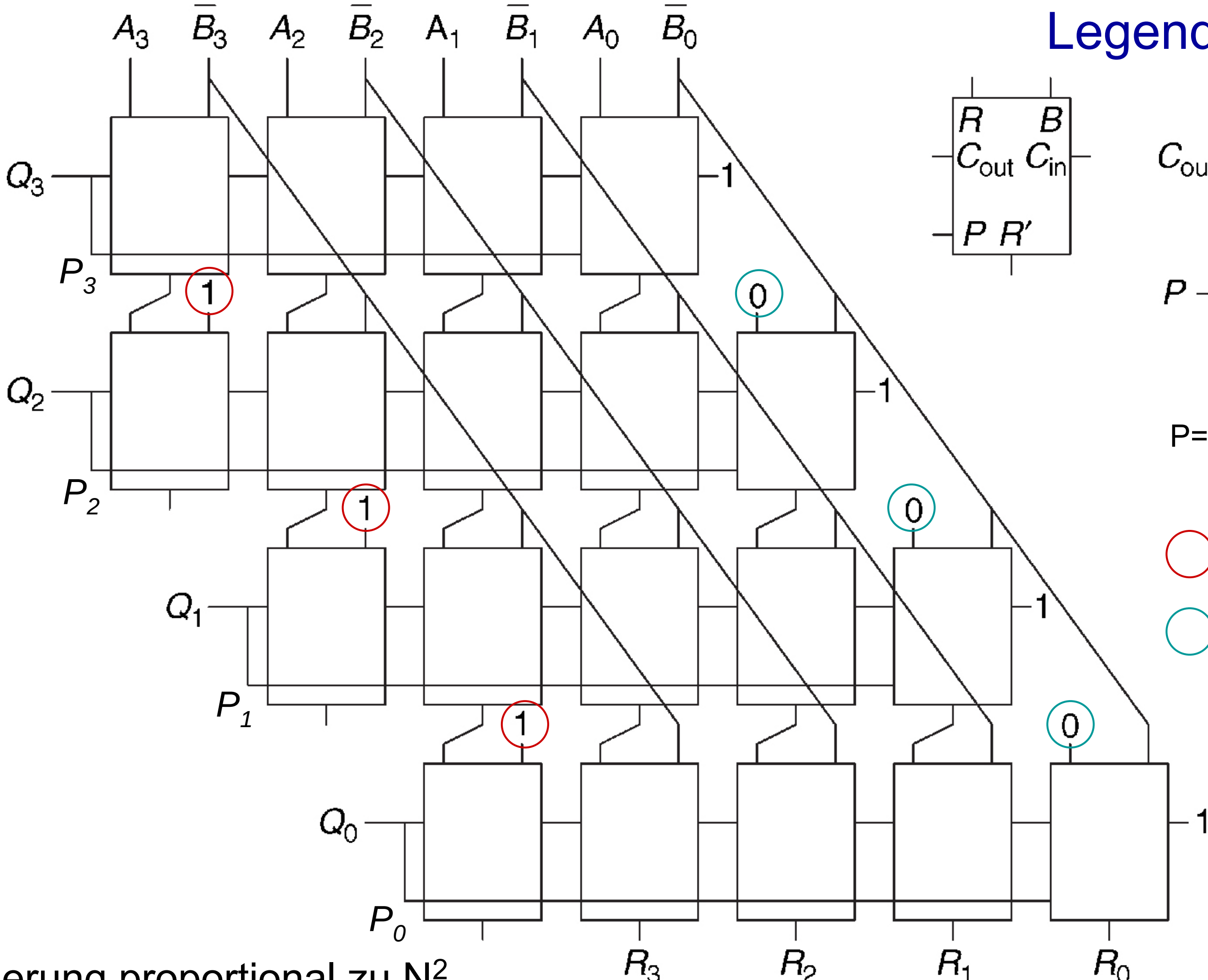
Ein Algorithmus für vorzeichenlose Division

- $Q = A / B$: Quotient
- $R = A \text{ mod } B$: Rest
- D : aktuelle Differenz

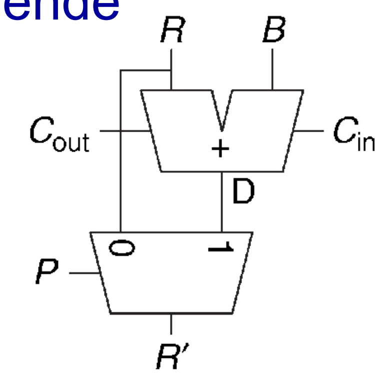
```
R = A                  // partieller Rest
for i = N-1 to 0       // über alle Stellen der Binärzahl
    D = R - B
    if D < 0 then Q_i = 0, R' = R      // R < B
    else          Q_i = 1, R' = D      // R ≥ B
    if i <> 0 then R = 2 R'
```

Vorsicht: Dieser Algorithmus funktioniert nur für Zahlenbereich $[2^{N}-1,2^{N-1}]$
liefert Ergebnisse als 1,N-1 Festkommazahl

4 x 4 Dividierer



Legende



P=1 wenn Differenz negativ ist

Sign-extension von B

Right-shift von R

Verzögerung proportional zu N^2

Binäre Gleitkommazahlen

- Binärkomma liegt immer genau rechts von höchstwertiger 1
- Ähnlich zur wissenschaftlichen Darstellung von Dezimalbrüchen
- Beispiel: 4.387.263 in wissenschaftlicher Darstellung

$$\mathbf{4{,}387263 \times 10^{6}}$$

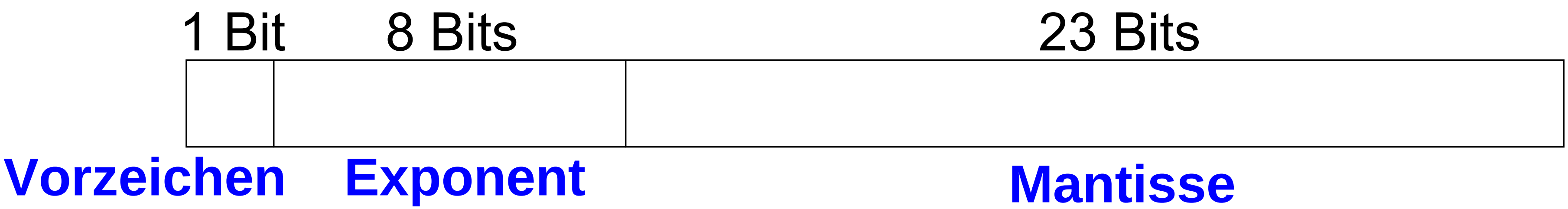
- Allgemeine Schreibweise:

$$\pm \mathbf{M} \times \mathbf{B}^{\mathbf{E}}$$

wobei

  - **M** = Mantisse
  - **B** = Basis
  - **E** = Exponent
  - Im Beispiel: M = 4,387263 , B = 10, and E = 6

Binäre Gleitkommazahlen



- **Beispiel:** Stelle den Wert 228_{10} als 32b-Gleitkommazahl dar
  - Im folgenden drei Versionen, nur die letzte davon ist eine Standarddarstellung!
    - IEEE 754, *single precision format*

Binäre Gleitkommadarstellung: 1. Versuch

- Wandele Dezimalzahl in Binärdarstellung um:
  - $228_{10} = 11100100_2 = 1{,}11001 \times 2^7$
- Trage nun Daten in die Felder des 32b Wortes ein:
  - Vorzeichenbit ist positiv (0)
  - Die 8b des Exponenten stellen den Wert 7 dar
  - Die verbliebenen 23 Bit stellen die Mantisse dar

1 Bit	8 Bits	23 Bits
0	00000111	11 1001 0000 0000 0000 0000
Vorzeichen	**Exponent**	**Mantisse**

Binäre Gleitkommadarstellung: 2. Versuch

- Beobachtung: Das erste Bit der Mantisse ist so immer 1
  - $228_{10} = 11100100_2 = 1{,}\mathbf{11001} \times 2^7$
- Man kann sich das explizit Abspeichern der führenden 1 sparen
  - Die führende eine wird implizit immer als präsent angenommen
- Stattdessen: Speichere nur den Bruchanteil (die “Nachkommastellen”) explizit ab

1 Bit	8 Bits	23 Bits
0	00000111	110 0100 0000 0000 0000 0000
Vorzeichen	**Exponent**	**Bruchanteil**

Binäre Gleitkommadarstellung: 3. Versuch

- Exponent kann auch negativ sein
  - Idee: Zweierkomplement. Wäre möglich, hat aber praktische Nachteile
  - Besser: Exponent relativ zu konstantem Grundwert (Exzess, Biaswert) angeben
- Hier: Biaswert = 127 (01111111_2)
  - Exponent mit Bias = Biaswert + Exponent
  - Exponent 7 wird also gespeichert als:

$$127 + 7 = 134 = 0x10000110_2$$

- Damit **IEEE 754 32-bit Gleitkommadarstellung** von 228_{10}

1 Bit	8 Bits	23 Bits
0	10000110	110 0100 0000 0000 0000 0000
Vorz.	**Exponent mit Bias**	**Bruchanteil**

Beispiel IEEE 754 Gleitkommadarstellung

- Stelle -58.25_{10} gemäß dem IEEE 754 32-bit Gleitkommastandard dar

Beispiel IEEE 754 Gleitkommadarstellung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Stelle -58.25_{10} gemäß dem IEEE 754 32-bit Gleitkommastandard dar
- 1. Wandele in Binärdarstellung um:
  - $58{,}25_{10}$ =
- 2. Trage Felder des 32b Gleitkommawortes ein:
  - Vorzeichen:
  - 8 Bits für Exponent:
  - 23 Bits für Bruchanteil:

1 Bit	8 Bits	23 Bits
Vorz.	**Exponent**	**Bruchanteil**

- In Hexadezimalschreibweise:

Beispiel IEEE 754 Gleitkommadarstellung

- Stelle -58.25_{10} gemäß dem IEEE 754 32-bit Gleitkommastandard dar
- 1. Wandele in Binärdarstellung um:
  - $58{,}25_{10} = 111010{,}01_2 = 1.1101001 \times 2^5$
- 2. Trage Felder des 32b Gleitkommawortes ein:
  - Vorzeichen: 1 (negativ)
  - 8 Bits für Exponent: $(127 + 5) = 132 = 10000100_2$
  - 23 Bits für Bruchanteil: 110 1001 0000 0000 0000 0000

1 Bit	8 Bits	23 Bits
1	100 0010 0	110 1001 0000 0000 0000 0000
Vorz.	**Exponent**	**Bruchanteil**

- In Hexadezimalschreibweise: 0xC2690000

IEEE 754 Gleitkommadarstellung: Sonderfälle

- Nicht alle benötigten Werte nach dem Schema darstellbar
  - Beispiel: 0, hat keine führende 1

Wert	Vorz.	Exponent	Bruchanteil
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	Ein Wert <> 0

NaN steht für “Not a Number” und stellt häufig Rechenfehler dar

Beispiele: $\sqrt{-1}$ oder log(-5).

