Algorithmen zur Integer-Multiplikation

- Multiplikation zweier n-Bit Zahlen ist zurückführbar auf wiederholte **bedingte Additionen und Schiebeoperationen**

  (in einfachen Prozessoren wird daher oft auf Multiplizierwerke verzichtet !)
- das Produkt p zweier **vorzeichenloser n-Bit Zahlen** a und b erfordert **$2n$ Bit**, Zahlenbereich von p: $0 \ldots 2^{2n} - 2^{n+1} + 1$
- einfacher Algorithmus:

```
p = 0
for i = 0 to n-1 {
   if (b_i = 1) p = p + a · 2^i
}
```

- Beispiel (für $n = 5$):

```
01010 · 01101
-------------
        01010
       00000
      01010
     01010
    00000
-------------
   0010000010
```


Algorithmen zur Multiplikation (Forts.)

- modifizierter Algorithmus:

```
p = 0
for i = 0 to n-1 {
   if (b_i = 1)
      (p_2n-1 ,..., p_n ) = (p_2n-1,..., p_n) + a
   shift right p by 1
}
```

Beispiel (für $n = 5$):

```
     01010    ·    01101
     -------------------
     00000
   + 01010             add
     -----
     01010
     001010            shift
     0001010           shift
   + 01010             add
     -----
     0110010
     00110010          shift
   + 01010             add
     -----
     10000010
     010000010         shift
 p = 0010000010        shift
```

- in der $2n$-Bit Variablen p werden n partielle Produkte akkumuliert; Rechtsschieben von p ersetzt die Multiplikation von a mit 2^i

Algorithmen zur Multiplikation (Forts.)

- Erweiterung für **vorzeichenbehaftete** n-Bit Zahlen a und b:

1) bei Kodierung durch Vorzeichen und Betrag:

- für das Produkt p werden $2n{-}1$ Bit benötigt!
- vorzeichenlose Multiplikation der $(n{-}1)$-Bit Beträge $|a|$ und $|b|$ ergibt $(2n-2)$-Bit Produkt $|p|$ mit $|p|$ aus $0 \ldots 2^{2n-2} - 2^n + 1$
- separate Generierung des korrekten Vorzeichenbits $p_{2n-2} = a_{n-1} \oplus b_{n-1}$

2) bei Kodierung im Einerkomplement:

- für das Produkt p werden $2n{-}1$ Bit benötigt!
- symmetrischer Zahlenbereich für p: $-2^{2n-2} + 2^n - 1 \ldots 2^{2n-2} - 2^n + 1$
- Addition von **Korrekturtermen** erforderlich, da Algorithmus für $a < 0$ oder $b < 0$ falsche Ergebnisse: liefert

$a \cdot -b = a \cdot (2^n - 1 - b) = a \cdot 2^n - a - a \cdot b$ **(statt $2^{2n} - 1 - a \cdot b$)**

$-a \cdot b = (2^n - 1 - a) \cdot b = b \cdot 2^n - b - a \cdot b$ **(statt $2^{2n} - 1 - a \cdot b$)**

$-a \cdot -b = (2^n - 1 - a) \cdot (2^n - 1 - b)$

$= 2^{2n} - 2^n(a + b + 2) + a \cdot b + a + b + 1$ **(statt $a \cdot b$)**

Algorithmen zur Multiplikation (Forts.)

3) bei Kodierung im Zweierkomplement:

- asymmetrischer Zahlenbereich für p: $-2^{2n-2} + 2^{n-1} \ldots 2^{2n-2}$
- für das Produkt p werden $2n$ Bit benötigt, wobei das Bit p_{2n-2} nur für einen einzelnen Produktwert relevant ist!
- Addition von **Korrekturtermen** erforderlich, da Algorithmus für $a < 0$ oder $b < 0$ falsche Ergebnisse liefert:

$a \cdot -b = a \cdot (2^n - b) = a \cdot 2^n - a \cdot b$ **(statt $2^{2n} - a \cdot b$)**

$-a \cdot b = (2^n - a) \cdot b = b \cdot 2^n - a \cdot b$ **(statt $2^{2n} - a \cdot b$)**

$-a \cdot -b = (2^n - a) \cdot (2^n - b) = 2^{2n} - a \cdot 2^n - b \cdot 2^n + a \cdot b$ **(statt $a \cdot b$)**

- Fallunterscheidung:

für $a \cdot -b$ wird $2^{2n} - a \cdot 2^n$ = **$2^n \cdot (2^n - a)$** zu p addiert

für $-a \cdot b$ wird $2^{2n} - b \cdot 2^n$ = **$2^n \cdot (2^n - b)$** zu p addiert

für $-a \cdot -b$ wird $2^{2n} - (a + b) \cdot 2^n$ = **$2^n \cdot (2^n - a - b)$** zu p addiert

- **Realisierung**: p wird mit Korrekturterm anstatt mit 0 initialisiert!

