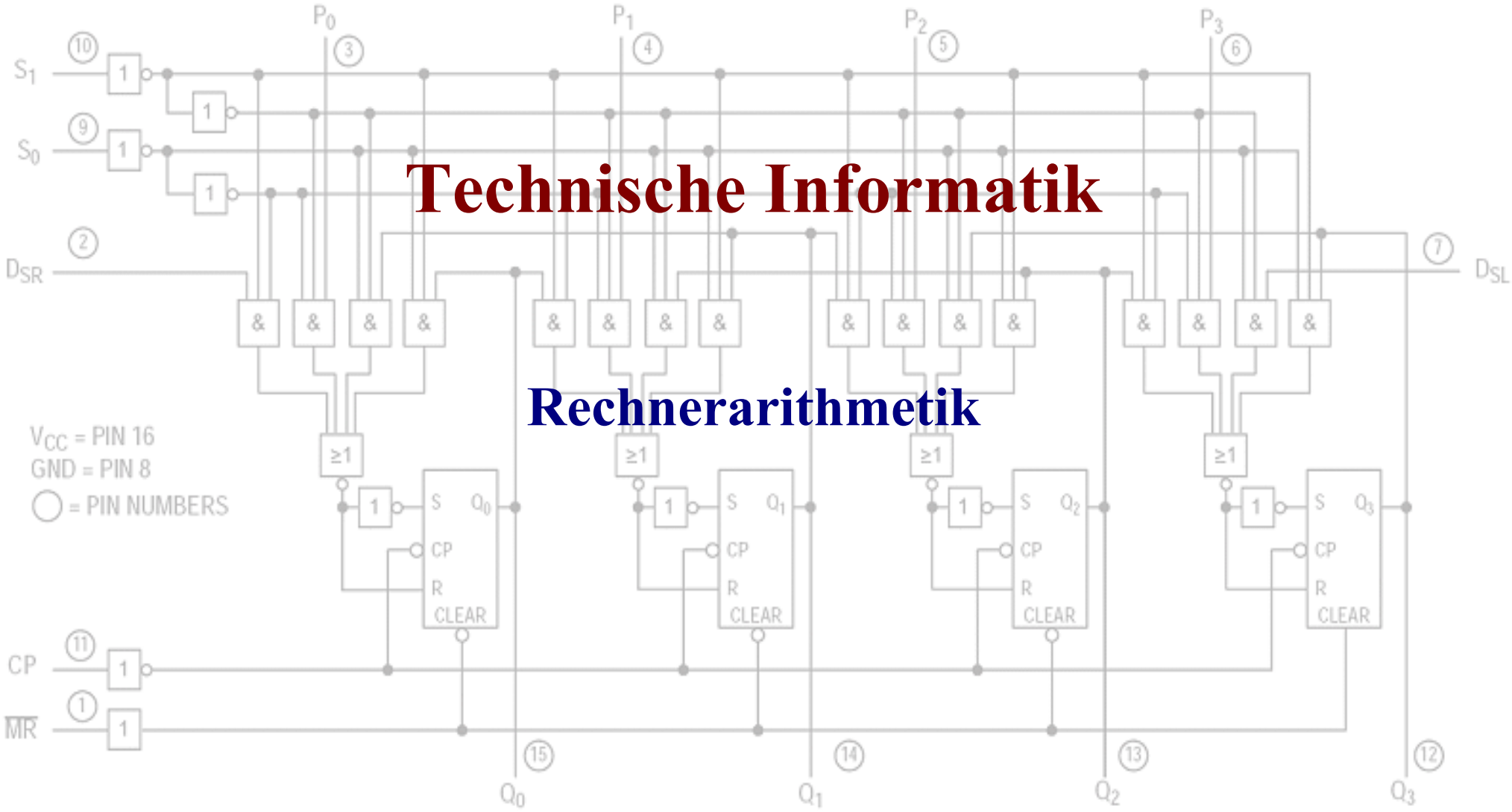
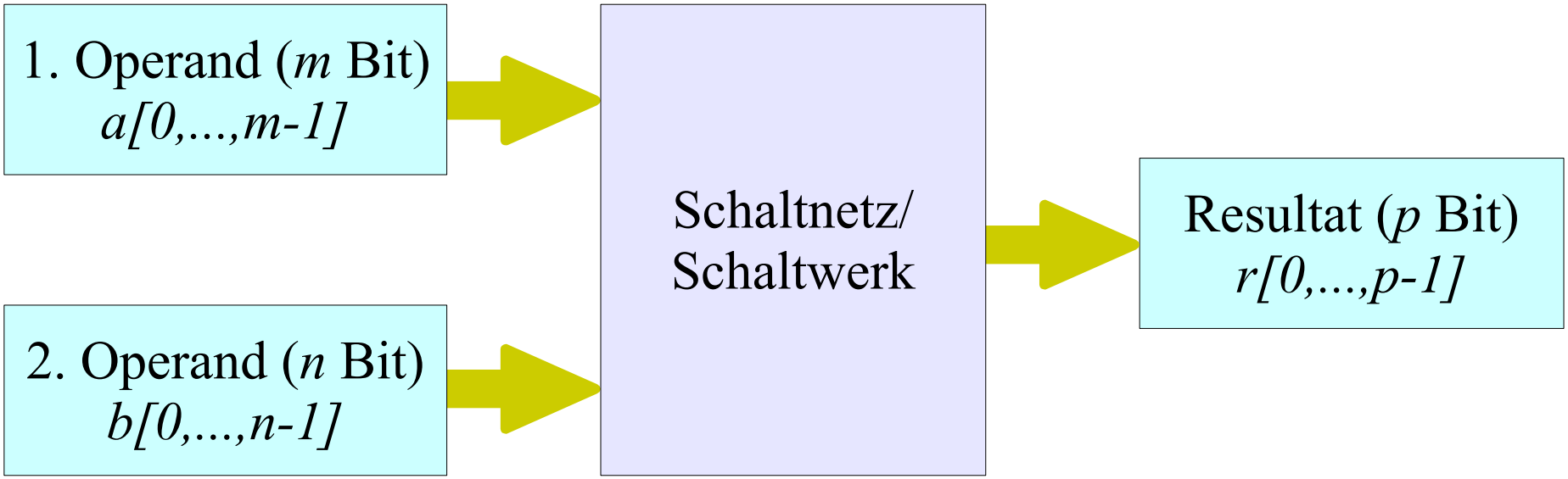
Technische Informatik