Genauigkeit der Gleitkommadarstellungen

- Einfache Genauigkeit (*single-precision*):
  - 32-bit Darstellung
  - 1 Vorzeichenbit, 8 Exponentenbits, 23 Bits für Bruchanteil
  - Exponentenbias = 127
- Doppelte Genauigkeit (*double-precision*):
  - 64-bit Darstellung
  - 1 Vorzeichenbit, 11 Exponentenbits, 52 Bits für Bruchanteil
  - Exponentenbias = 1023

Rundungsmodi für Gleitkommazahlen

- Overflow: Betrag der Zahl ist zu groß, um korrekt dargestellt zu werden
- Underflow: Zahl ist zu nah bei 0, um korrekt dargestellt zu werden

- Rundungsmodi:
  - Abrunden zu minus Unendlich
  - Aufrunden zu plus Unendlich
  - Hin zu Null
  - Hin zu nächster darstellbarer Zahl

- **Beispiel:** Runde 1,100101 ($1{,}578125_{10}$) auf 3 Bits Bruchanteil
  - Ab: 1,100
  - Auf: 1,101
  - Zu Null: 1,100
  - Zu nächster: 1,101 (1,625 liegt näher an 1,578125 als an 1,5)

Addition von Gleitkommazahlen mit gleichem Vorzeichen

1. Exponenten- und Bruchanteilfelder aus Gleitkommawort extrahieren
2. Bruchanteil um führende 1 erweitern, um Mantisse zu bilden
3. Vergleiche Exponenten
4. Schiebe Mantisse von Zahl mit kleinerem Exponenten nach rechts (bis Exponenten gleich sind)
5. Addiere Mantissen
6. Normalisiere Mantisse und passe Exponent an, falls nötig
7. Runde Ergebnis entsprechend dem gewählten Rundungsmodus
8. Baue Gleitkommawort aus Exponenten und Bruchanteil des Ergebnisses

Beispiel: Addition von Gleitkommazahlen

Addiere die beiden Gleitkommazahlen

0x3FC00000

0x40500000

Beispiel: Addition von Gleitkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Extrahiere Exponenten und Bruchanteile aus 32b Worten

1 Bit	8 Bits	23 Bits
0	01111111	100 0000 0000 0000 0000 0000
Vorz.	**Exponent**	**Bruchanteil**

1 Bit	8 Bits	23 Bits
0	10000000	101 0000 0000 0000 0000 0000
Vorz.	**Exponent**	**Bruchanteil**
S	**E**	**F**

1. Zahl (N1): S1 = 0, E1 = 127, F1 = ,1

2. Zahl (N2): S2 = 0, E2 = 128, F2 = ,101

2. Erweitere Bruchanteile um führende 1, um Mantissen zu bilden

M1: 1,1

M2: 1,101

Beispiel: Addition von Gleitkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

3. Vergleiche Exponenten

   128 – 127 = 1, N1 muss also um ein Bit geschoben werden

4. Mantisse von Zahl mit kleinerem Exponenten um entsprechend nach rechts schieben

   schiebe M1: 1,1 >> 1 = 0,11 $(\times 2^1)$

5. Mantissen addieren (haben jetzt den gleichen Exponenten)

$$
\begin{array}{rr}
 & 0{,}11 \times 2^1 \\
+ & 1{,}101 \times 2^1 \\
\hline
 & 10{,}011 \times 2^1
\end{array}
$$

embedded systems + applications

Beispiel: Addition von Gleitkommazahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

6. Normalisiere Mantisse und passe Exponenten an, falls nötig
   $10{,}011 \times 2^1 = 1{,}0011 \times 2^2$
7. Runde Ergebnis entsprechend Rundungsmodus
   Hier nicht nötig (pass in 23b)
8. Baue neues Gleitkommawort für Ergebnis aus Exponent und Mantisse
   $S = 0$, $E = 2 + 127 = 129 = 10000001_2$, $F = 001100..$

1 Bit	8 Bits	23 Bits
0	10000001	001 1000 0000 0000 0000 0000
Vorz.	**Exponent**	**Bruchanteil**

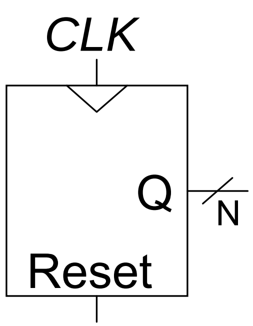
In Hexadezimalschreibweise: 0x40980000

embedded systems + applications

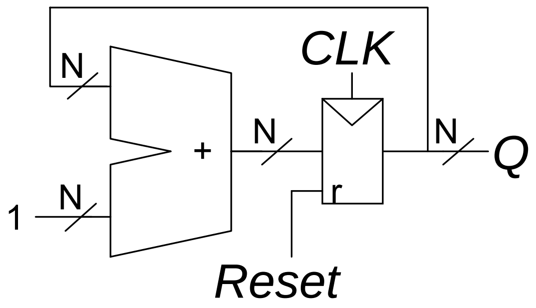
Zähler

- 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



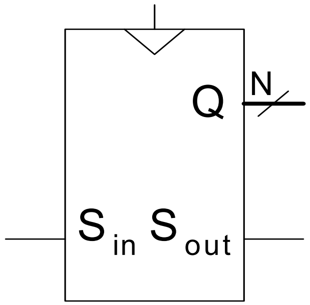
Aufbau



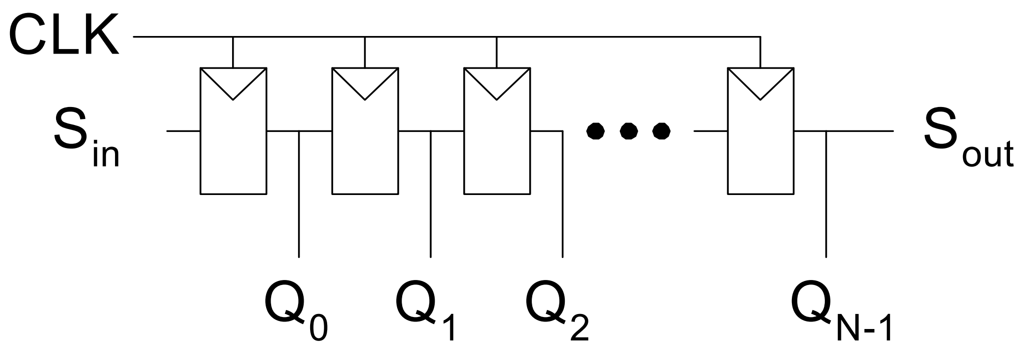
Schieberegister

- 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:

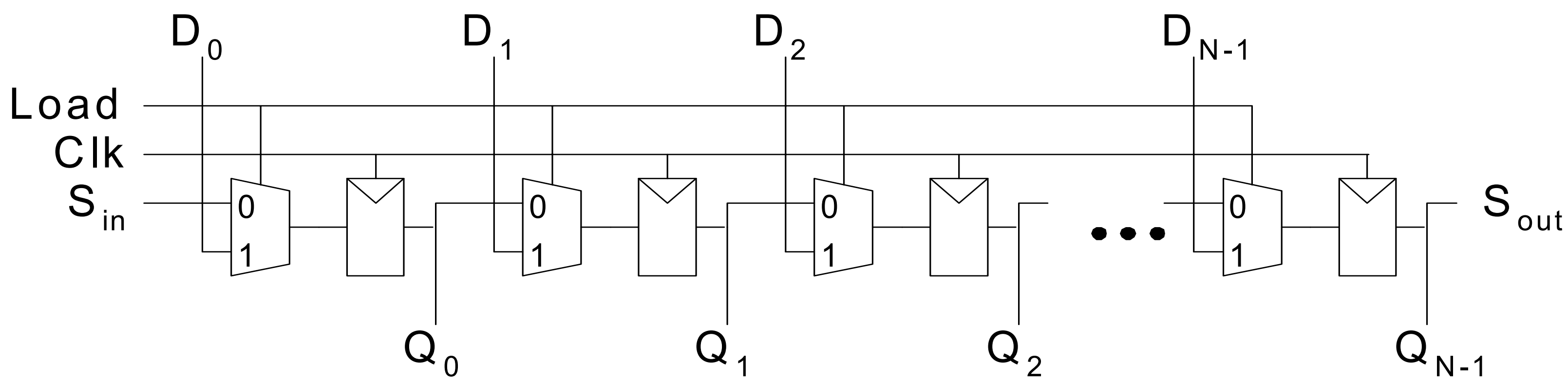


Aufbau:



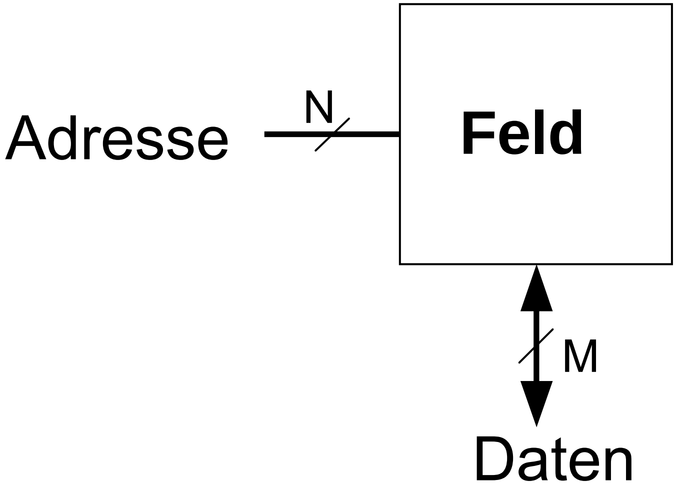
Schieberegister mit parallelem Laden

- 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})



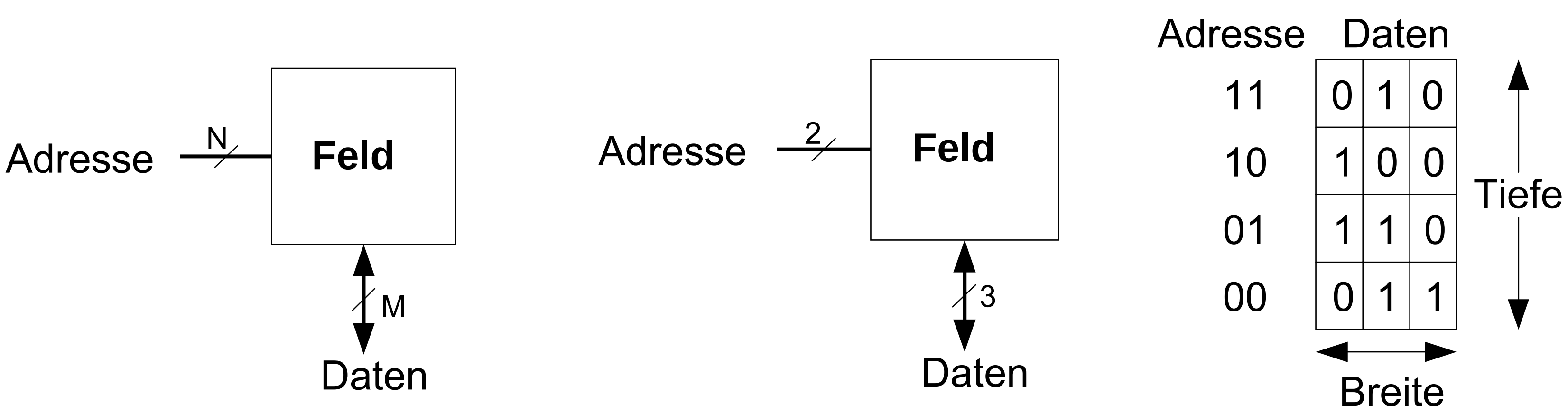
Speicherfelder

- Können effizient größere Datenmengen speichern
- Drei weitverbreitete Typen:
  - Dynamischer Speicher mit wahlfreiem Zugriff
    - (*Dynamic random access memory*, DRAM)
  - Statischer Speicher mit wahlfreiem Zugriff
    - (*Static random access memory*, SRAM)
  - Nur-Lesespeicher (*Read only memory*, ROM)
- An jede *N*-bit Adresse kann ein *M*-bit breites Datum geschrieben werden



Speicherfelder

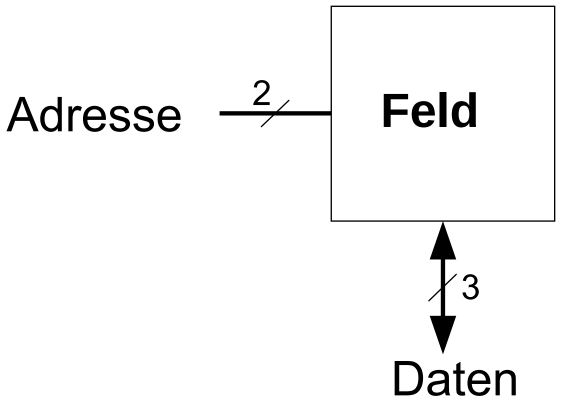
- Zweidimensionales Feld von Bit-Zellen
- Jede Bit-Zelle speichert ein Bit
- Feld mit *N* Adressbits und *M* Datenbits:
  - 2^N Zeilen und *M* Spalten
  - **Tiefe:** Anzahl von Zeilen (Anzahl von Worten)
  - **Breite:** Anzahl von Spalten (Bitbreite eines Wortes)
  - **Feldgröße:** Tiefe × Breite = $2^N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



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:

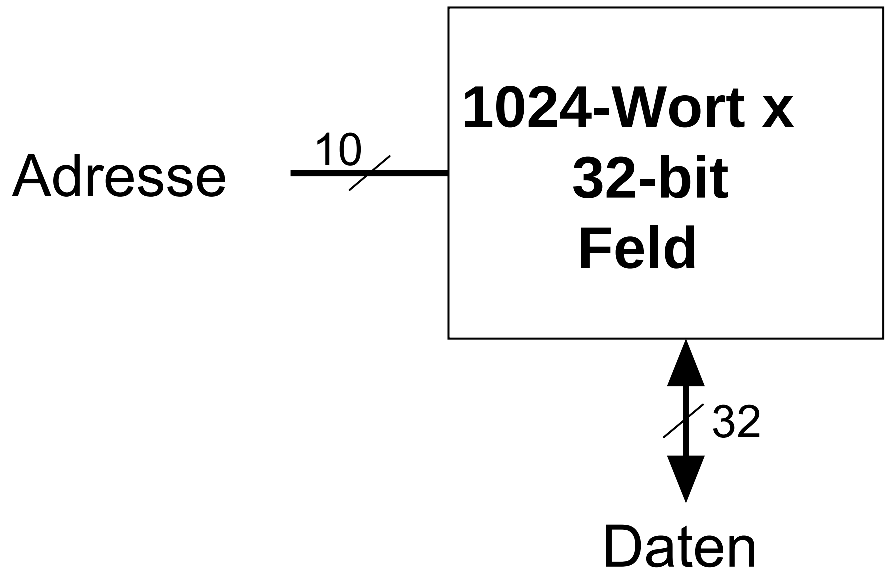


Adresse	Daten		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

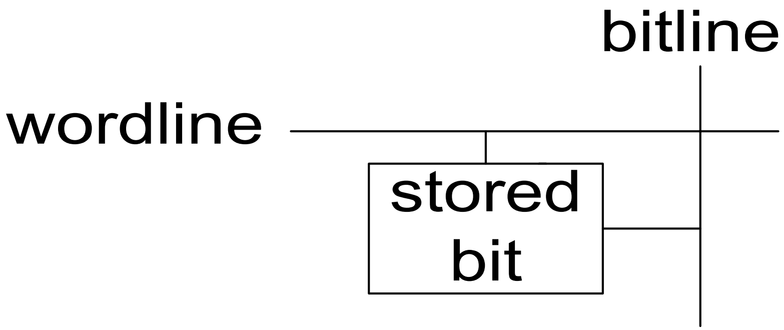
Tiefe

Breite

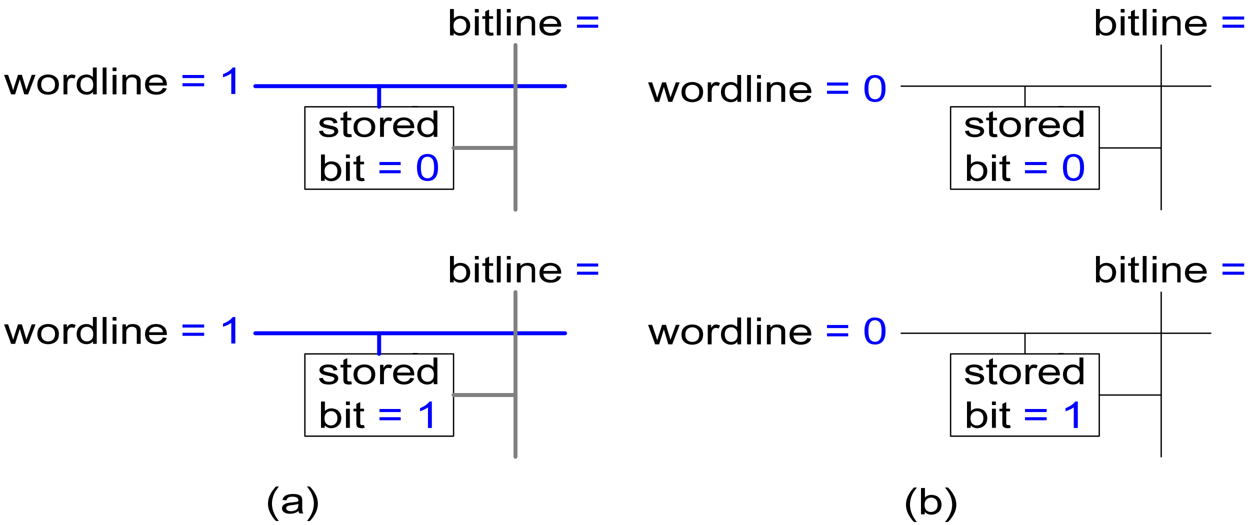
Speicherfelder



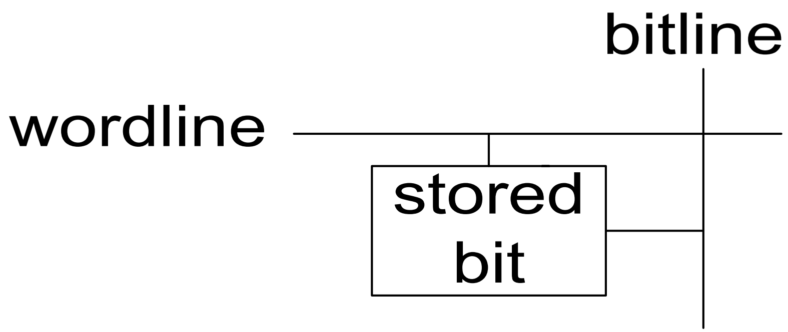
Bit-Zellen für Speicherfelder



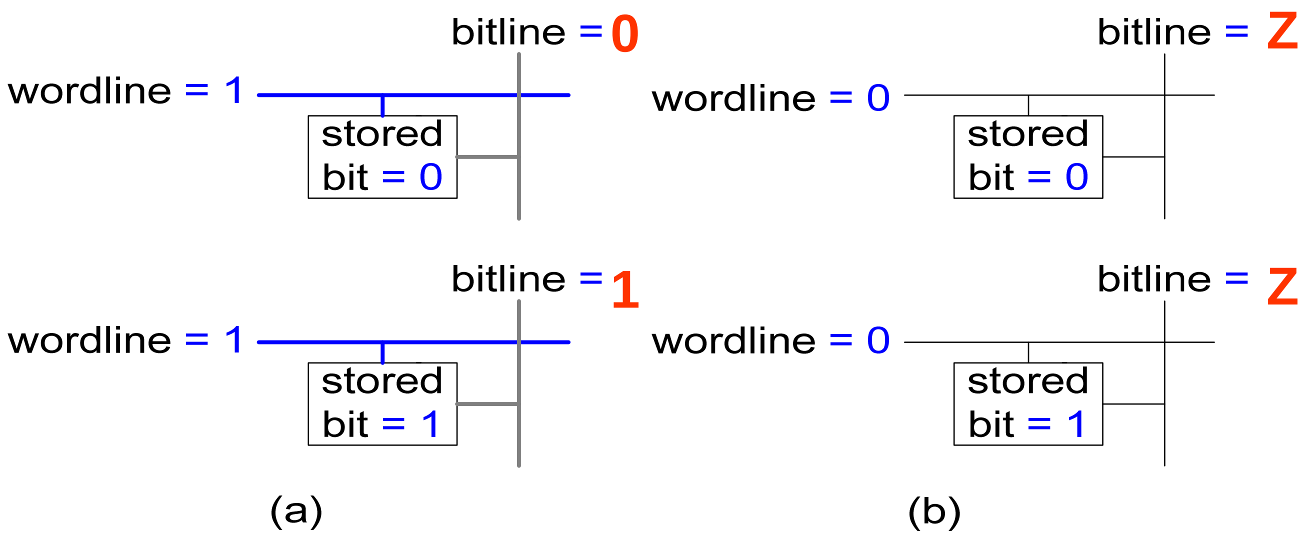
Beispiel:



Aufbau von Speicherfeldern aus Bit-Zellen



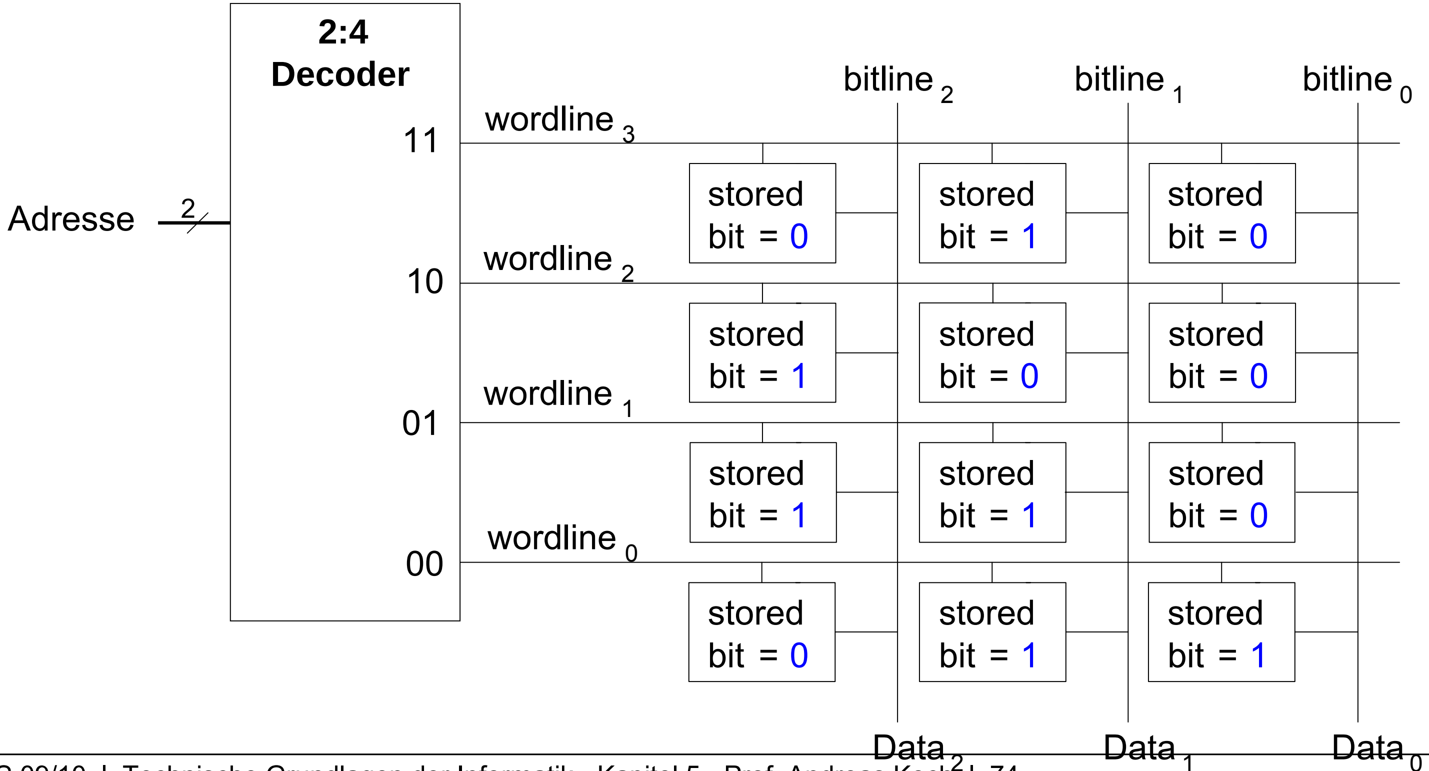
Beispiel:



embedded systems + applications

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



Arten von Speicher: Historische Sicht

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

RAM: Random-Access Memory

- **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, …

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

Arten von RAM

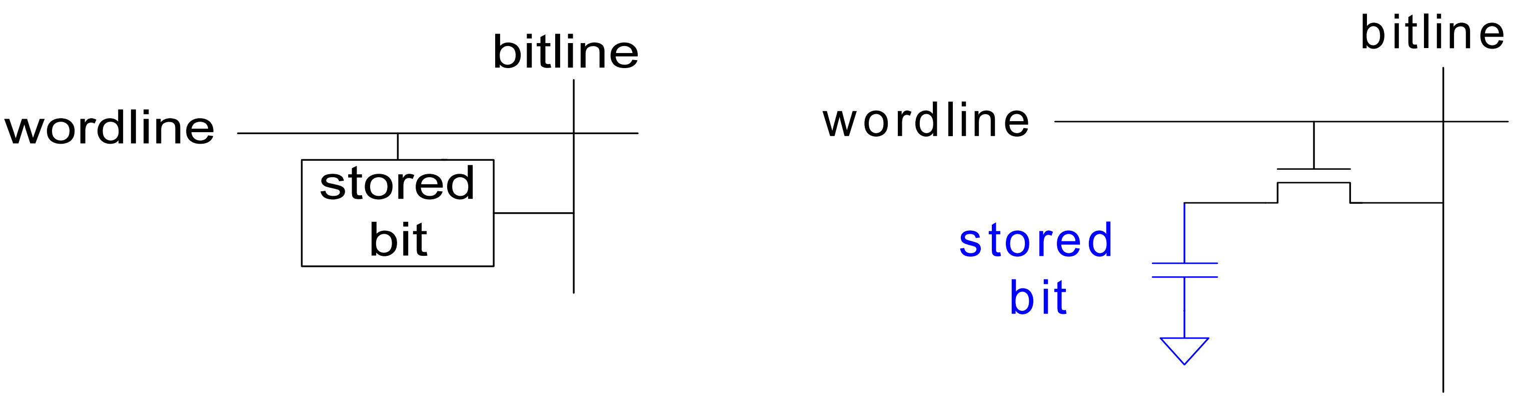
- Zwei wesentliche Typen:
  - Dynamisches RAM (DRAM)
  - Statisches RAM (SRAM)
- Verwenden unterschiedliche Speichertechniken in den Bit-Zellen:
  - DRAM: Kondensator
  - SRAM: Kreuzgekoppelte Inverter