Beschleunigung der Multiplikation

- $n \times n$ Bit Multiplikation benötigt n Schritte, jeweils aus:
  - einer n-Bit Addition, z.B. Carry Ripple Addierer mit Zeit $(2n-1)\tau$
  - einer Schiebeoperation auf $2n$-Bit Wort
- Möglichkeiten der Beschleunigung:
  1) Beschleunigung der Addition durch **Einsatz schnellerer Addierer**
  2) **vorzeitige Terminierung** in Schritt i, wenn $b_{n-1} = b_{n-2} = \ldots = b_i = 0$
  3) Schieben über **Ketten aus Nullen oder Einsen** im Multiplikator
  4) **Analyse mehrerer Bits des Multiplikators** und Addition von entsprechenden Vielfachen des Multiplikanden in jedem Schritt
  5) Berücksichtigung des Übertrags aus Addition in Schritt i erst bei Addition in Schritt i+1 ($\Rightarrow$ ***„Carry Save“* Addition**)
  6) **parallele Addition** mehrerer partieller Produkte in jedem Schritt
- auch Kombinationen von 1) bis 6) üblich!

Schieben über Ketten aus Nullen oder Einsen

- wenn die k Multiplikator-Bitstellen $\boldsymbol{b_{i+k-1} = \ldots = b_{i+1} = b_i = 0}$ sind, kann in Schritt i das bisher akkumulierte Produkt p direkt um k Stellen nach rechts geschoben werden
- wenn die k Multiplikator-Bitstellen $\boldsymbol{b_{i+k-1} = \ldots = b_{i+1} = b_i = 1}$ sind, entspricht dies einer Multiplikation von a mit dem Term

  $2^{i+k-1} + 2^{i+k-2} + \ \ldots\ + 2^{i+1} + 2^i = 2^{i+k} - 2^i$

  somit können die k Additionen können ersetzt werden durch
  - in Schritt i: **Subtraktion von *a*** und Rechtsschieben von p um k
  - in Schritt i+k: **Addition von *a*** und Rechtsschieben von p um 1
- Nachteile:
  - Multiplikationszeit ist nicht mehr vorhersagbar, d.h. hängt vom Wert des Multiplikators b ab !
  - hoher Aufwand für Barrel-Shifter und für Analyse des Multiplikators!

Analyse mehrerer Bits des Multiplikators

- Idee:
  - Analyse von ***k* benachbarten Bitstellen b_{i+k-1} ... b_i** des Multiplikators b
  - **Addition des (b_{i+k-1} ... b_i)-fachen** von a zu p
  - Rechtsschieben von p um k Positionen
- Vorgehensweise („*multiplier scanning*“) für $k = 2$:

$b_{i+1}\ b_i$	*durchzuführende Operationen*
00	schiebe p um 2 Stellen nach rechts
01	**Addiere a zu p** und schiebe p um 2 Stellen nach rechts
10	**Addiere $2a$ zu p** und schiebe p um 2 Stellen nach rechts
11	**Addiere $3a$ zu p** und schiebe p um 2 Stellen nach rechts

- Bereitstellung aller Vielfachen von a durch Linksschieben und Addieren (z.B. $3a = 2a + a$, d.h. schiebe a um 1 nach links und addiere a)
- wird nur für $k \leq 3$ eingesetzt !
  (da für $k > 3$ der Aufwand für die Bereitstellung aller Vielfachen zu hoch ist)