Rechnerarithmetik



Grundprinzipien

- Grundstruktur der Implementierung einer Rechenoperation
  - Je nach Operation entfällt der 2. Operand
  - Eingangswortbreiten können verschieden sein
  - Ausgangswortbreite i.d.R größer als Eingangswortbreite



Grundprinzipien

- **1. Ansatz**: Direkte Codierung in Schaltnetzen
  - Methode: Wahrheitstabelle für alle Zahlen aufstellen und direkt in Gattern oder Look-Up-Tabellen implementieren
  - Vorteil: potentiell schnelle Befehlsausführung
  - Nachteil: sehr großer Schaltungsaufwand, praktisch nicht realisierbar

    bei **16-Bit** Arithmetik: $2^{2 \cdot 16}$ Datenwort => **8 GByte**

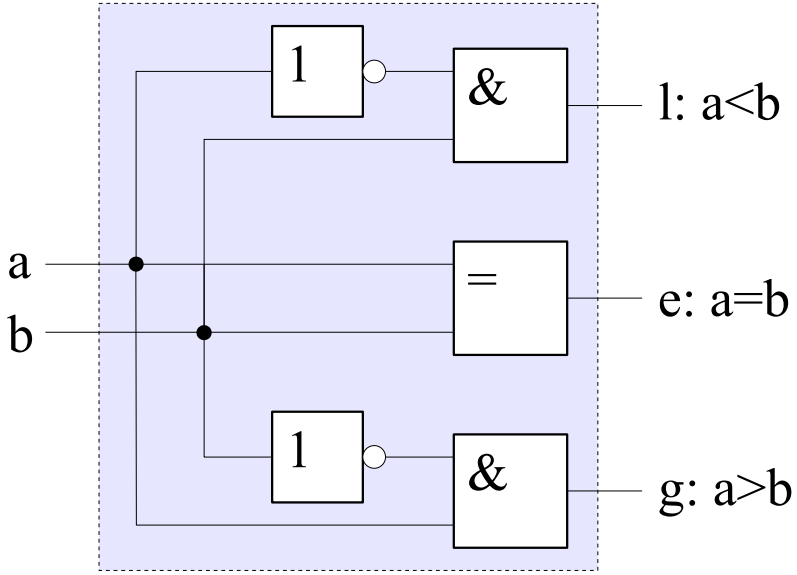
    bei **32-Bit** Arithmetik: $2^{2 \cdot 32}$ Datenwort => **$8 \cdot 10^{19}$ Byte**

- **2. Ansatz**: Rückführung auf Basisoperationen und Implementierung der Gesamtfunktion in Schaltwerken
  - Methode: Aufspalten der Zielfunktion in Teilfunktionen. Lösung der Aufgabe durch iteratives Ausführen der Teilfunktionen
  - Vorteil: praktisch realisierbar
  - Nachteil: i.d.R. Geschwindigkeitskompromiss

Komperator

- Komperator: Schaltung zum Vergleich zweier Dualzahlen A und B:
  $A > B$, $A < B$ oder $A = B$?

- 1 Bit Wortbreite:



a	*b*	*l:a<b*	*e:a=b*	*g:a>b*
0	0	0	1	0
0	1	1	0	0
1	0	0	0	1
1	1	0	1	0

$$l = \bar{a} \cdot b$$
$$e = a \cdot b + \bar{a} \cdot \bar{b}$$
$$g = a \cdot \bar{b}$$

