ulm university universität ulm

Grundlagen der Rechnerarchitektur

[CS3100.010]

Wintersemester 2014/15

Heiko Falk

Institut für Eingebettete Systeme/Echtzeitsysteme
Ingenieurwissenschaften und Informatik
Universität Ulm

ulm university universität ulm

Kapitel 5

Rechnerarithmetik

Inhalte der Vorlesung

1. Einführung
2. Kombinatorische Logik
3. Sequentielle Logik
4. Technologische Grundlagen
5. **Rechnerarithmetik**
6. Grundlagen der Rechnerarchitektur
7. Speicher-Hardware
8. Ein-/Ausgabe

Inhalte des Kapitels (1)

5. Rechnerarithmetik

- Einleitung
- Binäre Addition
  - *Ripple Carry* Addierer
  - Serielles Addierwerk
  - *Carry-Look-Ahead*-Addierer
  - *Carry-Select*-Addierer
  - *Carry-Save*-Addierer
- Binäre Subtraktion
  - Zweierkomplement-Darstellung
  - Subtraktion im Zweierkomplement
- ...

Inhalte des Kapitels (2)

5. Rechnerarithmetik

- …
- Binäre Multiplikation
  - Vorzeichenlose Multiplikation
    - Serielles Schaltwerk
    - *Array*-Multiplizierer
  - Vorzeichenbehaftete Multiplikation
    - Booth-Algorithmus
    - Implementierung in VHDL
- Binäre Division
  - *Restoring*-Division
  - *Non-Restoring*-Division
- BCD-Arithmetik

Einordnung

Vorlesung „Grundlagen der Betriebssysteme“

- Repräsentation von Zahlen in der Computer-Hardware ist bekannt
  - Natürliche Zahlen: Darstellung zu einer Basis *b*
  - Ganze Zahlen: Zweierkomplementdarstellung
  - Reelle Zahlen: Festkomma- und Gleitkommadarstellung, IEEE 754
- Algorithmischer Ablauf der Grundrechenarten auf den verschiedenen Zahlendarstellungen ist ebenfalls bekannt

Unklar

- Wie können die Algorithmen für die Grundrechenarten effizient in Hardware umgesetzt werden?

☞ Wie sind Rechenwerke *(arithmetical logical units, ALUs)* intern aufgebaut?

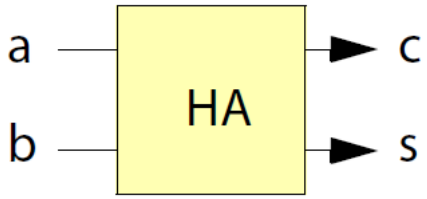
Erinnerung: Halbaddierer

Addition in erster (rechter) Spalte

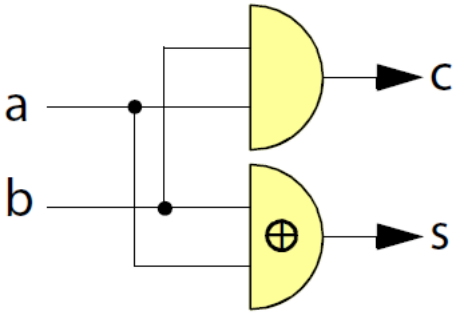
- Zwei Eingänge: erstes Bit von jeder Zahl
- Zwei Ausgänge: erstes Bit des Ergebnisses, *Carry*-Bit
- Wahrheitstabelle

a	*b*	*s*	*c*
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Blockschaltbild



- Gatterschaltung



$c = a * b$

$s = \overline{a} * b + a * \overline{b} = a \oplus b$ (XOR)

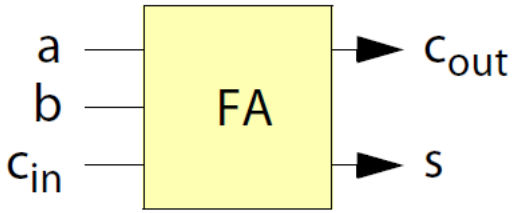
Erinnerung: Volladdierer (1)

Addition in anderen Spalte

- Drei Eingänge: je ein Bit der Summanden, und *Carry* von voriger Stelle
- Zwei Ausgänge: Summen-Bit des Ergebnisses, *Carry*-Bit
- Wahrheitstabelle

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

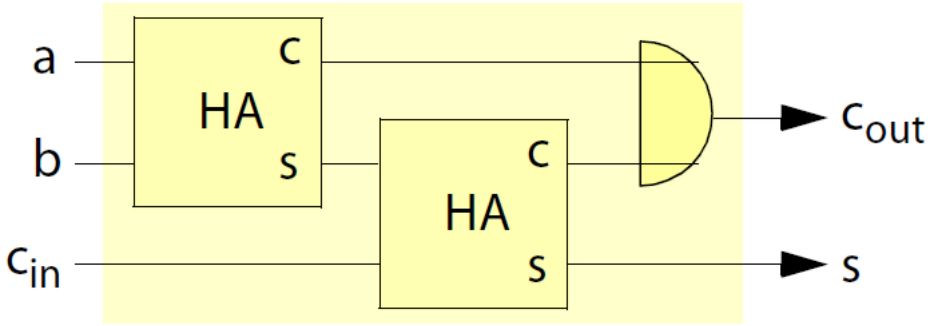
Blockschaltbild



Erinnerung: Volladdierer (2)