Robert Dennard, 1932 -

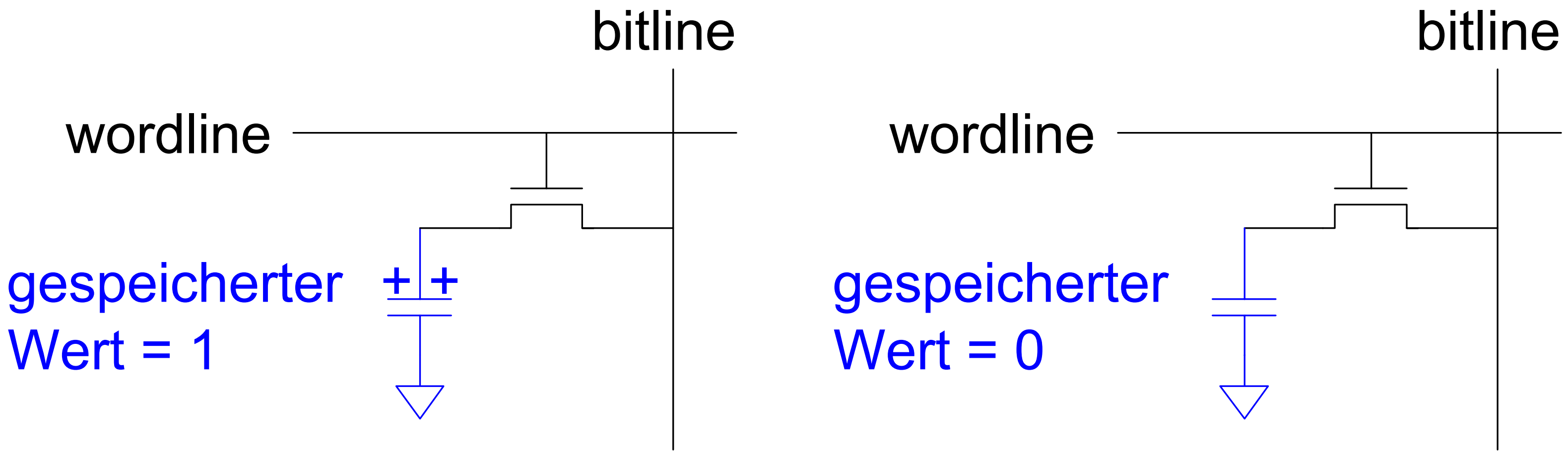
- Erfand 1966 bei IBM das DRAM
- Anfangs große Skepsis, ob Technik praktikabel
- Seit Mitte der 1970er Jahre ist DRAM die am weitesten verbreitete Speichertechnik in Computern

DRAM Bit-Zelle

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

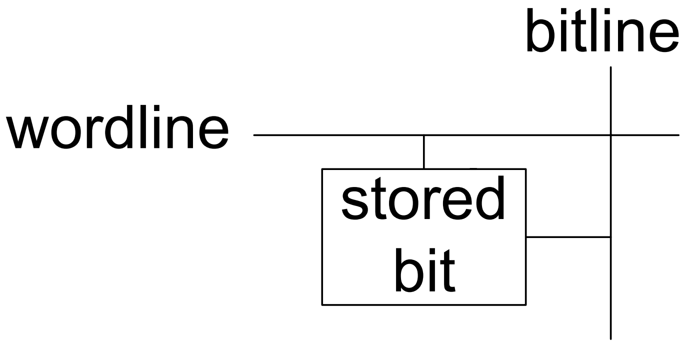


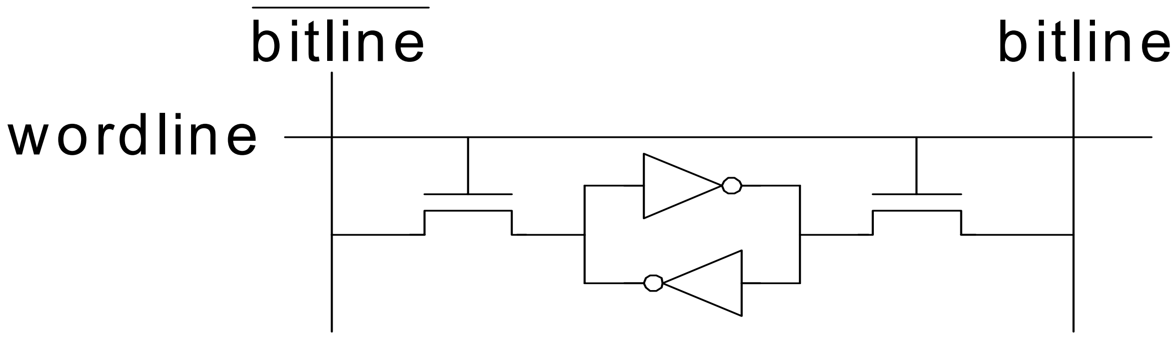
DRAM Bit-Zelle



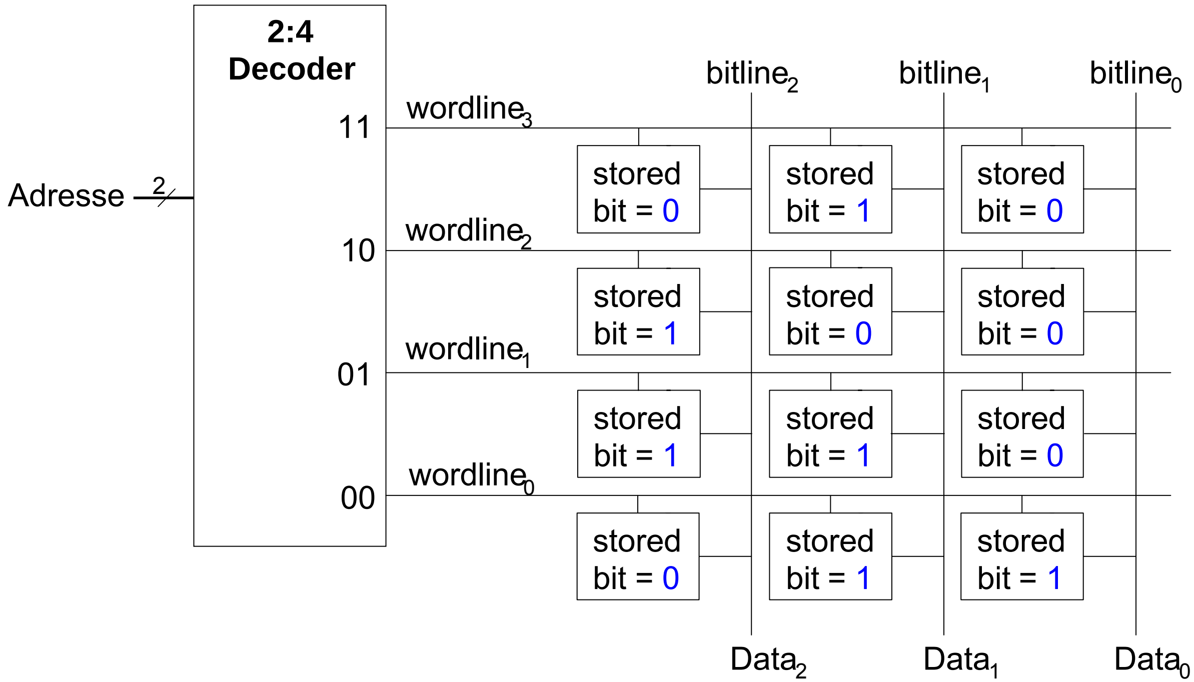
SRAM Bit-Zelle

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

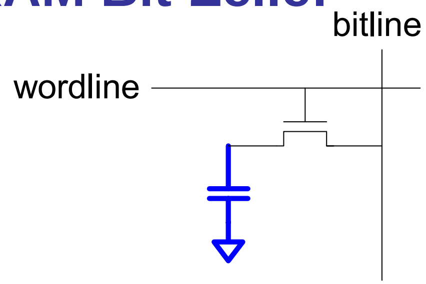




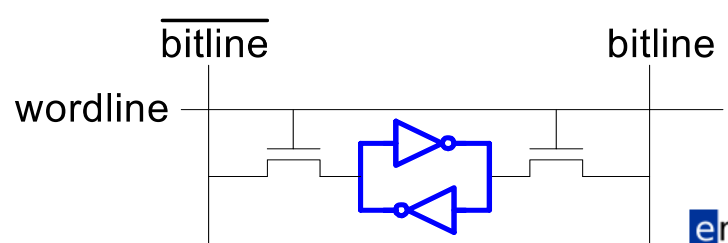
Speicherfelder



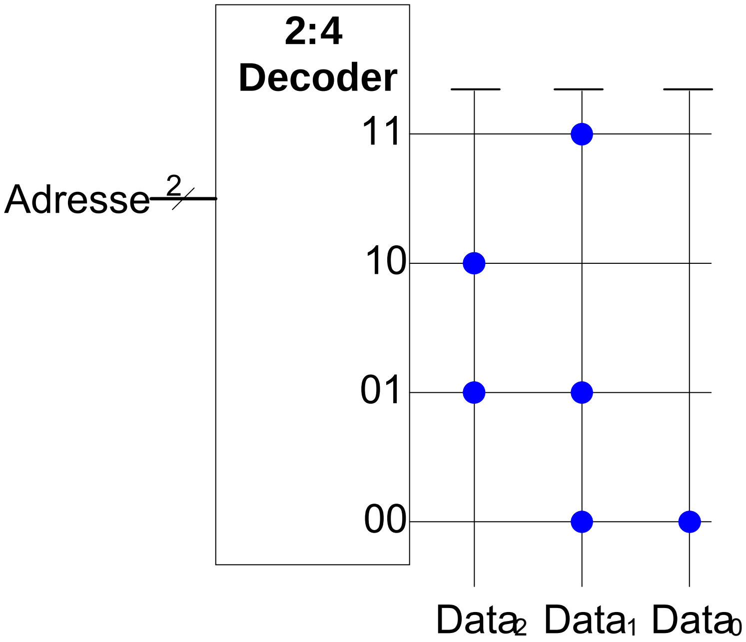
DRAM Bit-Zelle:

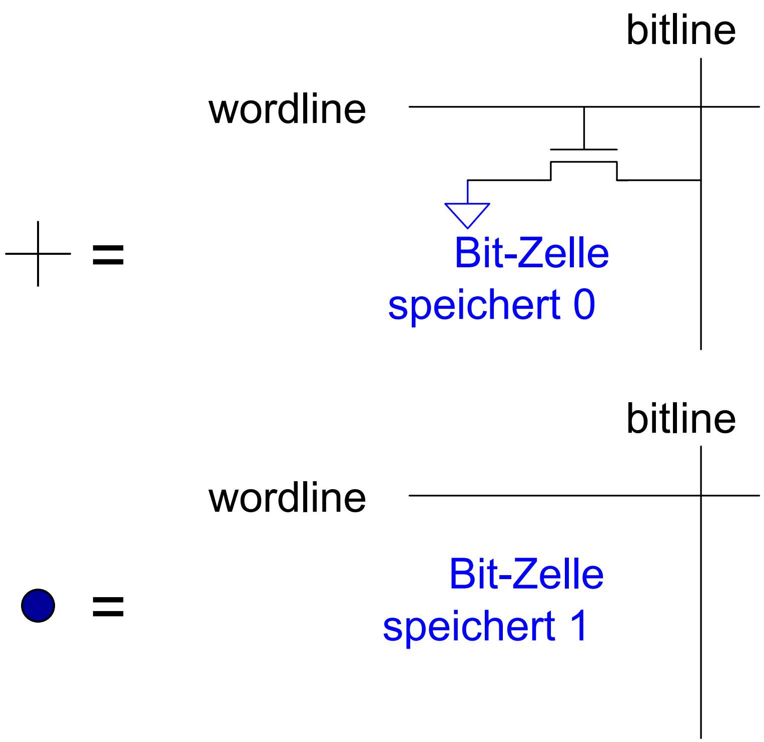


SRAM Bit-Zelle:



ROMs: Aufbau der Bit-Zellen



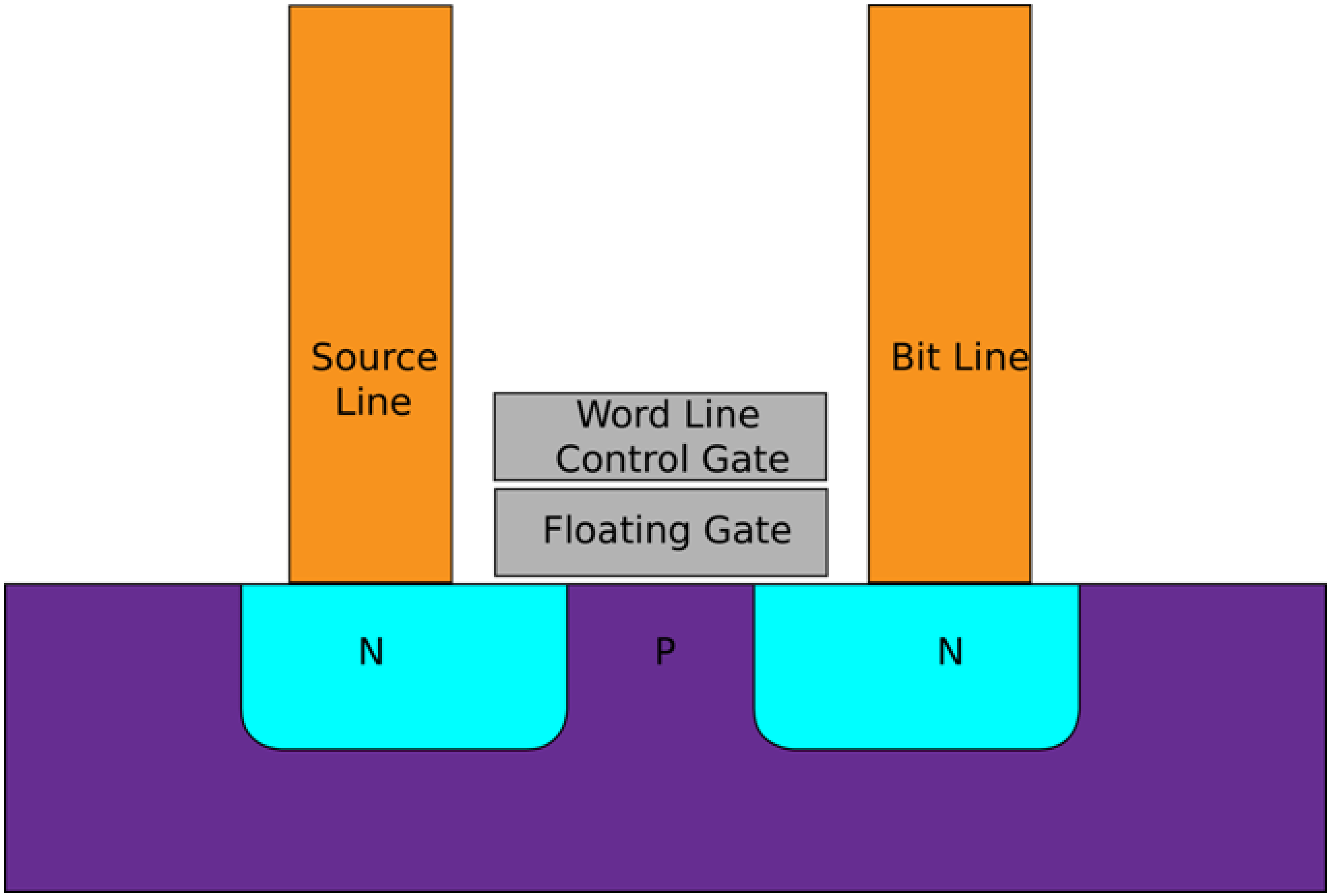


Bitlines sind schwach auf HIGH getrieben

Fujio Masuoka, 1944-

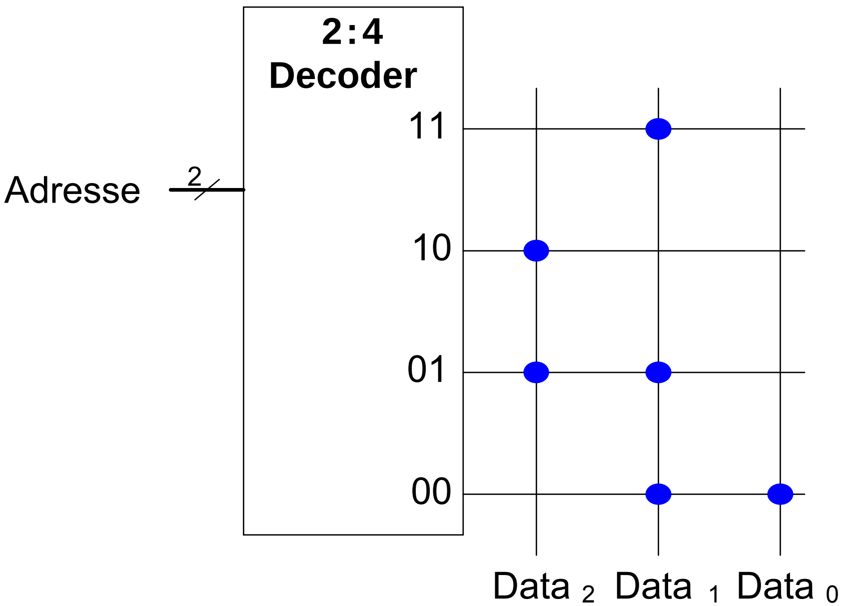
- Entwickelte Speicher und schnelle Schaltungen bei Toshiba von 1971-1994
- Erfand Flash-Speicher als eigenes ungenehmigtes Projekt in den späten 1970ern
  - An Wochenenden und abends
- Löschvorgang erinnerte ihn an Kamerablitz
  - Deshalb Flash-Speicher
- Toshiba kommerzialisierte Technik nur zögerlich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