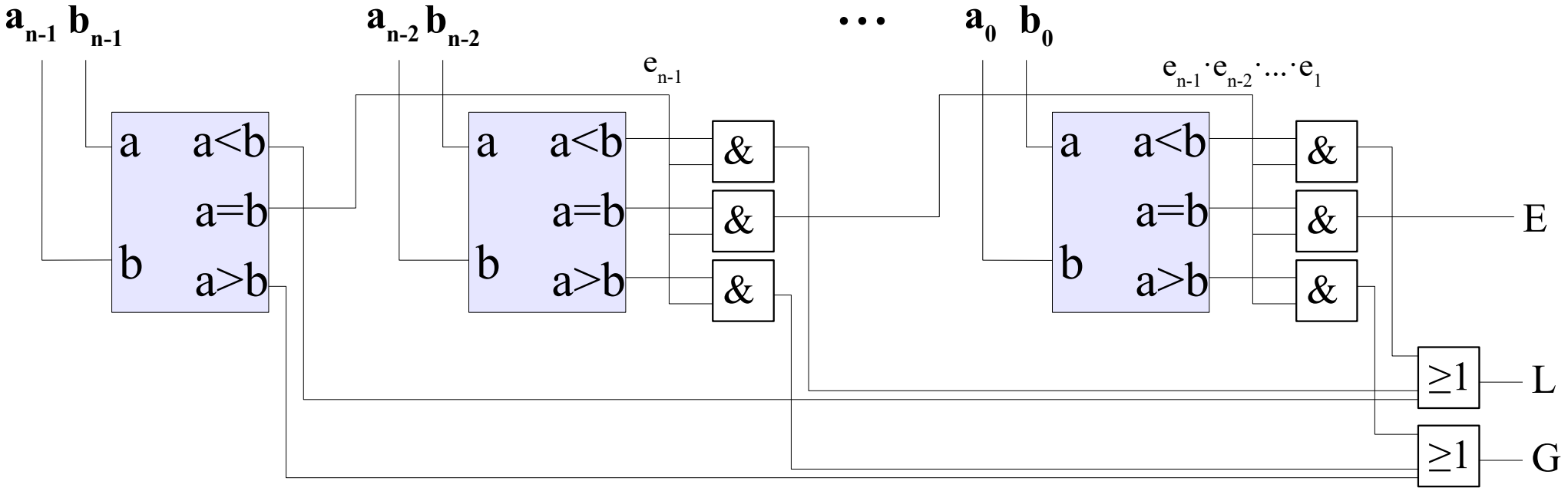
Komperator

- Erweiterung auf n Bit durch Kaskadierung:

$$E:(A=B) = e_{n-1} \cdot e_{n-2} \cdot \ldots \cdot e_0$$

$$L:(A<B) = l_{n-1} + (e_{n-1}) \cdot l_{n-2} + \ldots + (e_{n-1} \cdot e_{n-2} \cdot \ldots \cdot e_1) \cdot l_0$$

$$G:(A>B) = g_{n-1} + (e_{n-1} \cdot g_{n-2} + \ldots + e_{n-1} \cdot e_{n-2} \cdot \ldots \cdot e_1) \cdot g_0$$



Addierer

- Addition einer mehrstelligen Dualzahl (positive Ganzzahl oder Zweierkomplement)
- Beispiel:

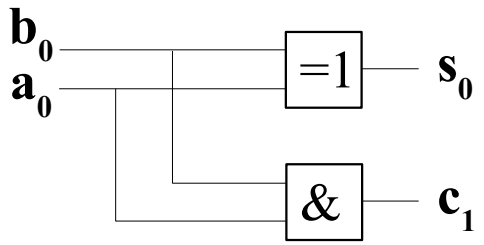
```
    0 1 0 1 1
+   0 1 1 1 0
  ---------------  <- Übertrag
  = 1 1 0 0 1
```

$$c_i = (a_i \oplus b_i) \cdot c_i + (a_i \cdot b_i)$$

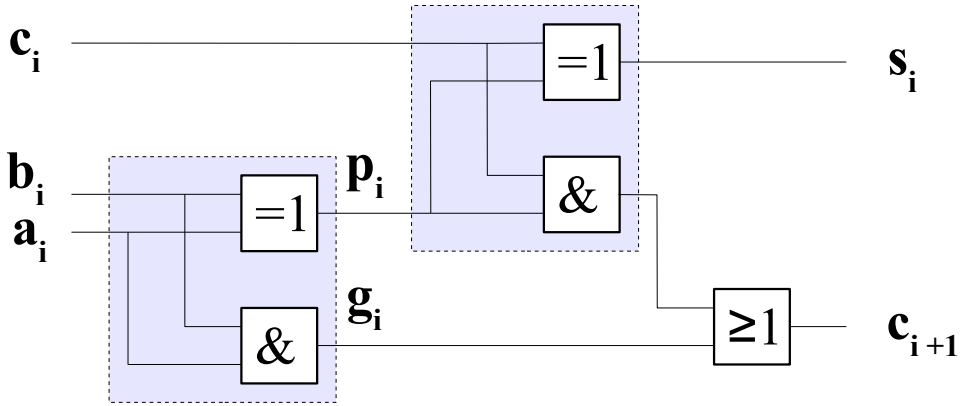
$$s_i = (a_i \oplus b_i) \oplus c_i$$

- Basiselemente:
  - Halbaddierer (HA) zur Addition der niederwertigsten Bits (ohne Übertragseingang)
  - Volladdierer (VA) zur Addition aller höherwertigen Bits

Halbaddierer:

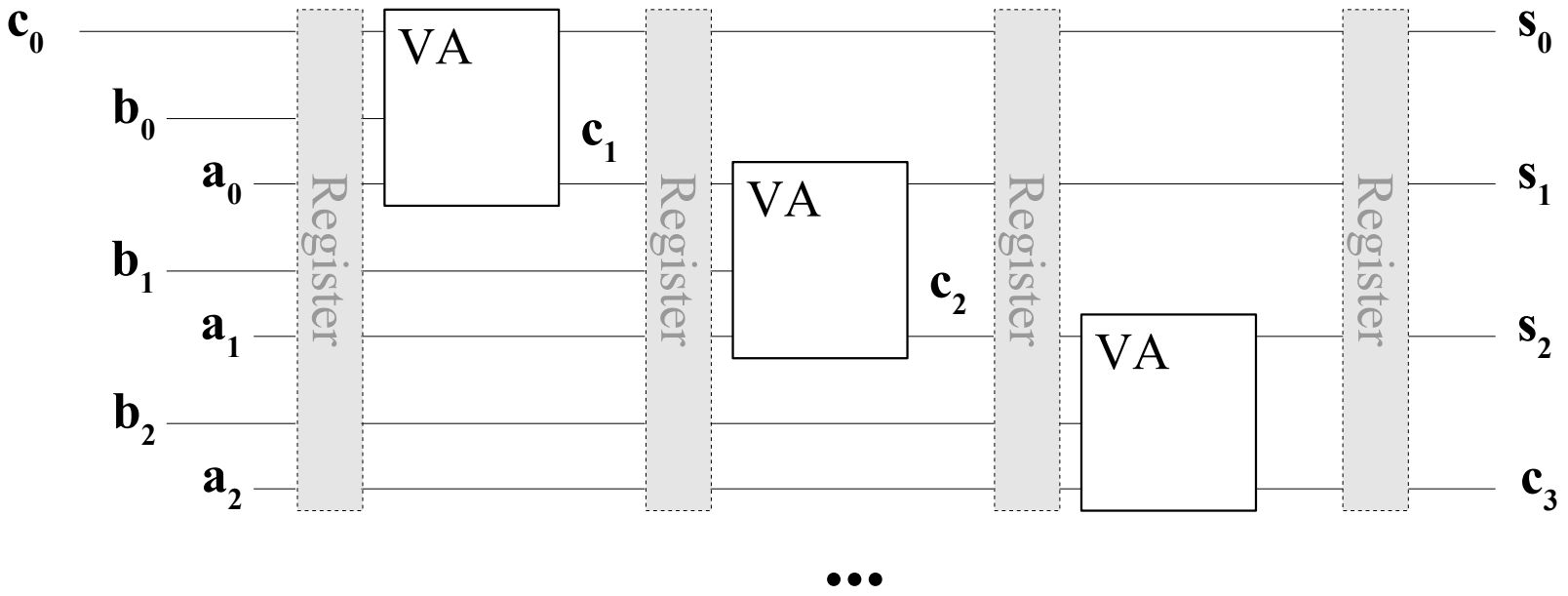


Volladdierer:



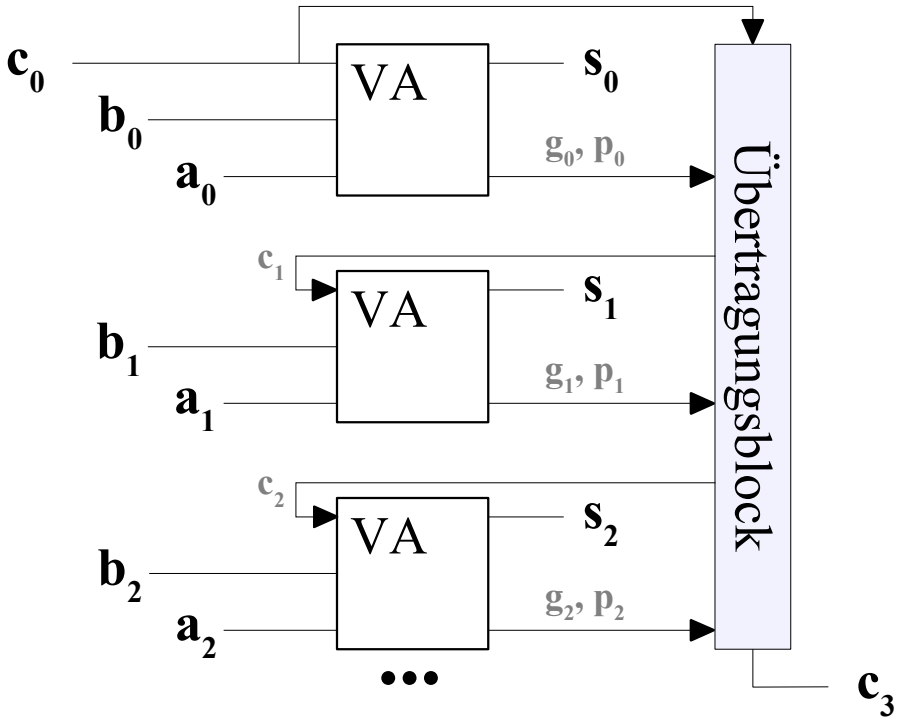
Addierer mit seriellem Übertrag

- Addition *n*-stelliger Dualzahlen durch Kaskadierung (Ripple Carry-Addierer)
  - Für jede Stelle wird ein Volladdierer benötigt, für die niederwertigste Stelle reicht ggf. ein Halbaddierer
  - Rechenzeit proportional zur Wortbreite *n*
  - Hazard: Das Ergebnis ist erst gültig, wenn Signal durch alle Gatter gelaufen ist!
  - Variante: Pipeline-Struktur mit Register für Zwischenergebnisse



Addierer mit parallelem Übertrag

- Schnelle Addition *n*-stelliger Dualzahlen durch direkte Berechnung des Übertrags (Carry Lock-Ahead - Addierer)
  - Für jede Stelle wird ein Volladdierer benötigt, für die niederwertigste Stelle reicht ggf. ein Halbaddierer
  - Die Carry-Bits werden (nicht-rekursiv) aus den Zwischenwerte p_i (Propagate) und g_i (Generate) des Volladdierers berechnet (Übertragungsblock)