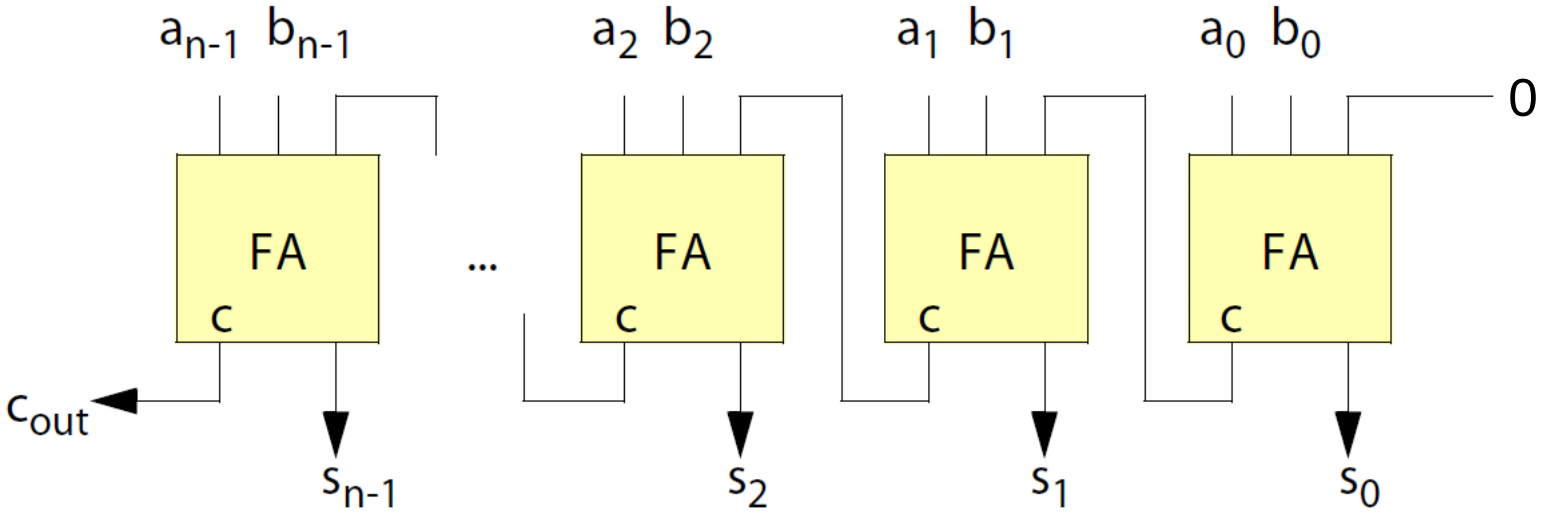
Schaltung

– Aufbau mit Halbaddierern



Paralleles Addierwerk

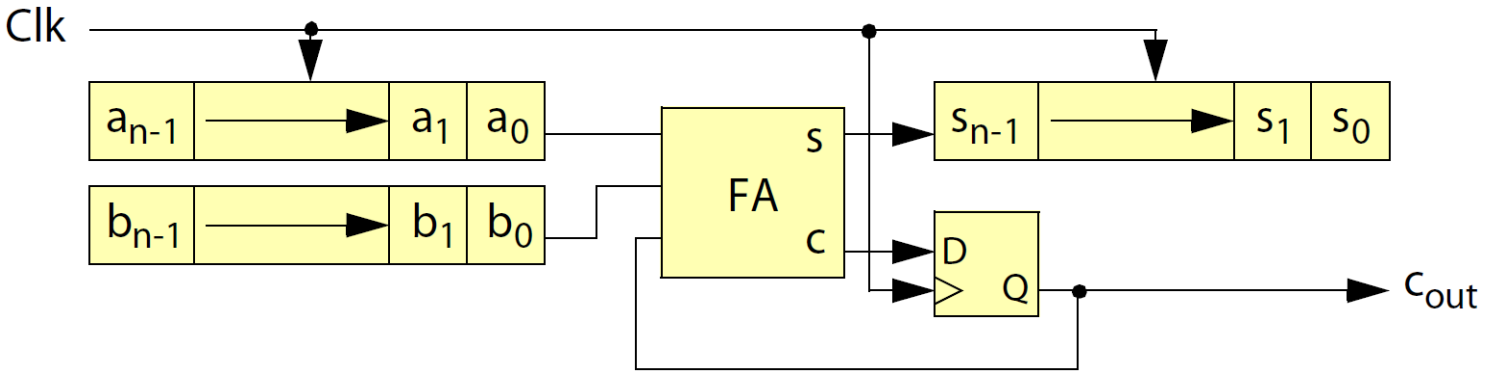
Schaltnetz zur Addition *n*-Bit langer Summanden



- Lange Gatterlaufzeit, bis Endergebnis stabil anliegt
  - Gatterlaufzeit: $t = 2n * \Delta t$
- *Ripple Carry Adder (RCA)*

Serielles Addierwerk

Synchrones Schaltwerk zur Addition *n*-Bit langer Summanden



- Schieberegister für jeden Summanden und für das Ergebnis
- Pro Takt wird genau eine Stelle der Summanden addiert
- Additionsergebnis liegt nach *n* Takten vor
- *Carry*-Flip-Flop muss zu Beginn mit 0 initialisiert werden!

Carry-Look-Ahead-Addierer (1)

Beschleunigung der Addition

- Vermeidung des sequentiellen Durchlaufs der Überträge
- Idee: parallele Berechnung aller Überträge für jede Stelle *i*

Es gilt für Stelle *i*

- $c_{i+1} = a_i * b_i + (a_i + b_i) * c_i$
  $= G_i + P_i * c_i$ mit
  - $G_i = a_i * b_i$ gibt an, ob Stelle *i Carry* generiert *(Generate)*
  - $P_i = a_i + b_i$ gibt an, ob Stelle *i Carry* weitergeben muss *(Propagate)*, falls vorherige Stelle *Carry* generiert oder weitergibt
- Schaltfunktionen für Überträge