Multiplikation nach Booth

- Idee:
  - Kombination der Analyse von zwei Multiplikatorbits und des Schiebens über Ketten aus Nullen oder Einsen
  - Verzicht auf Addition des Vielfachen von a innerhalb einer Kette aus Einsen im Multiplikator b, statt dessen Subtraktion von a bei Beginn einer Kette aus Einsen (10) und Addition von a am Kettenende (01)
  - realisierbar durch **überlappende** Analyse von zwei Bitstellen $b_i\ b_{i-1}$ und **Umkodierung** von b_i ($\Rightarrow$ 1 bei Addition, –1 bzw. $\overline{1}$ bei Subtraktion)
- Vorgehensweise („*multiplier recoding*“):

$b_i\ b_{i-1}$	*durchzuführende Operationen*	*recoding*
00	schiebe p um 1 Stelle nach rechts	0
01	**Addiere a zu p** und schiebe p um 1 Stelle nach rechts	1
10	**Subtrahiere a von p** und schiebe p um 1 Stelle nach rechts	$\overline{1}$
11	schiebe p um 1 Stelle nach rechts	0

Multiplikation nach Booth (Forts.)

- Ergänzung von $b_{-1} = 0$ erforderlich
- funktioniert auch bei im Zweierkomplement kodierten negativen Zahlen (ursprüngliches Ziel von Booth)!
- Beispiele:

1) $(10)_{10} \times (-13)_{10}$

```
01010 × 10011|0 ← b-1
--------------
  1111110110  ← 100110
  00001010    ← 100110
  110110      ← 100110
--------------
 11101111110  = (-130)10
 └> ignorieren!
```

2) $(-10)_{10} \times (13)_{10}$

```
10110 × 01101|0 ← b-1
--------------
  0000001010  ← 011010
  111110110   ← 011010
  00001010    ← 011010
  110110      ← 011010
--------------
 11101111110  = (-130)10
 └> ignorieren!
```

3) $(-10)_{10} \times (-13)_{10}$

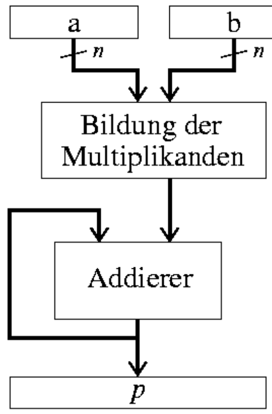
```
10110 × 10011|0 ← b-1
--------------
  0000001010  ← 100110
  11110110    ← 100110
  001010      ← 100110
--------------
 10010000010  = (130)10
 └> ignorieren!
```


Multiplikation nach Booth (Forts.)

- Verallgemeinerung für $k > 2$ Bits möglich:
  - Addition und Subtraktion des 1-fachen, 2-fachen, ... , $(k–1)$-fachen von a je Schritt, abhängig von Multiplikatorbits b_{i+k-2} ... , b_i, b_{i-1}
  - Rechtsschieben von p um $k–1$ Positionen je Schritt
  - Umkodieren des Multiplikators mit $b_i \in \{\overline{k-1}, \dots, \overline{1}, 0, 1, \dots, k-1\}$
  - typische Wahl: $k = 3$

Implementierung

- Möglichkeiten der Hardware-Implementierung einer $n \times n$ Bit Multiplikation:
  1) Verwendung eines ***n*-Bit Addierers** und eines 2*n*-Bit Schieberegisters
  2) Verwendung eines ***n*-Bit Carry-Save Addierers** und eines 2*n*-Bit Schieberegisters
  3) **parallele Addition** mit mehreren Carry Save Addierern
- allgemeiner Aufbau eines Multiplizierers:

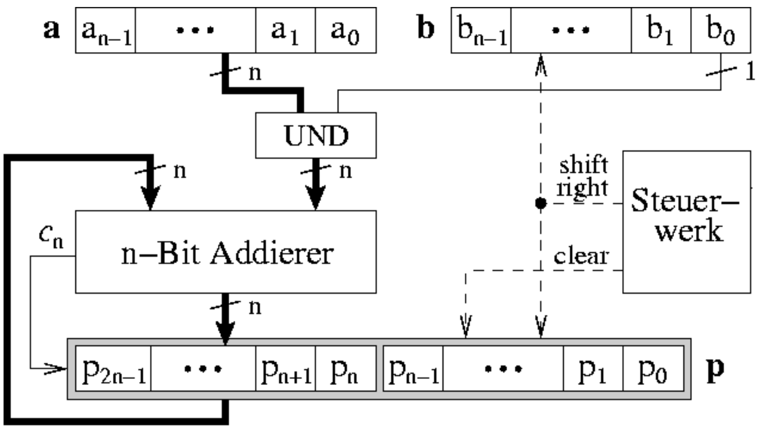


- Bemerkungen:
  - folgende Darstellung nur für einfachen Algorithmus, Verallgemeinerung für mehrere Multiplikatorbits bzw. Booth-Verfahren möglich
  - ein *n*-Bit (Schiebe-)Register kostet 8*n* CUs, Verzögerung für Laden bzw. Schieben ist τ