Generate-Bit (Übertrag wegen Eingangskombination):

$$g_i = a_i \cdot b_i$$

Propagate-Bit (Übertrag vorheriger Stufe):

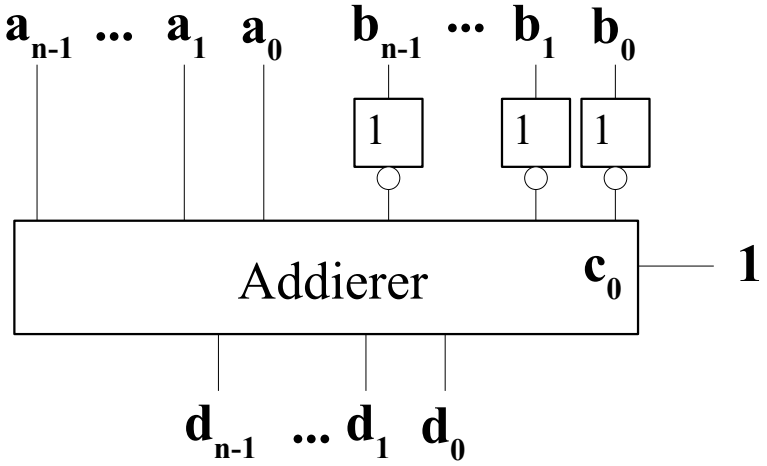
$$p_i = \overline{a_i} \cdot b_i + a_i \cdot \overline{b_i}$$

Carry-Bit (nicht-rekursive Berechnung möglich!):

$$\begin{aligned} c_{i+1} &= g_i + p_i \cdot c_i \\ &= g_i + p_i \cdot (g_{i-1} + p_{i-1} \cdot c_{i-1}) \\ &= g_i + p_i \cdot g_{i-1} + p_i \cdot p_{i-1} \cdot g_{i-2} + \ldots + p_i \cdot \ldots \cdot p_0 \cdot c_0 \end{aligned}$$

Addierer / Subtrahierer

- **Subtraktion**: Rückführung auf die Addition durch das Zweierkomplement
  - Invertierung aller Bits und Addition einer 1 (durch $c_0 = 1$)



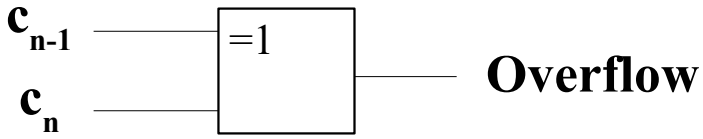
$$D = A - B = A + (-B)$$

$$mit: \; (-B) = (\overline{b_{n-1}} \cdot \ldots \cdot \overline{b_1} \cdot \overline{b_0})_{bin} + \mathbf{1}$$

  - **Overflow**
    - Durch Addition n-stelliger Zahlen mit gleichem Vorzeichen kann ein $(n+1)$-stelliges Ergebnis entstehen:

$$\begin{array}{lll} A>0 \;\; und \;\; B>0 \;\; und \;\; A+B<0 & \Rightarrow & overflow \\ A<0 \;\; und \;\; B<0 \;\; und \;\; A+B>0 & \Rightarrow & overflow \\ sonst & \Rightarrow & kein \;\; overflow \end{array}$$

    - In Zweierkomplement-Darstellung ist das Carry-Bit c_n alleine kein Merkmal für einen Überlauf!
    - Erkennung eines Überlaufs:
      Carry-Bits c_n und c_{n-1} sind verschieden



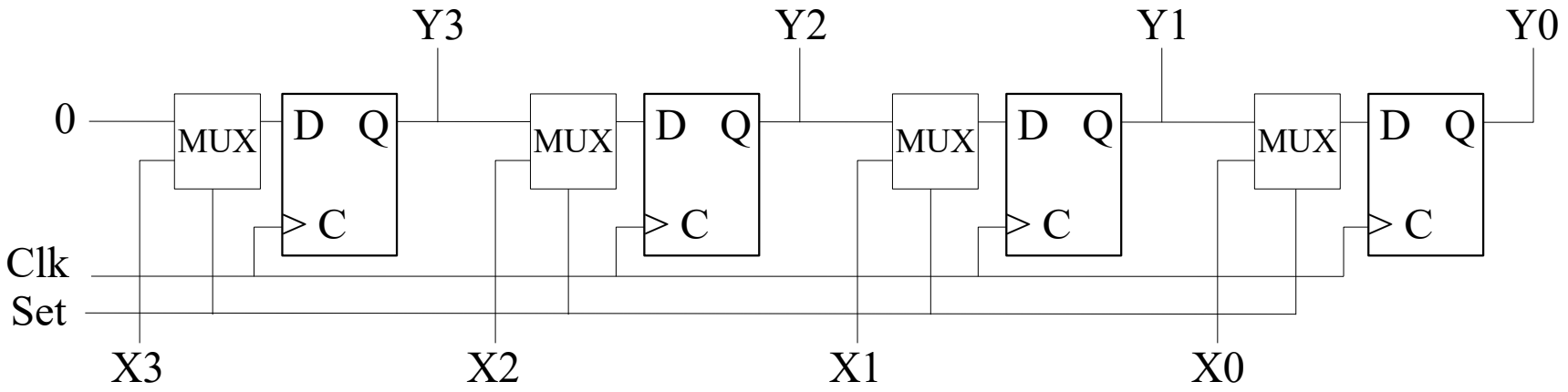
Shifter

- Verschiebung eines Bitmusters um *n* Bit
  - Nachrückende Stellen werden mit „0“ aufgefüllt
  - Rechts-Verschiebung → Division durch 2^n
  - Links-Verschiebung → Multiplikation mit 2^n
- Beispiel (Rechtsverschiebung):