$c_1 = G_0 + P_0 * c_0$

$c_2 = G_1 + P_1 * c_1 = G_1 + P_1 * G_0 + P_1 * P_0 * c_0$

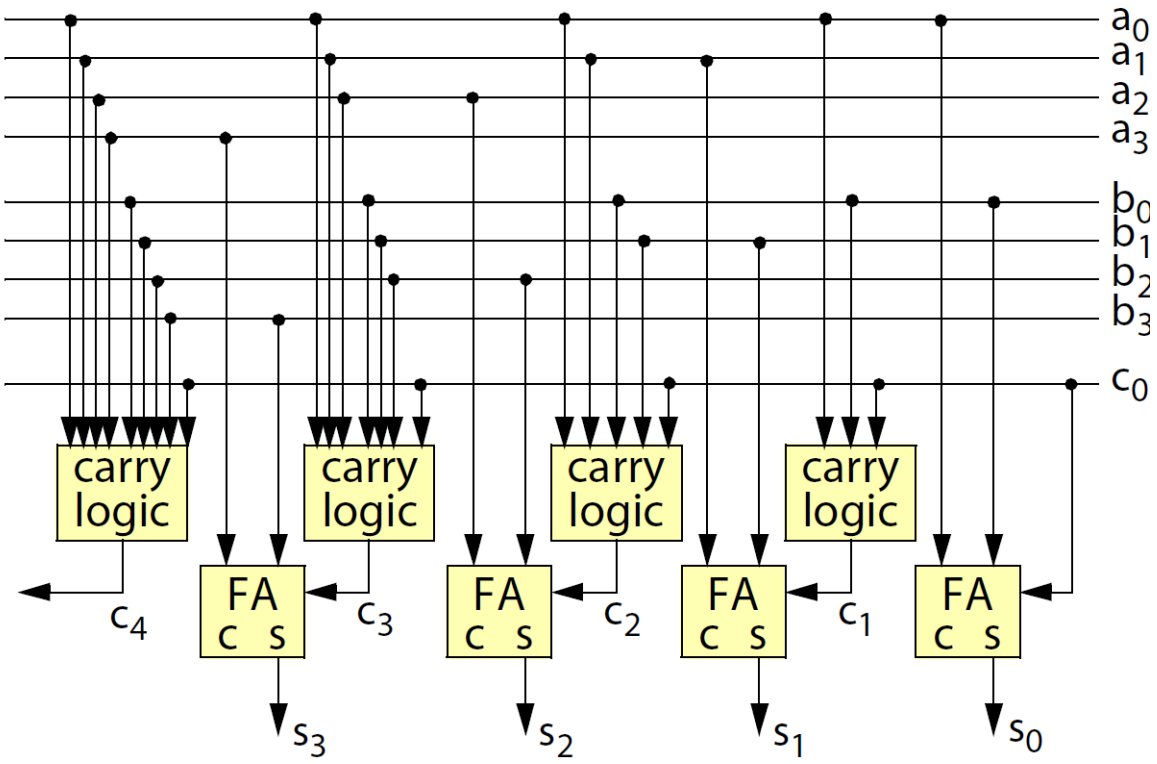
$c_3 = G_2 + P_2 * c_2 = G_2 + P_2 * G_1 + P_2 * P_1 * G_0 + P_2 * P_1 * P_0 * c_0$

...

Carry-Look-Ahead-Addierer (2)

Berechnung der Überträge mit maximal 2 Gatterlaufzeiten möglich

– Max. Anzahl der Gattereingänge hängt von der Breite des Addierers ab

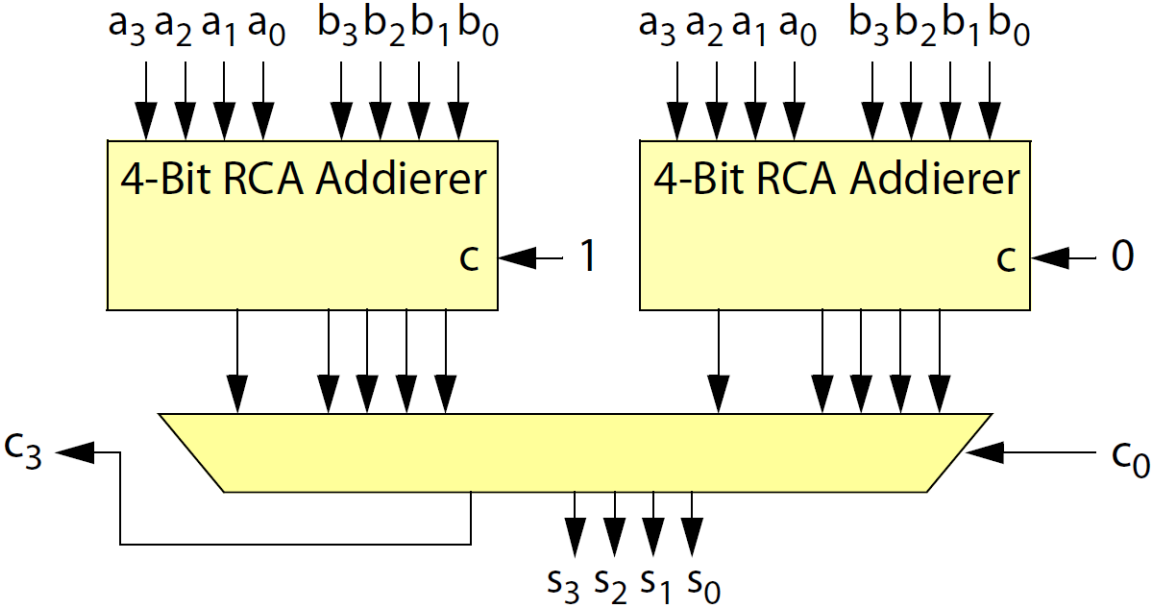


– Kaskadierung möglich
  – Pro CLA-Addierer nur $4\Delta t$ Verzögerung

Carry-Select-Addierer

Beschleunigung der Addition

– Idee: nicht auf das *Carry* des niederwertigen Blocks warten, sondern beide Ergebnisse berechnen und später selektieren