Flash-Speicher: Bit-Zelle



Quelle:Wikipedia

ROMs als Datenspeicher

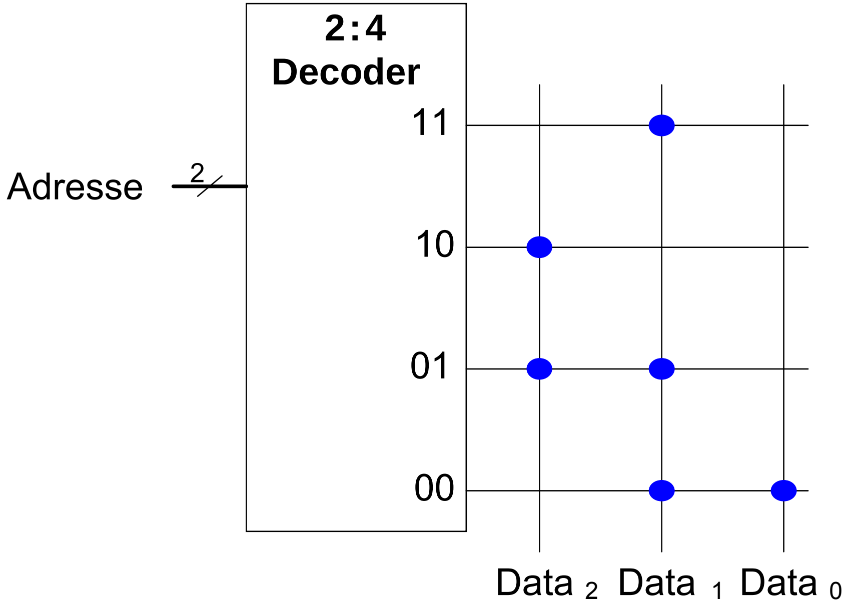


Adresse	Daten		
11	0	1	0
10	1	0	0
01	1	1	0
00	0	1	1

Tiefe

Breite

ROMs als Wertetabellen für boolesche Logik



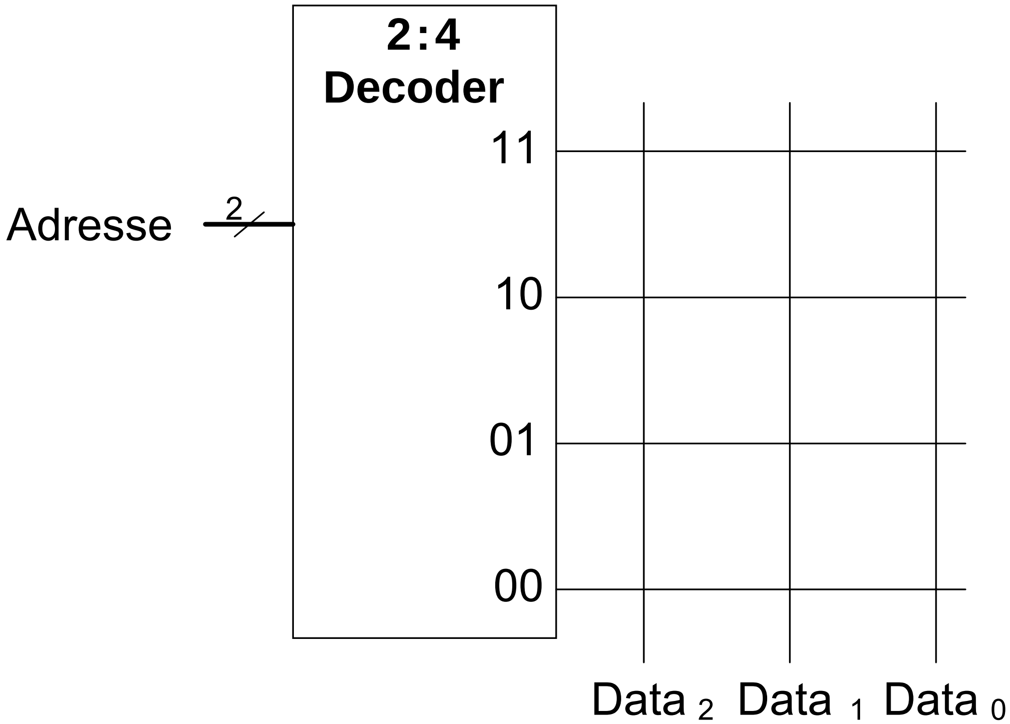
$$Data_2 = A_1 \oplus A_0$$

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

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

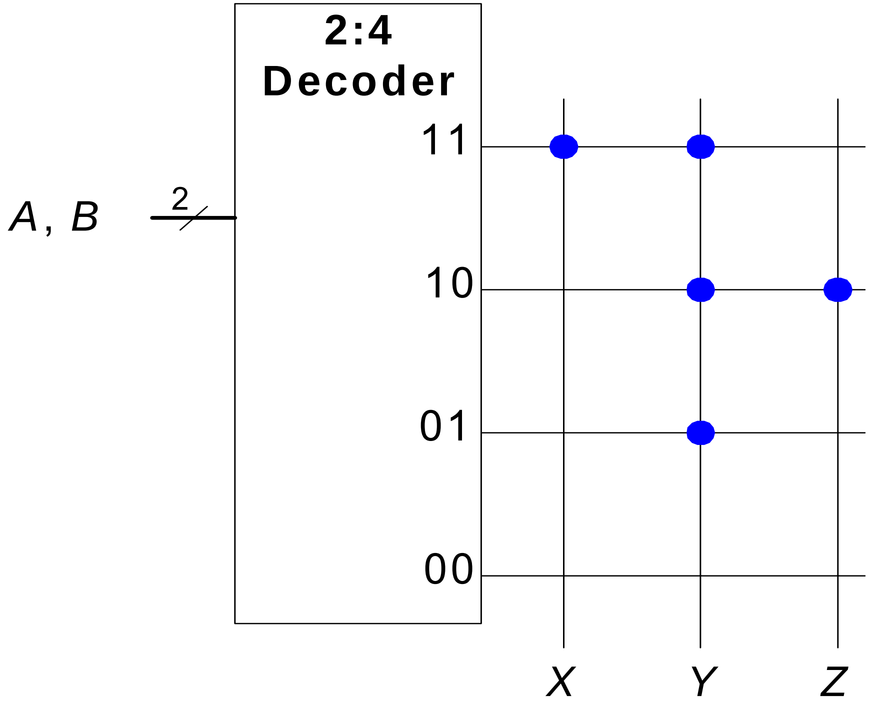
Beispiel: Logik aus ROMs

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

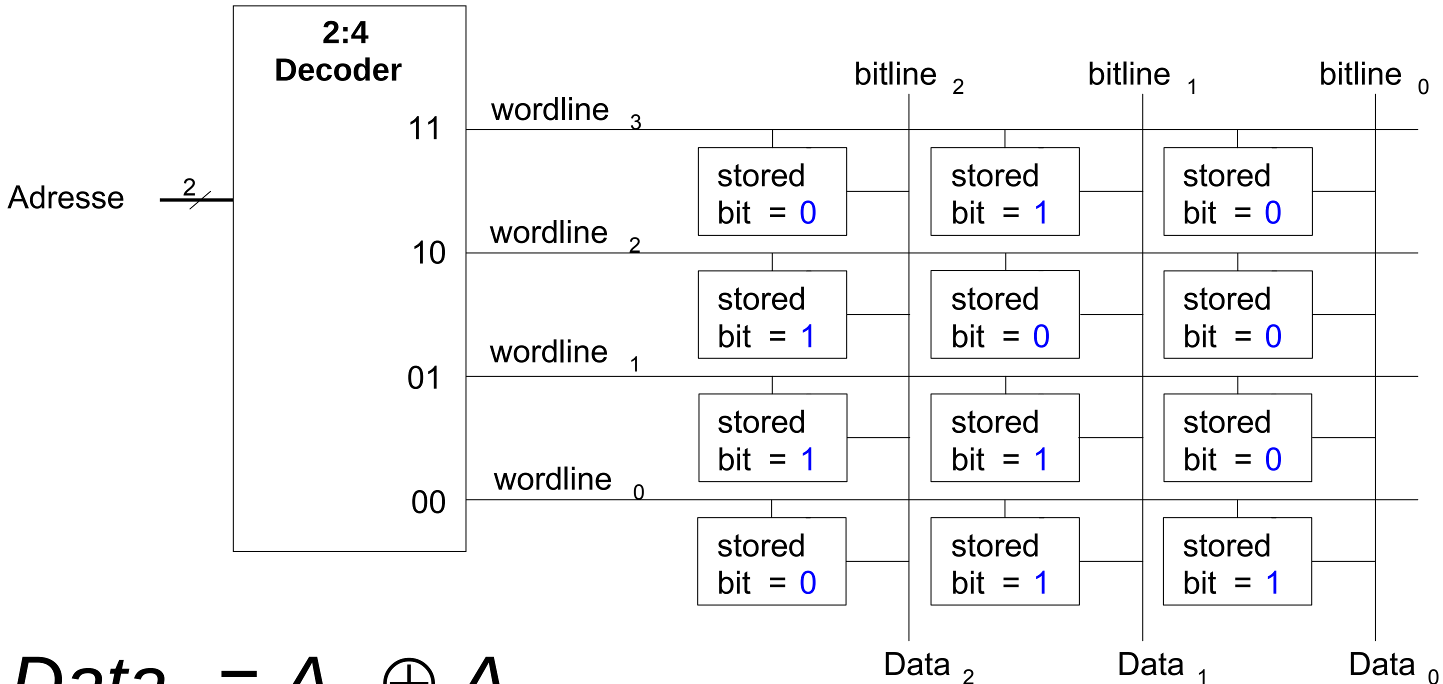


Beispiel: Logik aus ROMs

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



Logik aus beliebigem Speicherfeld



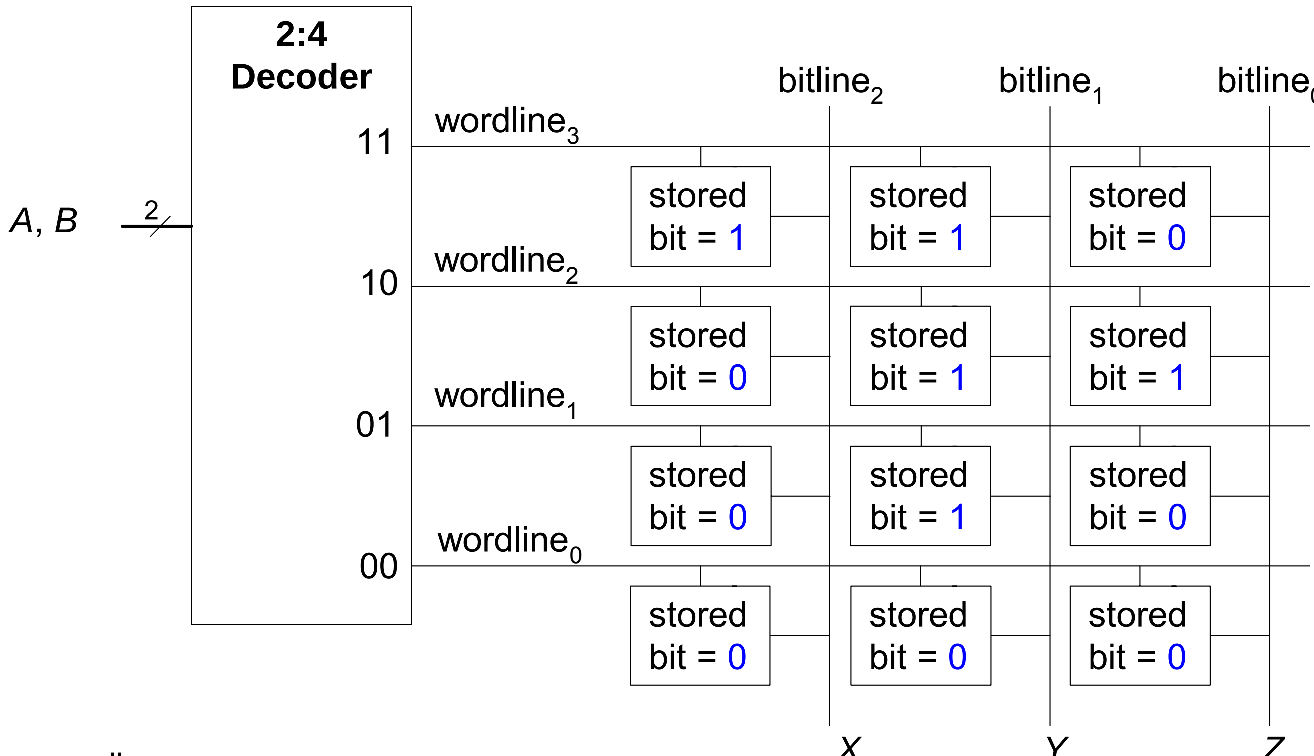
$$Data_2 = A_1 \oplus A_0$$

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

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

Logik aus beliebigem Speicherfeld

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



Andere Funktion nur durch Ändern der Speicherinhalte

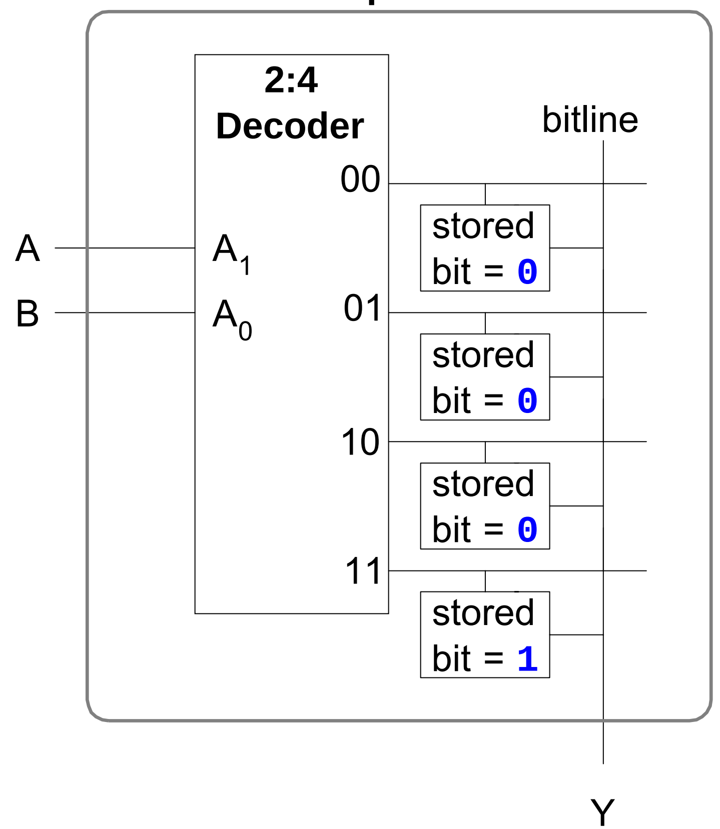
Logik aus beliebigen Speicherfeldern

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

Werte-tabelle

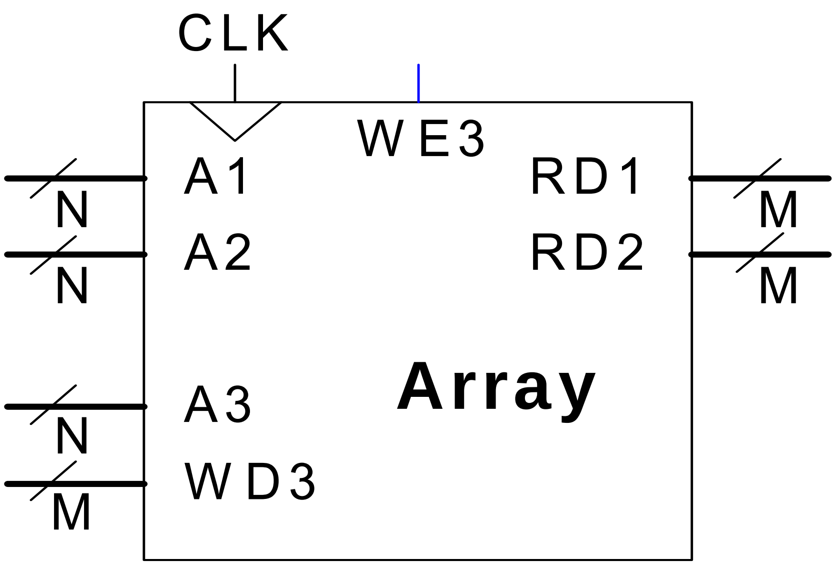
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

4-Wort x 1-bit Speicherfeld



Multi-Port-Speicher

- **Port:** Zusammengehörige Anschlüsse für Adresse und Datum
- Drei-Port Speicher
  - 2 Lese-Ports (A1/RD1, A2/RD2)
  - 1 Schreib-Port (A3/WD3, Signal WE3 löst Schreiben aus)
- Kleine Multi-Port-Speicher werden als Registerfelder bezeichnet
  - Werden z.B. in Prozessoren eingesetzt