$$12_{dez} = 1\ 1\ 0\ 0_{bin}$$

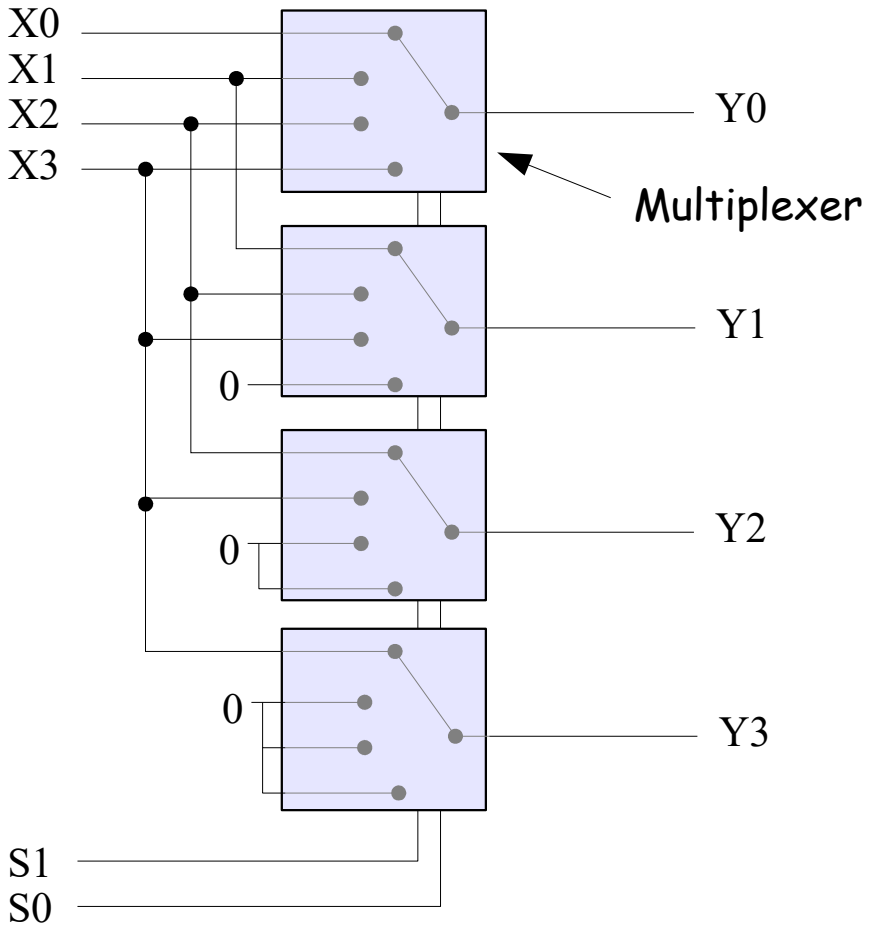
$$0\ 0\ 1\ 1_{bin} = 3_{dez}$$

- Implementierung durch Schieberegister
  - Startwert wird geladen (Multiplexer MUX)
  - *n*-fache Wiederholung des Schiebevorganges



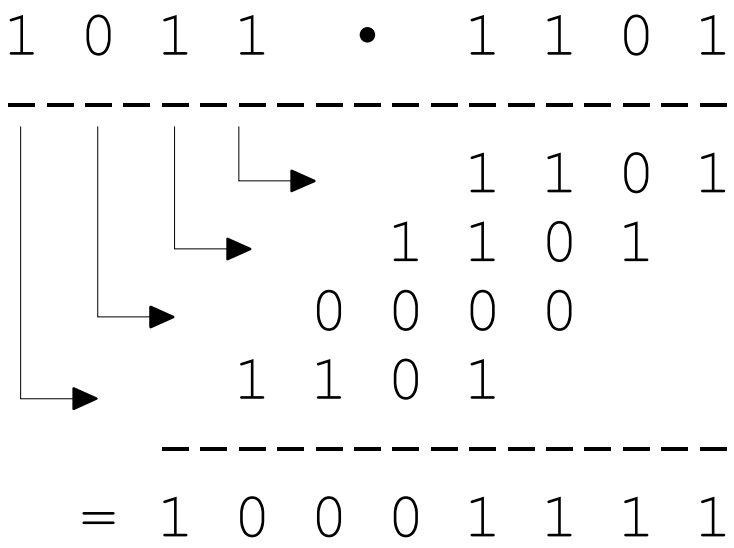
Barrel-Shifter

- Implementierung durch Multiplexer
  - Vorteil: Schneller, da Verschiebung um mehrere Bits in einem Zyklus möglich
  - Nachteil: (bedingt) höherer Schaltungsaufwand



Multiplizierer

- Methode: Äquivalent zum „schriftliche“ Multiplizieren mit Dezimalzahlen
  - Rückführung auf Shift, Multiplikation mit 1 oder 0 und abschließende Addition
- Beispiel:



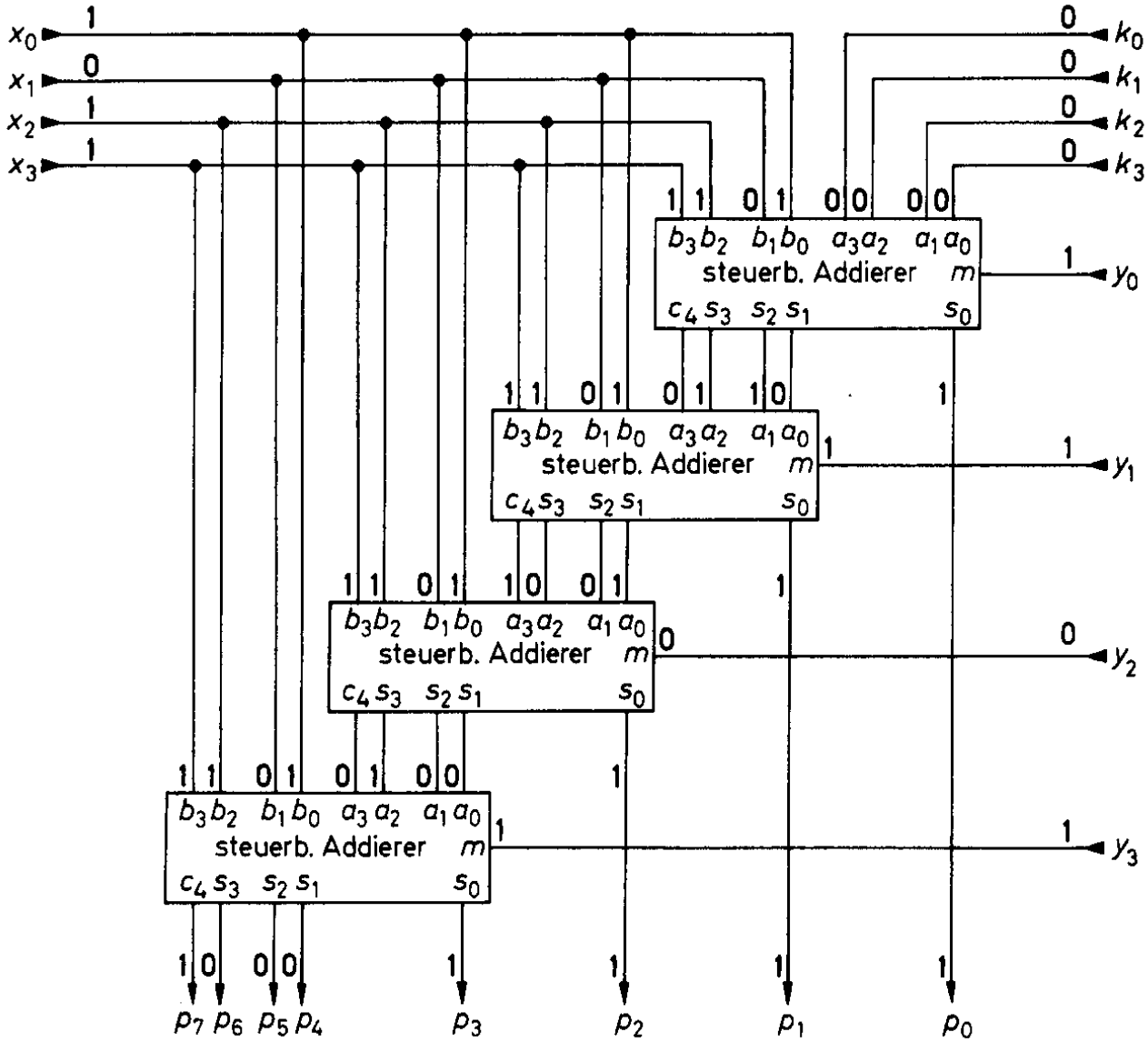
- Implementierung durch steuerbare, versetzt-parallele Addierer
  - Sehr schnell, aber aufwändig
- Implementierung durch einen Addierer, einen Shifter und eine Ablaufsteuerung
  - Langsam, aber weniger Gatter

Multiplizierer

- 4-Bit Multiplizierer
  - Hier: steuerbare versetzt parallele Addierer
  - Ausgang:

    $P = Y \bullet X + K$
  - Beispiel:

    11•13 = 143



Dividierer

- Methode: Ähnlich dem „schriftlichen“ Dividieren
  - Rückführung auf Shift, Addition und Multiplikation
  - *Dividend : Divisor = Quotient + Rest*
  - Dividend: *2n-Bit*, Divisor und Quotient: *n-Bit*
  - *Restoring-Methode*
- Division nur erlaubt, wenn
  - *Divisor ≠ 0* und *Divisor > höchstwertigen n Stellen des Dividenden*

Dividierer: Restoring-Methode

- Restoring-Methode:
  - 1. Dividend um eine Stelle nach links schieben
  - 2. Divisor von höchstwertigen *n* Stellen des Dividenden subtrahieren
  - 3. Wenn Ergebnis (= Rest-Dividend) negativ wird:
    Letzte Subtraktion rückgängig machen (Rückspeichern des Ergebnisses)
  - 4. Quotient um eine Stelle nach links schieben
  - 5. LSB im Quotienten setzen: *LSB = 0 wenn Rest-Dividend < 0*
    LSB = 1, wenn Rest-Dividend ≥ 0
  - 6. Wiederhole Schritte 1 bis 5 für alle *n* Stellen
  - 7. *Rest = höchstwertigen n Stellen des Dividenden*