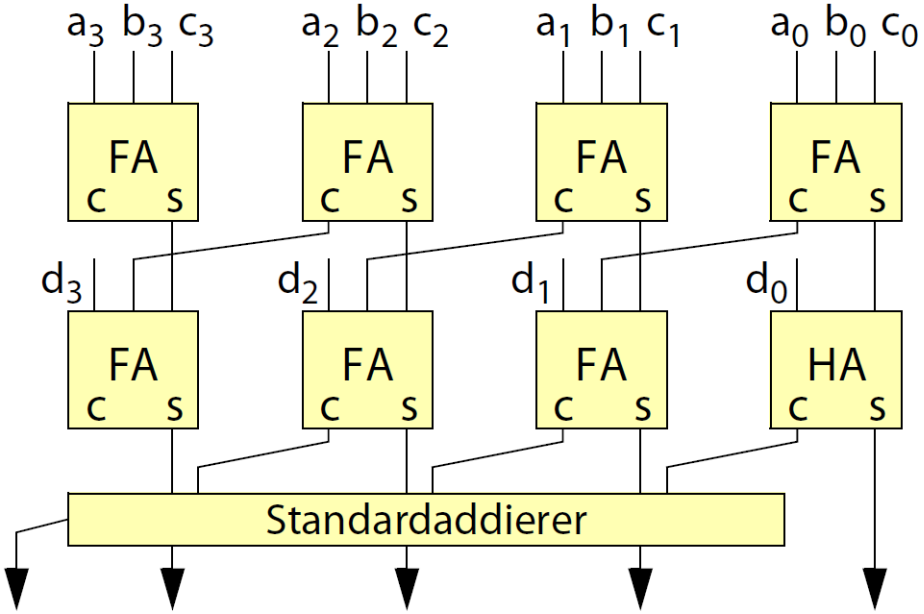
– Multiplexer selektiert Endergebnis

Carry-Save-Addierer

Mehr als zwei Summanden

– Überträge aus erster Addition werden in der nächsten Addition berücksichtigt
(keine Weitergabe der Überträge in der laufenden Addition)

– Beispiel: 4-Bit CSA für vier Summanden *a*, *b*, *c*, *d*



Zweierkomplement-Darstellung

Berechnung des Zweierkomplements einer Zahl *N* bei *n* Ziffern

- $C = 2^n - N$ bei n Ziffern/Bits
- Komplement C entspricht dem Wert $-N$

Darstellung positiver ganzer Zahlen

- Höchstwertiges Bit $z_{n-1} = 0$
- Andere Bits unbeschränkt
- Wert: $(z_{n-1}, ..., z_1, z_0)_2 = \sum_i z_i * 2^i$

Darstellung negativer ganzer Zahlen

- Höchstwertiges Bit $z_{n-1} = 1$
- Andere Bits unbeschränkt
- Wert: $(z_{n-1}, ..., z_1, z_0)_2 = -2^n + \sum_i z_i * 2^i$

Binäre Subtraktion

Subtrahierer kann ähnlich wie Addierer entwickelt werden

- Verwendung von Addierern zur Subtraktion
  - Idee: a – b = a + (-b)

Addition

- Einsatz von Standardaddierern für Zahlen im Zweierkomplement

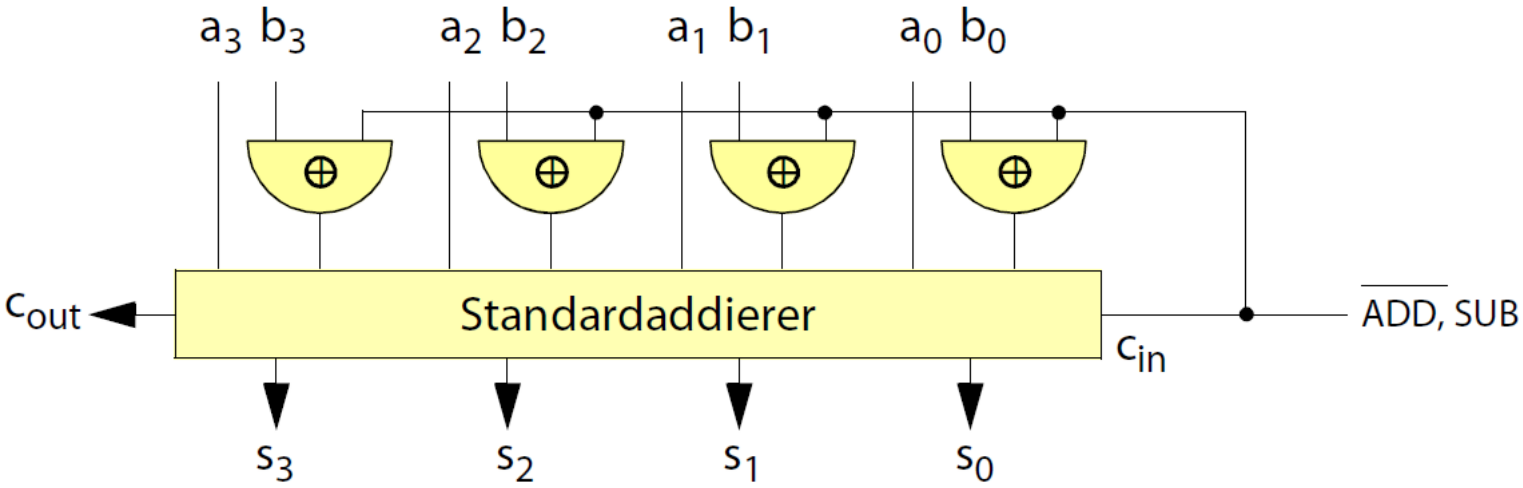
Subtraktion

- Vorherige Komplementbildung eines Summanden erfordert
  - Invertierung der Ziffern
  - Addition von 1

    kann durch gesetzten *Carry*-Eingang erzielt werden

Subtraktion im Zweierkomplement

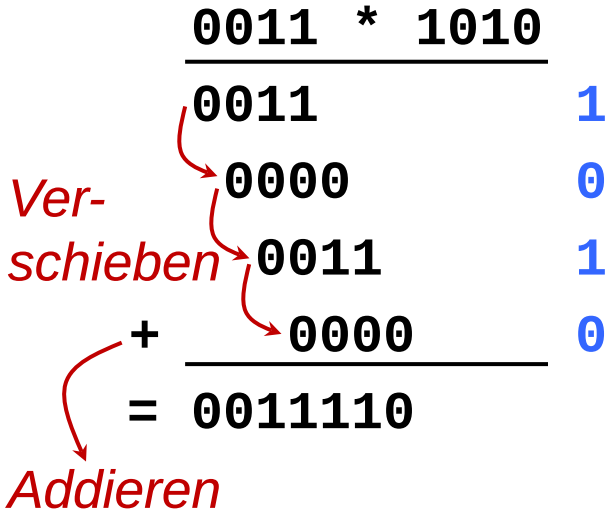
Addier- und Subtrahierwerk



- Beim Subtrahieren
  - Invertieren der b-Eingänge durch XOR-Gatter
  - Addieren von 1 durch gesetztes *Carry-in*
- Überlauferkennung: $c_{out} \neq c_{in}$
  - Bei Subtraktion ist gesetztes *Carry-out* der Normalfall