Speicherfeld in Verilog

```
// 256 x 3b Speicher mit einem Schreib/Lese-Port

module dmem( input              clk, we,
             input  [7:0]       a
             input  [2:0]       wd,
             output [2:0]       rd);

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

  assign rd = RAM[a];

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

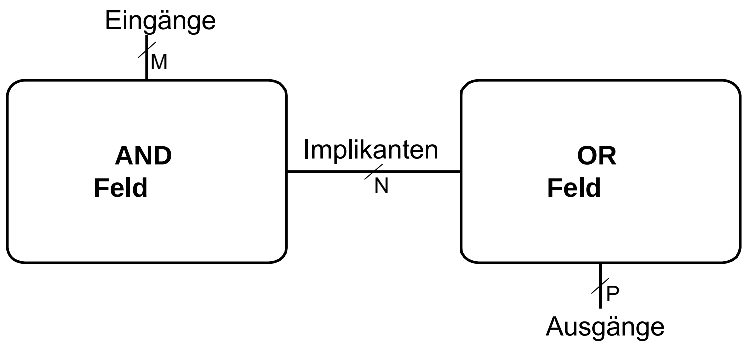
Logikfelder (*logic arrays*)

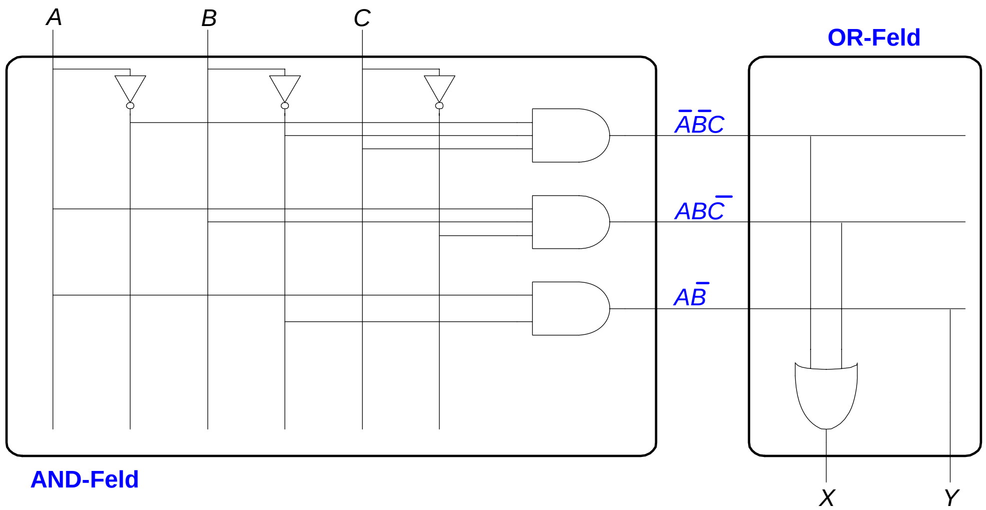
- Programmable Logic Arrays (PLAs)
  - AND Feld gefolgt von OR Feld
  - Kann nur kombinatorische Logik realisieren
  - Feste interne Verbindungen, spezialisiert für DNF (SoP-Form)

- Field Programmable Gate Arrays (FPGAs)
  - Feld von konfigurierbaren Logikblöcken (CLBs)
  - Können kombinatorische und sequentielle Logik realisieren
  - Programmierbare Verbindungsknoten zwischen Schaltungselementen

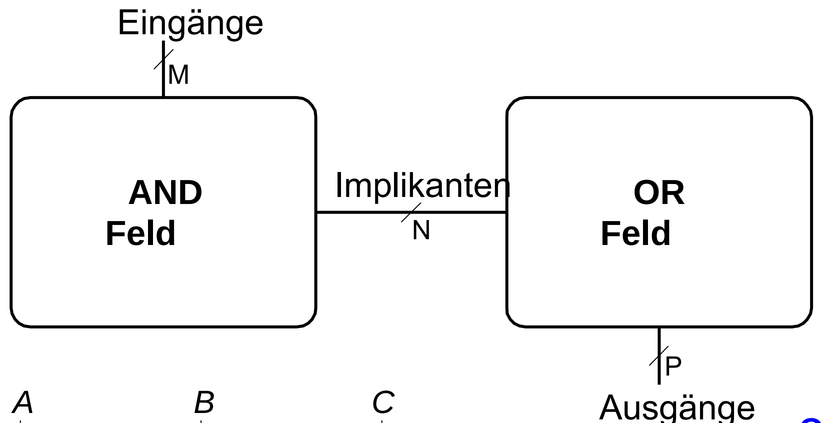
Boole‘sche Funktionen mit PLAs: Idee

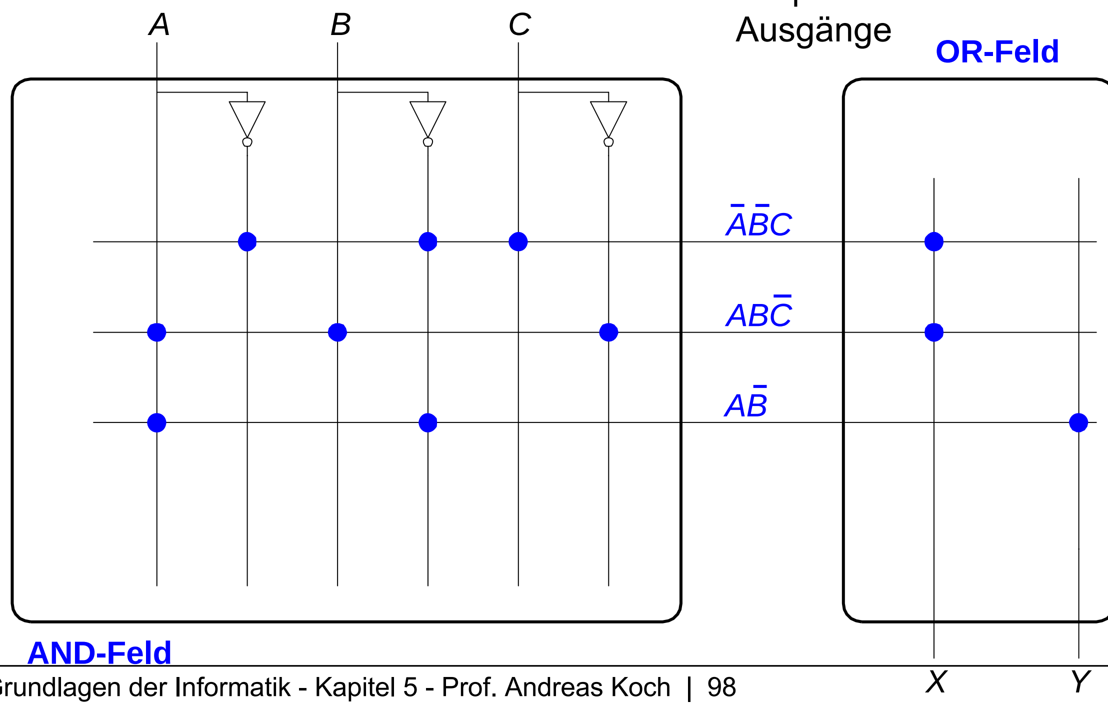
- $X = \overline{A}\,\overline{B}C + AB\overline{C}$
- $Y = A\overline{B}$





PLAs: Vereinfachte Schreibweise



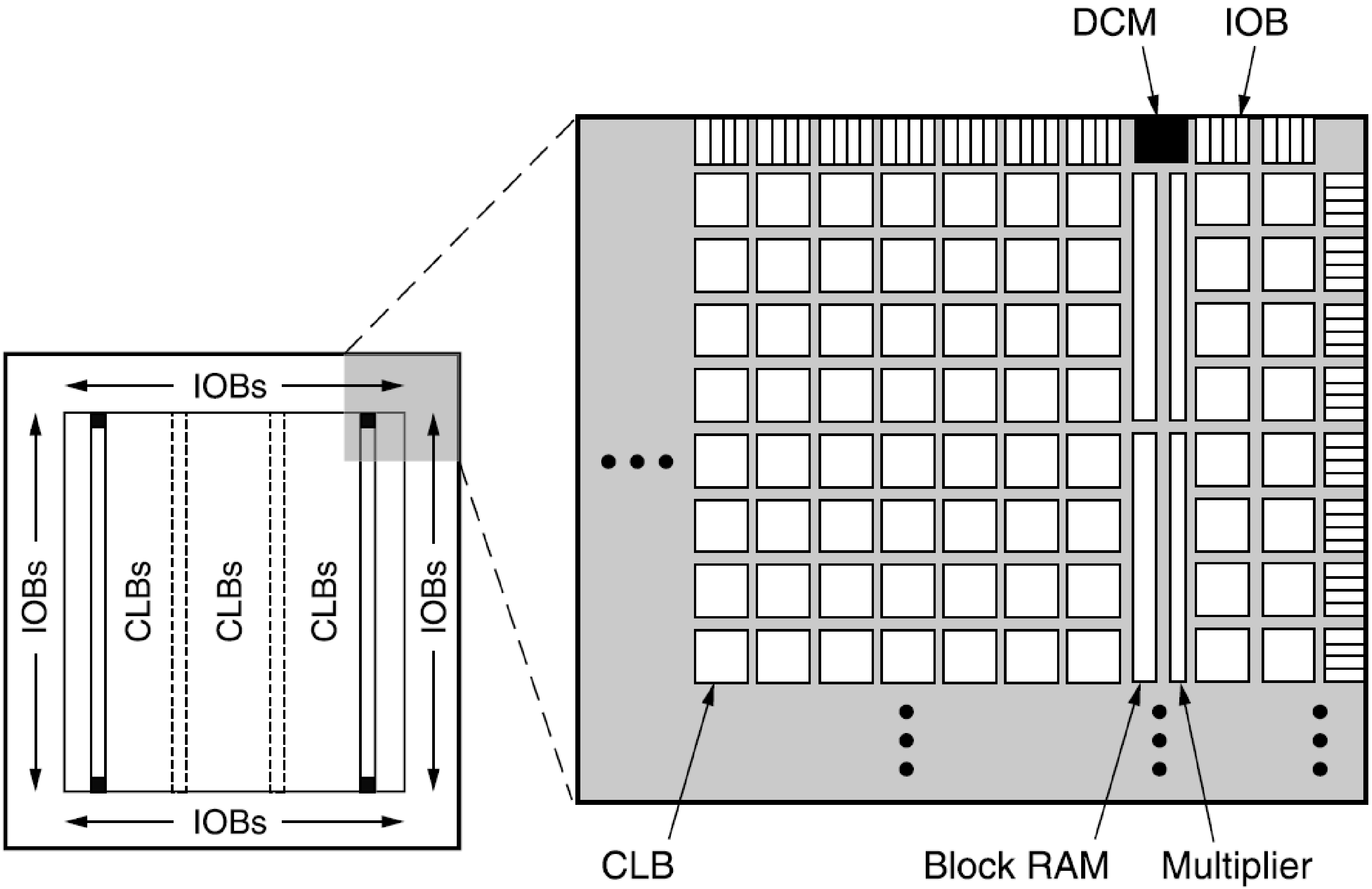


FPGAs: Field Programmable Gate Arrays

- 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

- Reale FPGAs enthalten oftmals noch weitere Arten von Blöcken
  - RAM
  - Multiplizierer
  - Manipulation von Taktsignalen (DCM)
  - Sehr schnelle serielle Verbindungen (11 Gb/s)
  - Komplette Mikroprozessoren
  - …

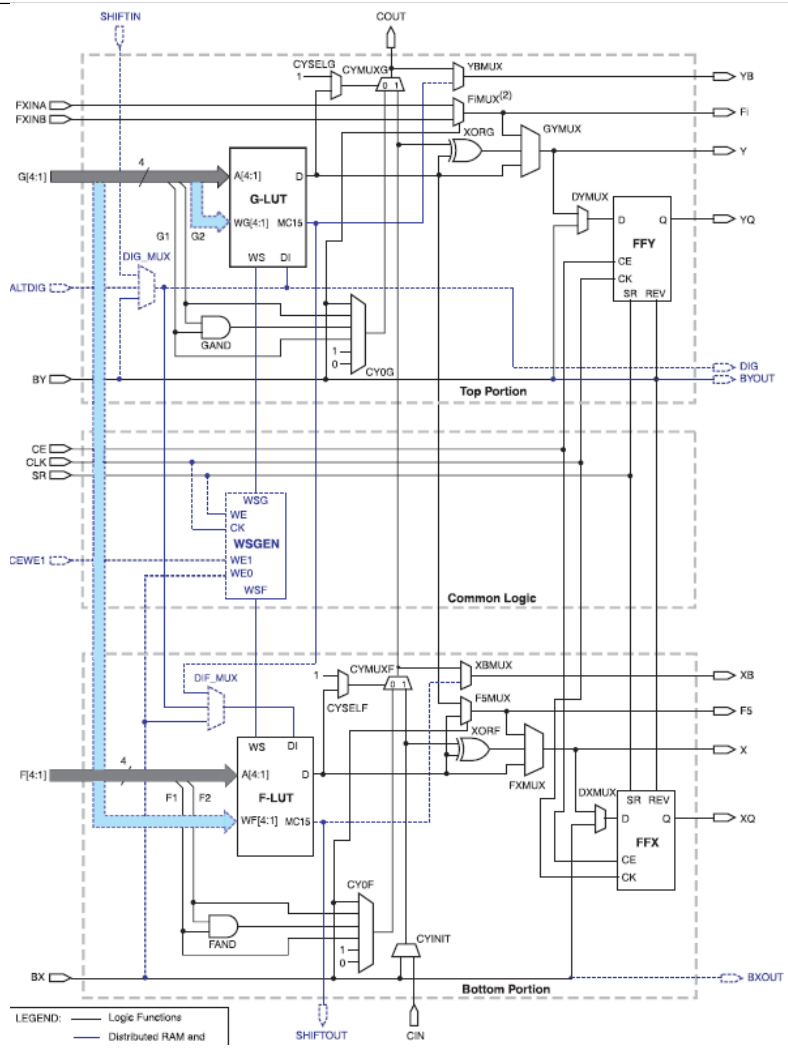
Xilinx Spartan 3 FPGA Schematic