Sequentieller Multiplizierer

- direkte Realisierung des modifizierten Algorithmus in Hardware:

  n UND-Gatter
  n-Bit Addierer
  2*n*-Bit Schieberegister **p**

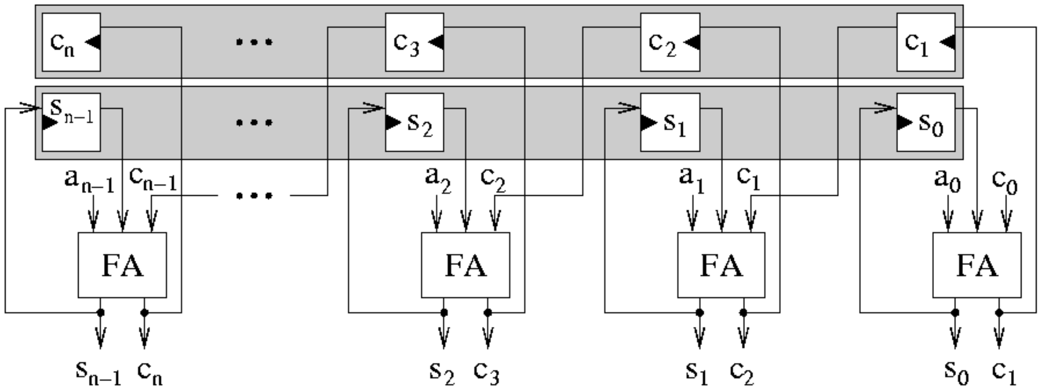


- Kosten (ohne **p**): $C_{Add} + 2n$ CUs, Zeit: $n \cdot (\Delta_{Add} + 3\tau)$
- Kosten und Zeit bei verschiedenen Addierern:

für $n = 32$:	Ripple	CLA	RCLA	Carry-Select
Kosten (CUs)	512	7456	1184	1056
Zeit (τ)	2112	224	416	288

Carry-Save Addierer (CSA)

- *Idee:* bei n aufeinander folgenden Additionen müssen die Carry-Signale nicht propagiert werden, sondern können erst bei der jeweils **folgenden** Addition berücksichtigt werden!

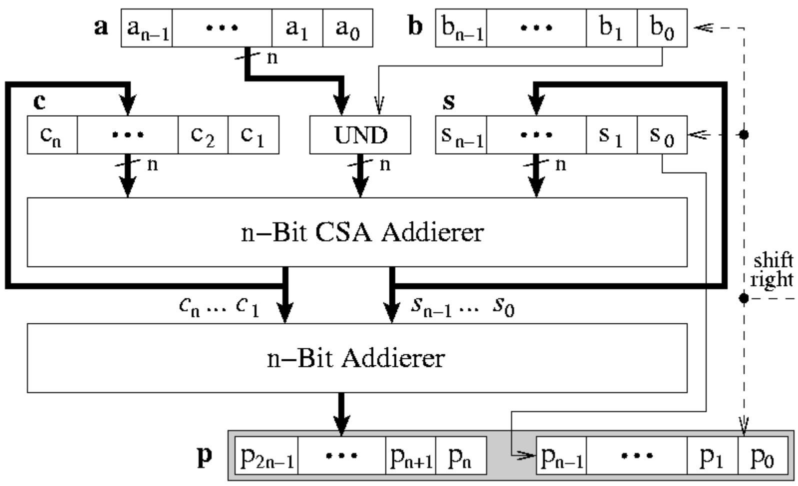


- in Schritt t wird $s_i(t) = s_i(t-1) \oplus a_i(t) \oplus c_i(t-1)$ berechnet
- nach n Schritten ist eine Addition der verbleibenden Überträge erforderlich (z.B. mit RCA oder CLA)

sequentieller Multiplizierer mit CSA

- Architektur eines CSA-basierten Multiplizierers:

  Anmerkungen:
  - nach jedem Schritt wird ***s*** um eine Stelle nach rechts geschoben
  - daher muß im CSA an der Bitposition i $c_{i+1}(t-1)$ anstatt $c_i(t-1)$ addiert werden!