Binäre Multiplikation

Schriftliche Multiplikation / „Schulmethode“ auf positiven Binärzahlen



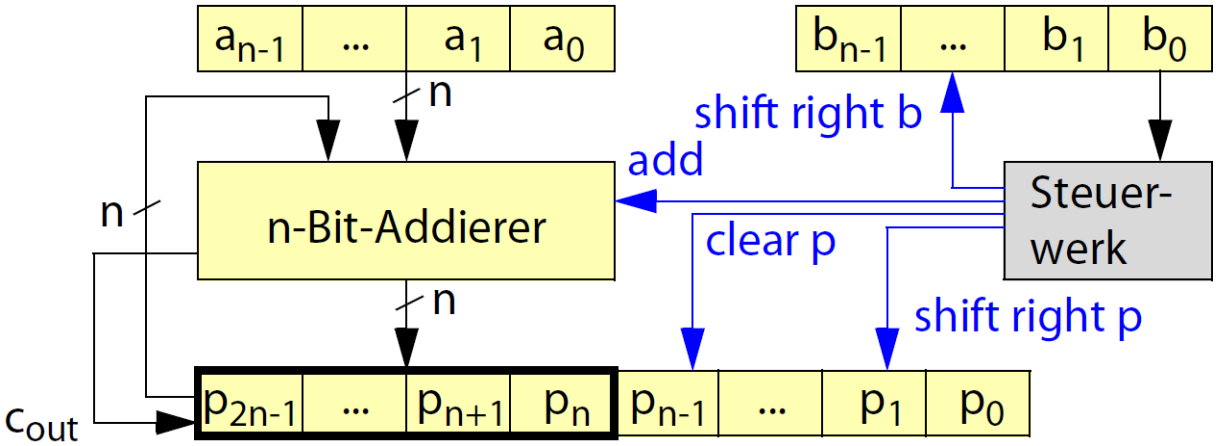
Kontrolle:
3 * 10 = 30
$30 = 11110_2$

Übertragung auf einen Computer

– Realisierung der Multiplikation durch Verschiebe-Operationen (Schieberegister) und einen Addierer

Vorzeichenlose Multiplikation (1)

Alternative A: Serielles Schaltwerk zur Multiplikation



- Lösche *p* *(clear p)*
- *n*-mal:
  - Ermittle b_0 *(shift right b)*
  - Addiere *a* auf $(p_{2n-1}, \ldots, p_{n+1}, p_n)_2$ oder nicht, je nach b_0 *(add)*
  - Verschiebe *p* einschließlich c_{out} der vorherigen Addition *(shift right p)*

Vorzeichenlose Multiplikation (2)

Alternative B: *Array*-Multiplizierer

– Schaltwerk für das schriftliche Multiplikationsschema

– Beispiel: $n = 4$

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	0
			a_3b_2	a_2b_2	a_1b_2	a_0b_2	0	0
+		a_3b_3	a_2b_3	a_1b_3	a_0b_3	0	0	0
	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