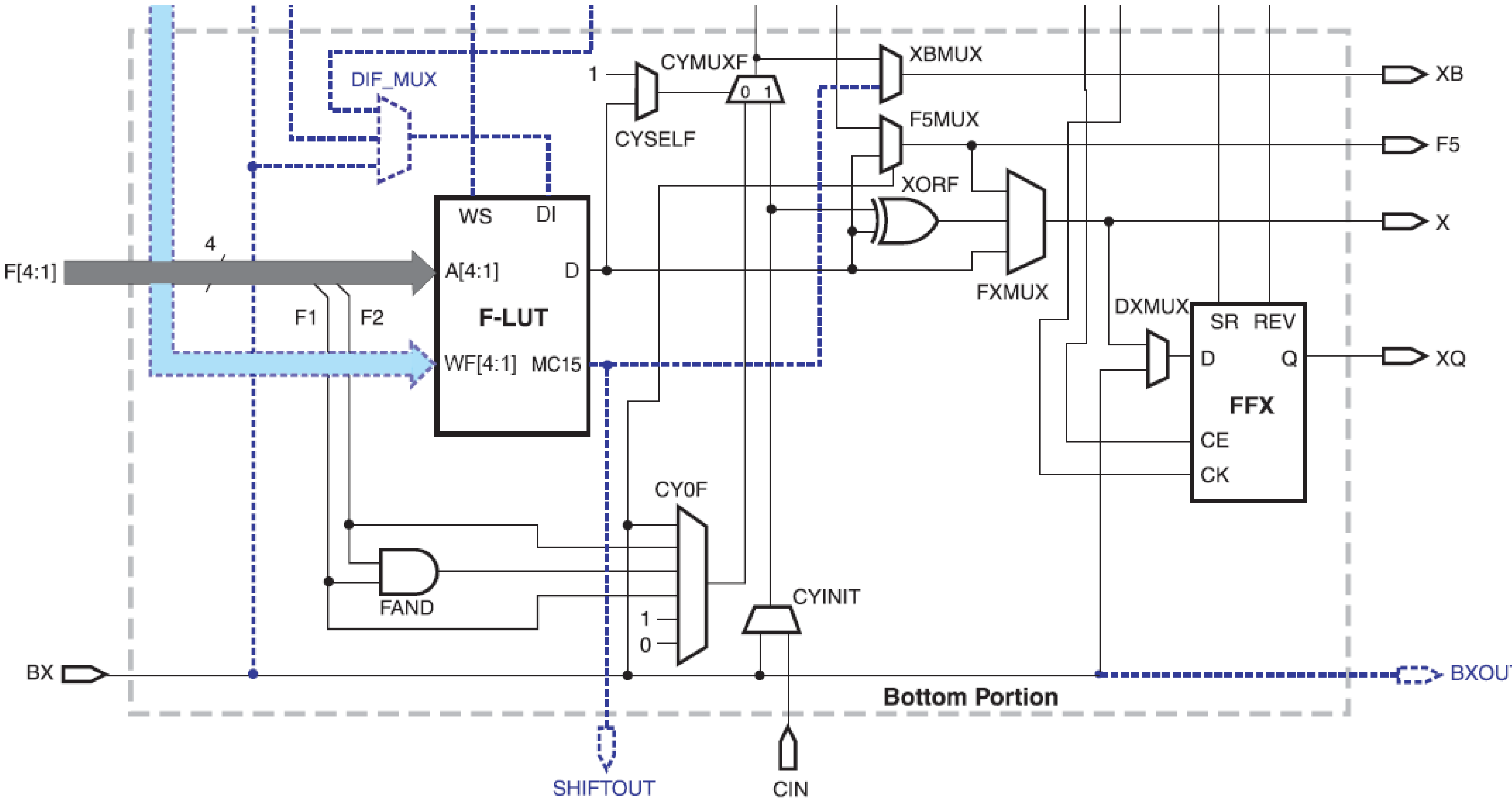
Konfigurierbare Logikblöcke (CLBs)

- Bestehen im wesentlichen aus:
  - LUTs (lookup tables): realisieren kombinatorische Funktionen
  - Flip-Flops: realisieren sequentielle Funktionen
  - Multiplexern: Verbinden LUTs und Flip-Flops

Xilinx Spartan 3 CLB



Xilinx Spartan CLB

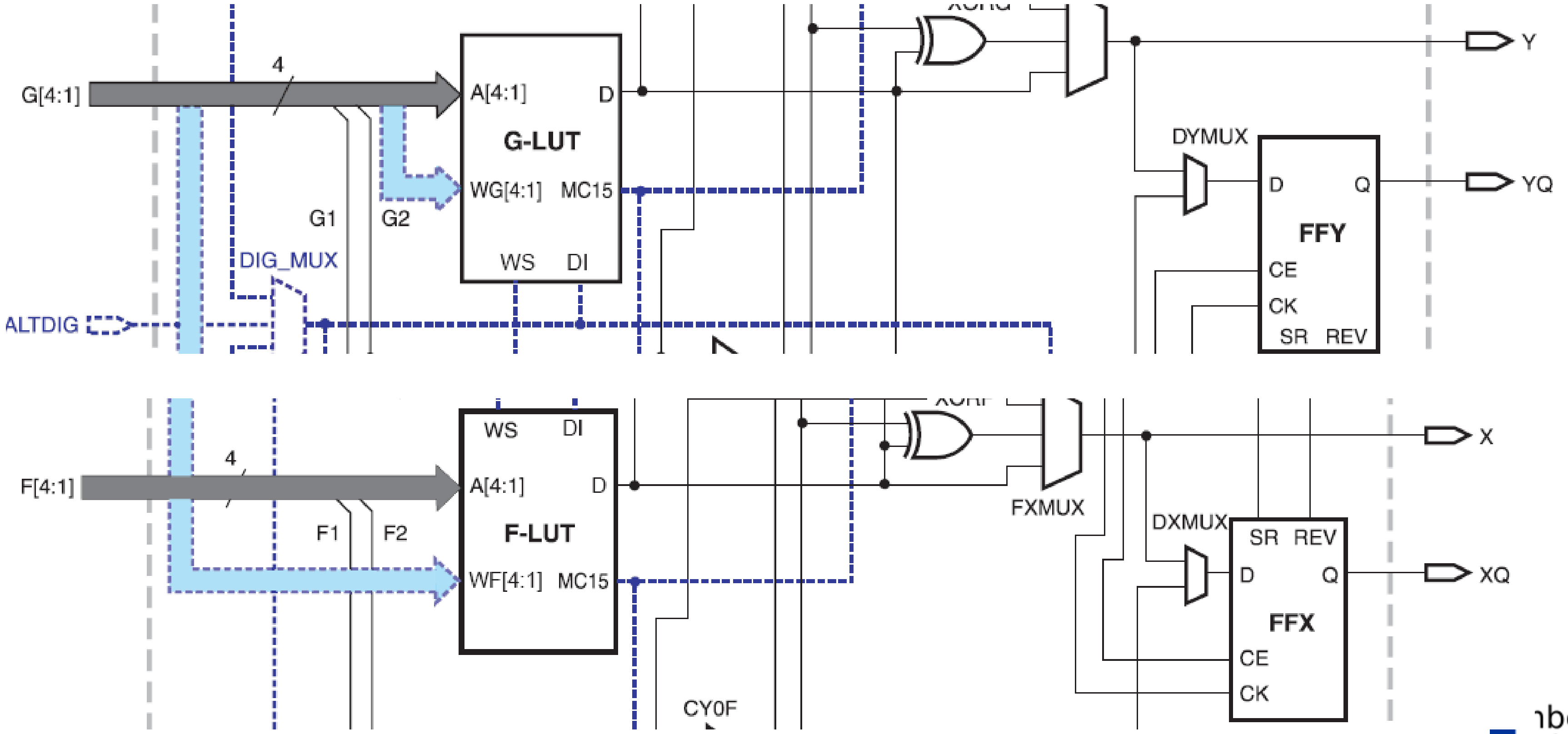


Xilinx Spartan 3 CLB

- Ein Spartan 3 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*

Beispiel: Kombinatorische Logik mit CLBs

- Berechnung der folgenden Funktionen mit dem Spartan 3 CLB
  - $X = \overline{A}B\overline{C} + AB\overline{C}$
  - $Y = A\overline{B}$

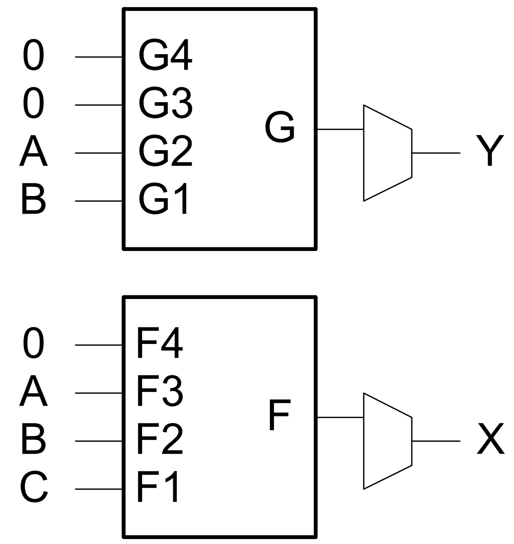


Beispiel: Kombinatorische Logik mit CLBs

- Berechnung der folgenden Funktionen mit dem Spartan 3 CLB
  - $X = \overline{A}\,\overline{B}C + AB\overline{C}$
  - $Y = A\overline{B}$

	(A)	(B)	(C)	(X)
F4	F3	F2	F1	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

		(A)	(B)	(Y)
G4	G3	G2	G1	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: Xilinx ISE
- Ist in der Regel ein iterativer Prozess
  - Planen
  - Implementieren
  - Testen
  - Wiederhole …
- Entwickler denkt nach
- Entwickler gibt Entwurf als Schaltplan oder HDL-Beschreibung ein
- Entwickler wertet Simulationergebnisse aus
- Wenn Simulation zufriedenstellend: Synthetisiere Entwurf in Netzliste
- Bilde Netzliste auf FPGA-Konfiguration ab (CLBs, IOBs, Verbindungsnetz)
- Lade Konfigurationsdaten (*bit stream*) auf FPGA
- Teste Schaltung nun in realer Hardware