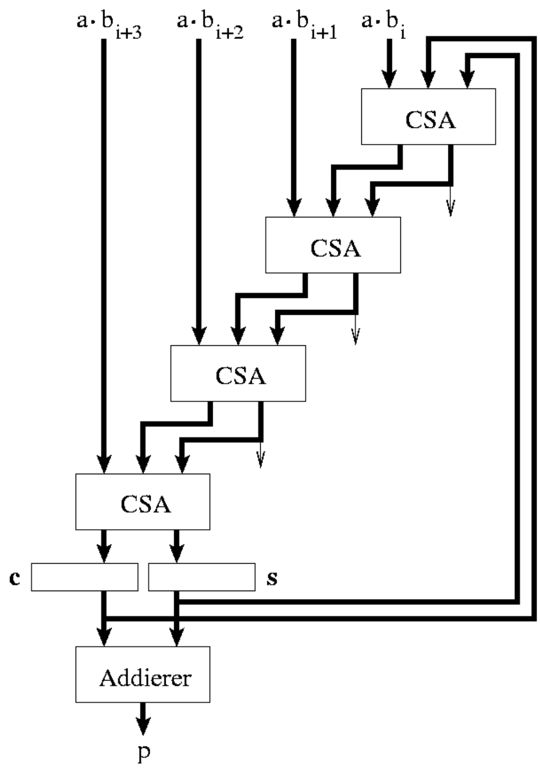


- Kosten (ohne **p**): $C_{Add} + (14+16+2)n$ CUs
- Zeit: $\Delta_{Add} + 6n\tau$

für n = 32:	Ripple	CLA	RCLA	Carry-Select
Kosten (CUs)	1472	8416	2144	2016
Zeit (τ)	255	196	202	198

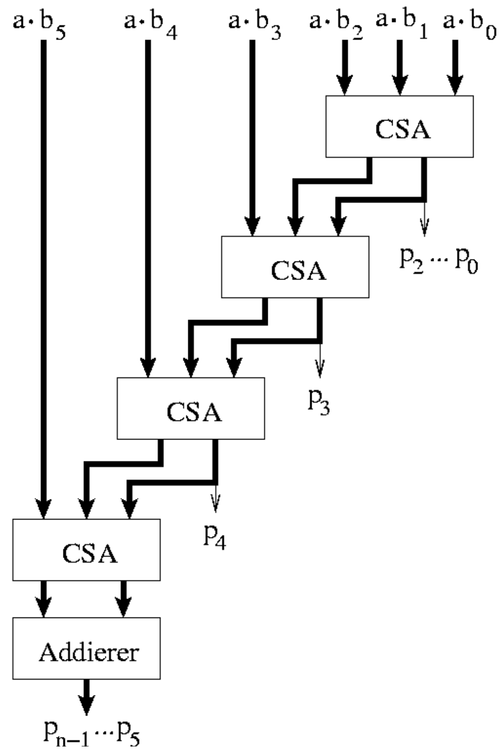
Paralleler Multiplizierer

- *Idee*: parallele Addition von k Teilprodukten $a \cdot b_{i+k-1}, \ldots, a \cdot b_i$ je Takt durch Verwendung von k CSA-Addierern
- Beispiel für $k = 4$ (vereinfacht):
- Summenausgänge jedes CSA müssen um eine Stelle nach rechts geschoben werden
- Aufwand:
  $C_{Add} + 14kn + 16n + 2kn$ CUs
- Zeit: $\Delta_{Add} + n/k \cdot (3k + 3)\tau$
- Probleme: Carry Ripple zwischen CSAs, Rückkopplung für c und s