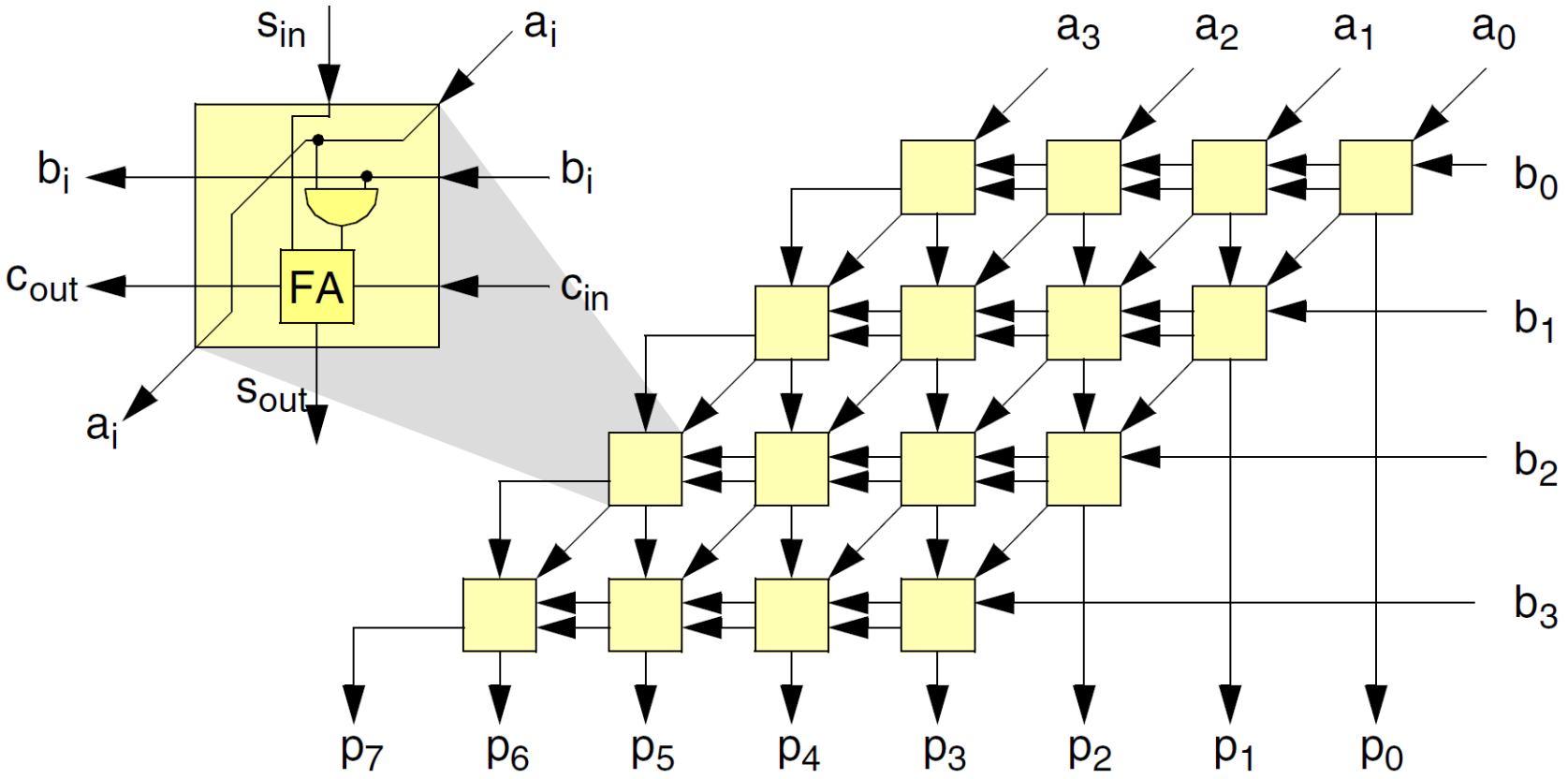
UND-Verknüfung

– Einsatz von *Carry-Save*-Addierern für die einzelnen Zeilen

Vorzeichenlose Multiplikation (3)

Alternative B: *Array*-Multiplizierer

– Schaltwerk für $n = 4$

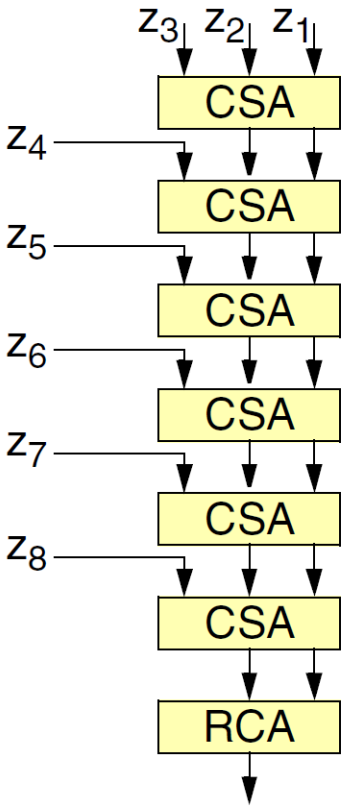


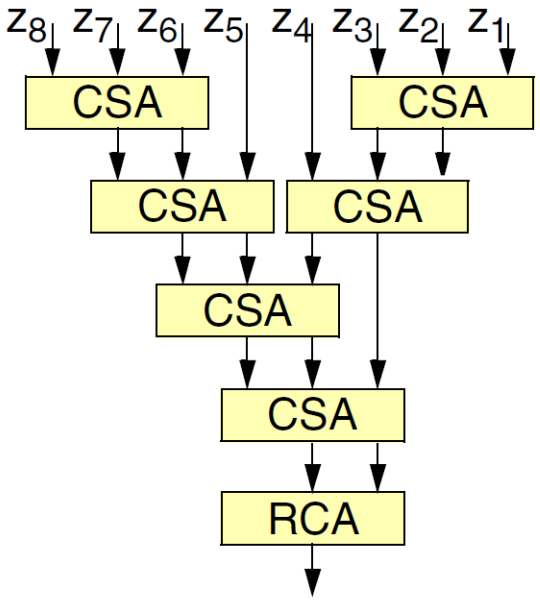
Vorzeichenlose Multiplikation (4)

Alternative C: Alternative B und baumförmige Addierer-Anordnung

– Baumförmige statt sequentielle Anordnung der *Carry-Save*-Addierer

– Beispiel: $n = 8$





– Vorteil: geringere Gatterlaufzeit

Vorzeichenlose Multiplikation (5)

Alternative D: Zweistufiges Schaltnetz für Multiplizierer

– Z.B. als PROM/ROM

☞ Siehe Kapitel 2, Folien 74ff.

– Schnellste Variante, aber extrem aufwändig

☞ $2^{2n} * 2n$ Bits in der Wertetabelle notwendig

Vorzeichenbehaftete Multiplikation: Booth-Algorithmus (1)

Wiederholung: Verfahren nach Booth

– Beispiel: -2 * 3 = -6 (für n = 4)

Fenster

$A * B$: **1110 * 0011**

$2n+1$ Bits breite Hilfsvariable R: **0000 0011 0**

Bits R_4, ..., R_1 mit B initialisiert, Rest 0

1. Fenster „10“: Subtrahiere A: **-1110**

   0010 0011 0

   Vorzeichenbehaftetes Schieben: **0001 0001 1**

2. Fenster „11“: nur Schieben: **0000 1000 1**

3. Fenster „01“: Addiere A: **+1110**

   1110 1000 1

   Vorzeichenbehaftetes Schieben: **1111 0100 0**

4. Fenster „00“: nur Schieben: **1111 1010 0**

Ergebnis: 11111010_2 = -6

Vorzeichenbehaftete Multiplikation: Booth-Algorithmus (2)

Booth-Algorithmus in Hardware-Beschreibungssprache VHDL

```
FUNCTION Booth(A, B : IN bit_vector) RETURN bit_vector IS
  CONSTANT n : natural := A'LENGTH;
  VARIABLE P : bit_vector(n-1 DOWNTO 0) := (OTHERS=>'0');
  VARIABLE Q : bit_vector(n DOWNTO 0) := (OTHERS=>'0');
BEGIN
  Q(n DOWNTO 1) := B;
  FOR i IN 0 TO n-1 LOOP
    CASE Q(1 DOWNTO 0) IS
      WHEN "10" => P := P - A;
      WHEN "01" => P := P + A;
      WHEN OTHERS =>                          -- keine Aktion
    END CASE;
    P & Q := sra(P & Q);              -- arithm. Rechts-Schieben
  END LOOP;
  RETURN P(n-1 DOWNTO 0) & Q(n DOWNTO 1);
END Booth;
```