Dividierer: Restoring-Methode

- Beispiel (4-Bit):

Dividend = 36_{dez} = 0010 0100

Divisor = 7_{dez} = 0111

(Komplement: -7_{dez} = 1001)

Quotient = 0101 = 5_{dez}

Rest = 0001 = 1_{dez}

```
  0010 0100   = Dividend

  0100 1000   1. Linksshift des Dividenden
 +1001        Subtraktion des Divisors
 =1101        Rest < 0, daher: Quotient = 0000

  0100 1000   Restauration

  1001 0000   2. Linksshift des Dividenden
 +1001        Subtraktion des Divisors
 =0010 0000   Rest ≥ 0, daher: Quotient = 0001

  0100 0000   3. Linksshift des Dividenden
 +1001        Subtraktion des Divisors
 =1101 0000   Rest < 0, daher: Quotient = 0010

  0100 0000   Restauration

  1000 0000   4. Linksshift des Dividenden
 +1001        Subtraktion des Divisors
 =0001 0000   Rest ≥ 0, daher: Quotient = 0101
```

Dividierer

- Implementierung
  - Algorithmus: Restoring-Methode
  - Relativ aufwändig, hohe Latenzzeit = niedriger Datendurchsatz
  - Pipelining praktisch kaum anwendbar

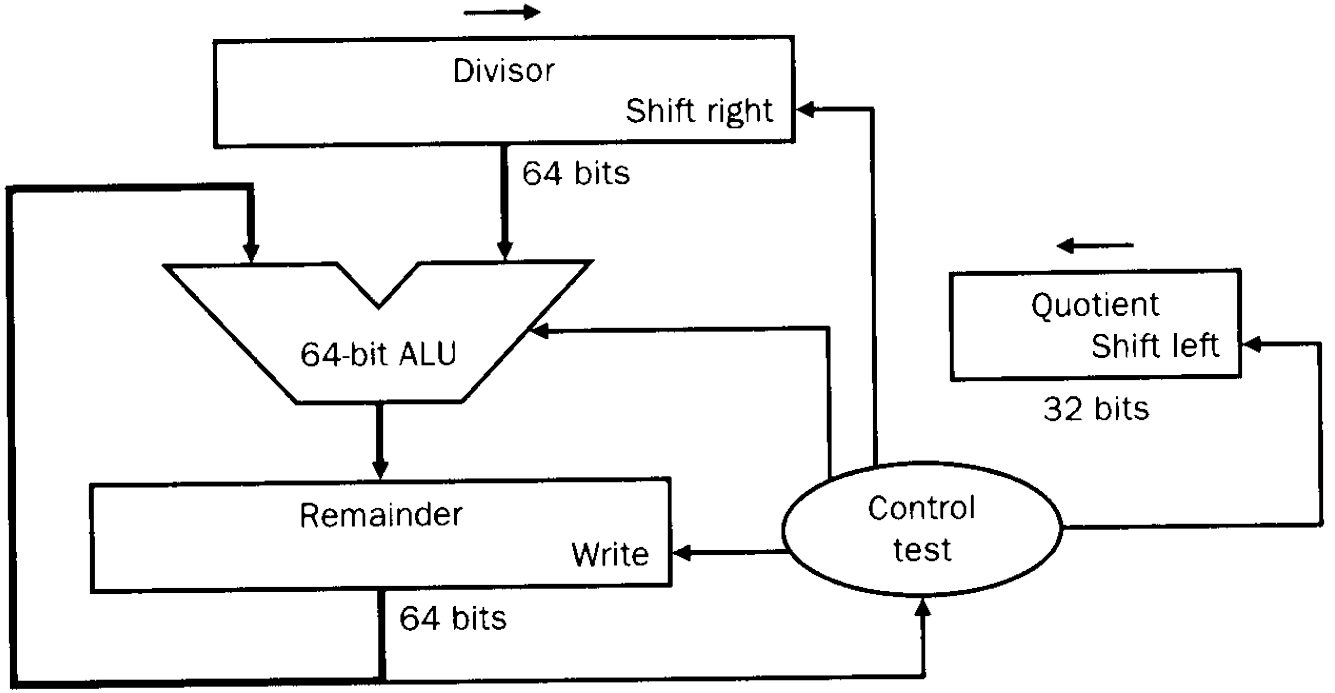


FIGURE 4.36 First version of the division hardware. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits. The 32-bit divisor starts in the left half of the Divisor register and is shifted right 1 bit on each step. The remainder is initialized with the dividend. Control decides when to shift the Divisor and Quotient registers and when to write the new value into the Remainder register.

Funktionsnetzwerke

- Funktionsnetzwerke zur Berechnung mathematischer Funktionen, z.B.
  - Trigonometrische Funktionen (Sinus, Cosinus)
  - Exponentialfunktionen
  - Logarithmus
- Berechnung durch iterative Näherungsverfahren und Rückführung auf die Grundrechenarten:
  - CORDIC-Algorithmen (COordinate Rotation DIgital Computer)
  - Darstellung der Operationen durch Vektorrotationen (vergl. Sinus und Cosinus bei Kreisbewegungen)
  - Im Wesentlichen werden nur noch Addition-, Shift- und Vergleichsoperationen benötigt
  - Vergleich Taschenrechner