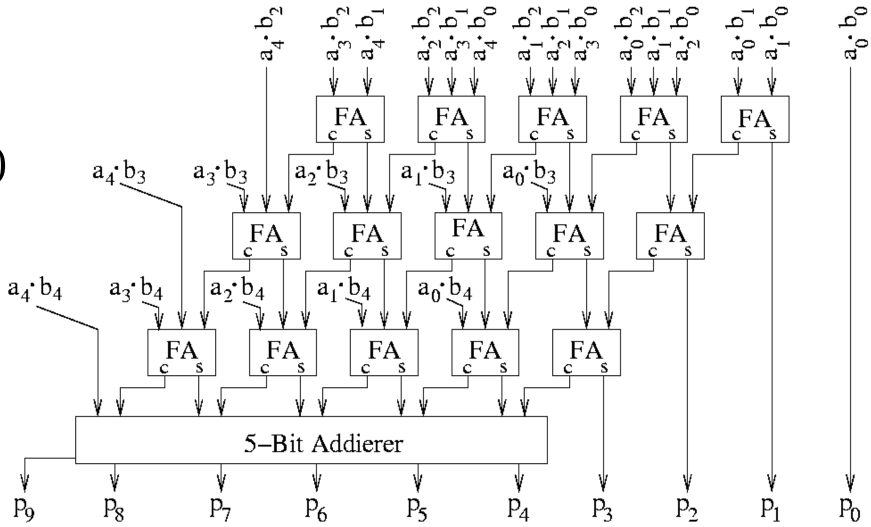
Paralleler Multiplizierer (Forts.)

- *Idee*: Auflösen der Rückkopplung und Aufbau einer **Addiererkette**
- Beispiel für $n = 6$ (vereinfacht):
- für jeden CSA ist eine geeignete Verschiebung der Eingangssignale erforderlich
- **rein kombinatorische Logik** !
- Aufwand:
  $C_{Add} + 14(n-2)(n-1) + 2n^2$ CUs
- Zeit: $\Delta_{Add} + \tau + 3(n-2)\tau$
  $= \Delta_{Add} + (3n-5)\tau$
- **Pipelining** prinzipiell möglich !



Paralleler Multiplizierer (Forts.)

- eine Addiererkette wird oft auch als **Feldmultiplizierer** („*array multiplier*“) bezeichnet
- Darstellung für $n = 5$:

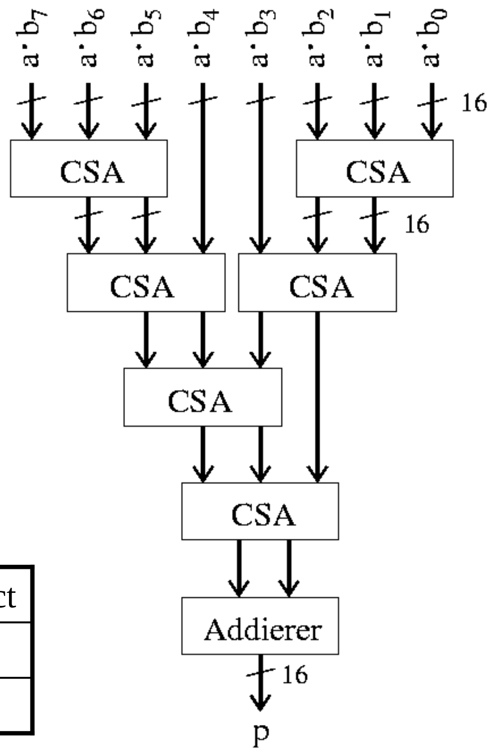


- Kosten und Zeit für verschiedene Addierer

für *n* = 32:	Ripple	CLA	RCLA	Carry-Select
Kosten (CUs)	15516	22524	16188	16060
Zeit (τ)	154	95	101	97

Paralleler Multiplizierer (Forts.)

- Beschleunigung möglich durch Verwendung eines **Addierbaums** („*Wallace tree*“)
- Beispiel für $n = 8$ (vereinfacht):
- Anzahl Stufen: $\lceil \log_{1.5} n/2 \rceil$
- Kosten: $C_{Add} + 16n^2 - 14$ CUs
- Zeit: $\Delta_{Add} + (3\lceil \log_{1.5} n/2 \rceil + 1)\tau$
- häufig eingesetzter Multiplizierer (oft kombiniert mit Pipelining)



für *n* = 32:	Ripple	CLA	RCLA	Carry-Select
Kosten (CUs)	17238	23762	17490	17362
Zeit (τ)	85	26	32	28