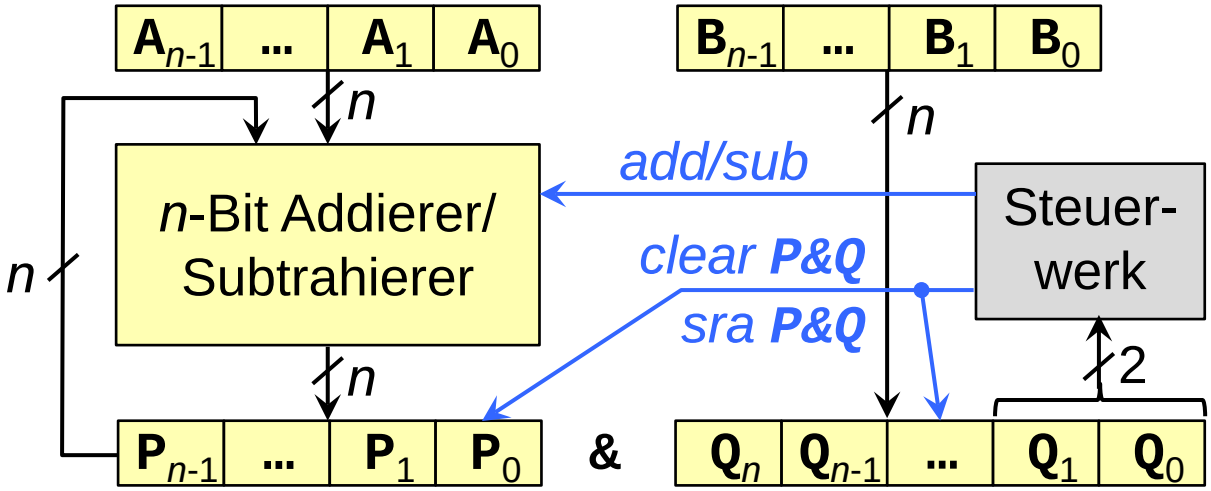
Vorzeichenbehaftete Multiplikation: Booth-Algorithmus (2)

Erklärungen zum VHDL-Code

- Eingabe sind Bit-Vektoren **A** und **B** im Zweierkomplement, Ausgabe ist wiederum ein Bit-Vektor.
- Voraussetzung: Eingaben **A** und **B** enthalten gleich viele Bits
- Die $2n+1$ Bits breite Hilfsvariable *R* von Folie 25 wird im VHDL-Code repräsentiert durch zwei Hilfsvariablen **P** und **Q**.
  P stellt die höherwertigsten *n* Bits von *R* dar, **Q** die niederwertigsten $n+1$. Hilfsvariable *R* ergibt sich aus **P** konkateniert mit **Q** (`P & Q`).
- Das typische Booth-Fenster wird im VHDL-Code repräsentiert durch Bits 0 und 1 der Hilfsvariable **Q**: `Q(1 DOWNTO 0)`.
- Abhängig vom Fensterinhalt wird in den höherwertigsten Bits von *R* (d.h. auf **P** im VHDL-Code) subtrahiert oder addiert.
- In jeder Iteration wird *R* (d.h. `P & Q` in VHDL) arithmetisch um ein Bit nach rechts geschoben

Vorzeichenbehaftete Multiplikation: Booth-Algorithmus (3)

Serielles Schaltwerk zur Multiplikation

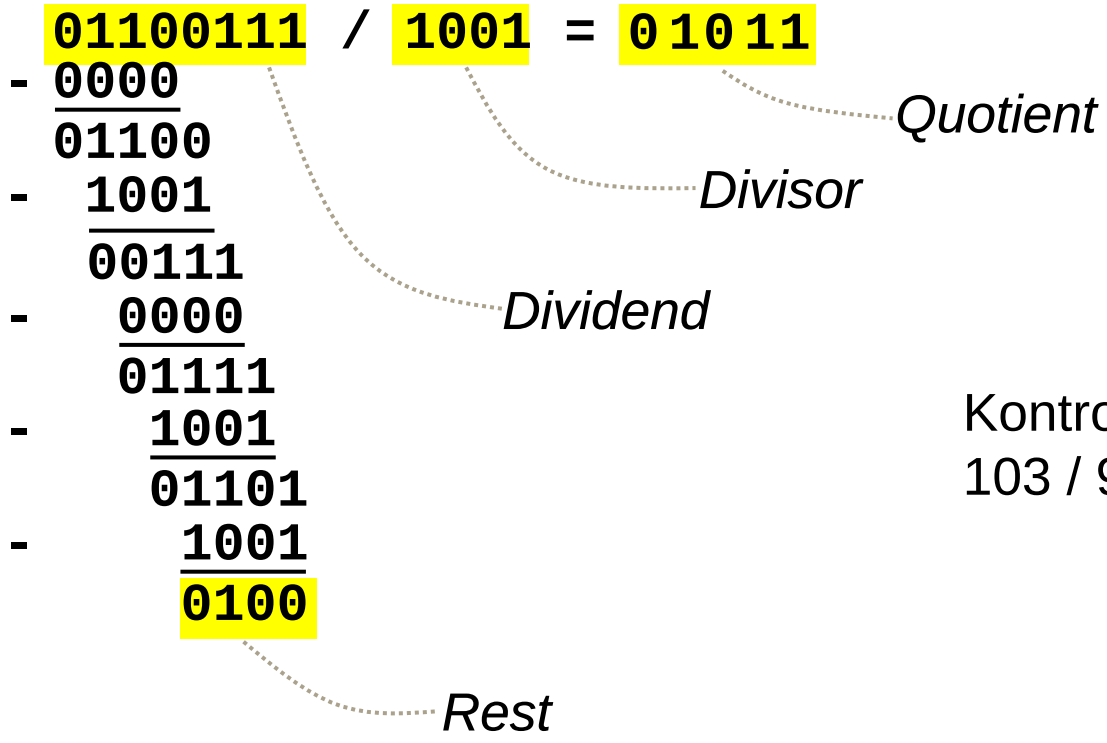


- Lösche **P** und **Q** *(clear **P&Q**)*
- *n*-mal:
  - Addiere bzw. subtrahiere **A** auf **P** oder nicht, je nach Q_1 und Q_0 *(add/sub)*
  - Verschiebe **P** und **Q** *(sra **P&Q**)*

Binäre Division

Schriftliche Division / „Schulmethode“

– Beispiel: 103 / 9 = ?



Kontrolle:
103 / 9 = 11 Rest 4

– Verfahren wird auch *Restoring*-Division genannt, da durch Korrektur ursprünglicher Dividend wiederhergestellt wird

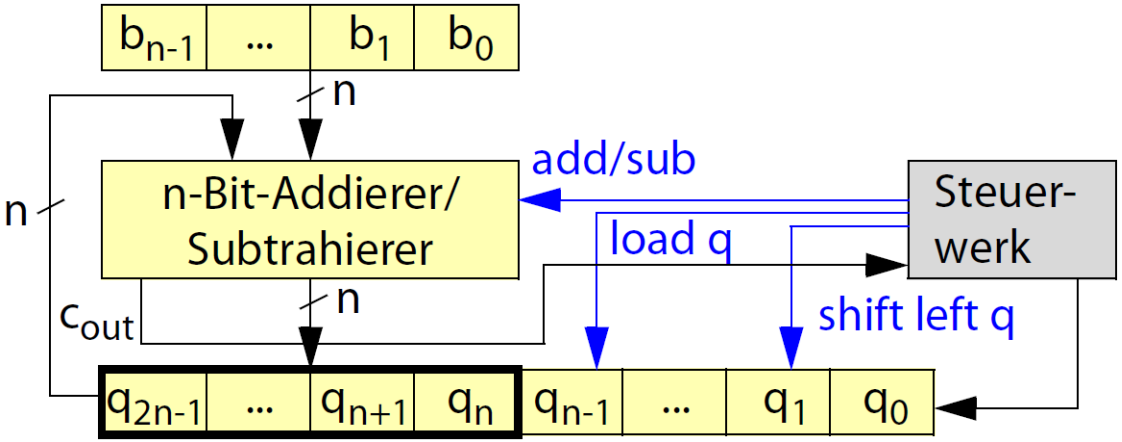
Restoring-Division (1)

Einsatz von Addition/Subtraktion und Schiebeoperationen

- In jedem Schritt testweise Subtraktion des skalierten Divisors b vom Dividenden a
  - $q_i = 1$ falls $a - b \geq 0$
  - $q_i = 0$ und Korrektur falls $a - b < 0$
- Nachholen signifikanter Ziffern zum Zwischenergebnis durch Schiebeoperation

Restoring-Division (2)

Serielles Dividierwerk für vorzeichenlose Zahlen



- Lade *q*: obere Hälfte = 0, untere Hälfte = *a* *(load q)*
- *n*-mal:
  - Schiebe *q* nach links *(shift left q)*
  - Subtrahiere *b* von oberer Hälfte von *q* *(sub)*:
    negativ: $q_0 = 0$ und addiere *b* zurück *(add)* / positiv: $q_0 = 1$
- Ergebnis: obere Hälfte *q* = Rest, untere Hälfte *q* = Quotient

Restoring-Division (3)

Vorzeichenbehaftete Division mit Zweierkomplement-Darstellung

- Verfahren im Prinzip identisch
- Jedoch:
  - Unterschiedliche Erkennung von Unterläufen
  - Propagierung des Vorzeichens in oberer Hälfte von *q* beim Laden

Non-Restoring-Division

Restoring-Division

– Divisor wird subtrahiert
– Falls Unterlauf (Ergebnis negativ): Divisor wird wieder addiert
– Im nächsten Durchlauf wird die Hälfte des Divisors wieder subtrahiert (wegen Links-*Shift* des Dividenden vor der Subtraktion)

Idee der *Non-Restoring*-Division

– Bei Unterlauf wird stattdessen nur die Hälfte des Divisors addiert

☞ Ersparnis einer Addition bzw. Subtraktion

BCD-Arithmetik (1)

BCD-Code *(Binary Coded Decimals)*

– 4-Bit Darstellung von Dezimalzahlen im Rechner

Dar-stellung	Wert	Dar-stellung	Wert
0000	0	1000	8
0001	1	1001	9
0010	2	1010	verbotene Codes
0011	3	1011	
0100	4	1100	
0101	5	1101	
0110	6	1110	
0111	7	1111	

– Zahlendarstellung mit mehreren 4-Bit breiten Ziffern, z.B. $(0001\ 0011\ 1001)_{BCD}$ entspricht der Dezimalzahl 139_{10}

BCD-Arithmetik (2)

Vorteil

– Leichtes Umwandeln von Dezimalzahlen in BCD-codierte Zahlen

Rechnen mit BCD-codierten Zahlen

– Spezielle Rechenwerke
– Unterstützung in gängigen Prozessoren
  – Zumindest für Addition und Subtraktion

Nachteil

– Oft nicht alle Rechenoperationen in Hardware unterstützt
– Rechenoperation in Software ineffizient
– Verschwendung von Speicherplatz

Zusammenfassung

Binäre Grundrechenarten

- Addition und Subtraktion auf Grundlage von Volladdierern leicht realisierbar
- Addier-/Subtrahierwerke haben je nach Aufbau unterschiedlich lange Gatterlaufzeiten
- Multiplikation und Division sind im Vergleich zu Addition/Subtraktion komplexer: benötigen mehr Aufwand in Hardware und/oder Rechenzeit
- Multiplizierer/Dividierer sind in der Regel als serielle Rechenwerke auf Grundlage von Schieberegistern und Addierern realisiert