Digitaltechnik

4 Arithmetik

Revision 1.1

Diskretisierung

Einfache Addierer

Carry-Select-Addierer

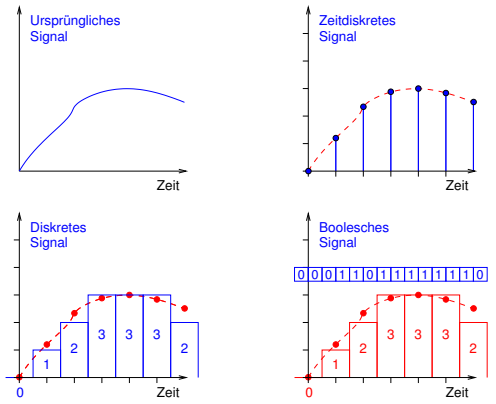
Conditional-Sum-Addierer

Conditional-Sum-Addierer

Carry-Look-Ahead Addierer

Multiplizierer

Diskretisierung



Diskretisierung

- Abstraktion kontinuierlicher physikalischer Größe zu endlich vielen Werten

$$W = \{l, l+1, \ldots, r\}$$

- Beispiele: Elektrische Spannung, Polarisationsrichtung, Wasserdruck
- kleinste Wertemenge besteht aus den Boole'schen Werten $\{0, 1\}$
- jede größere Wertemenge W lässt sich Boole'sch kodieren durch ein γ

$$\gamma : W \to 2^n \quad \text{mit} \quad n = \lceil \log_2 |W| \rceil, \quad \text{injektiv}$$

Aufzählungstypen

Abstrakte Aufzählungstypen wie

$$W = \{\text{Äpfel}, \text{Bananen}, \text{Birnen}\}$$

lassen sich durch injektive Abb. von

$$W \text{ nach } \{0, \ldots, 2^n - 1\} \text{ mit } n = \lceil \log_2 |W| \rceil$$

binär kodieren:

Äpfel	$\mapsto$	00_2
Bananen	$\mapsto$	01_2
Birnen	$\mapsto$	10_2
undef	$\mapsto$	11_2

Falls $|W|$ keine Zweierpotenz, muss man entweder gewisse Binärkombinationen unbelegt lassen oder mehrere mit demselben Wert assoziieren.

One-Hot-Encoding

m Werte werden durch m Bits kodiert

Beispiel:

Äpfel	$\mapsto$	001
Bananen	$\mapsto$	010
Birnen	$\mapsto$	100

✔ Kodierung vereinfacht sich immens.

- Einfach zu erkennende ungültige Kombinationen: genau ein Bit muss 1 sein.
- Vereinfacht auch die Schaltungen und macht Sie robust (mehr dazu später).

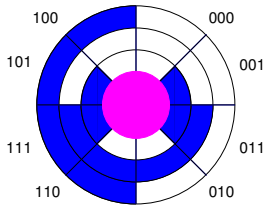
✘ Aber: Braucht exponentiell mehr Bits!

Gray-Code

Logarithmisch viele Bits in $|W|$, günstig für Zähler.

Aufeinanderfolgende Kombinationen unterscheiden sich an einer Bitposition

Motivation wie beim One-Hot-Encoding: robuste und einfache Schaltungen.



Binärkodierung für Zahlen

- geg. Wertebereich $W = \{l, \ldots, r\}$, zunächst $l = 0$ und $r = 2^n - 1$
- jedes $w \in W$ lässt sich als n-stellige Binärzahl darstellen:
  $0_{10} \mapsto 00_2$
  $1_{10} \mapsto 01_2$
  $2_{10} \mapsto 10_2$
  $3_{10} \mapsto 11_2$
- Wir schreiben $\langle d_{n-1} \ldots d_0 \rangle$ für die Zahl, die durch $d_{n-1} \ldots d_0$ repräsentiert wird:

$$\langle d_{n-1} \ldots d_0 \rangle := \sum_{i=0}^{n-1} d_i \cdot 2^i$$

- Beispiel: $\langle 10110 \rangle = 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 22$
  Man beachte das implizite Festlegen der Stelligkeit n.

Integers: Einerkomplement

Zusätzliches Vorzeichen-Bit s_n

Negative Zahlen $-x$ für $x \geq 0$ werden durch $2^{n+1} - 1 - x$ repräsentiert.

Die $(n+1)$-stellige Binärzahl $s_n d_{n-1} \ldots d_0$ im Einerkomplement kodiert

die positive Zahl	$\sum_{i=0}^{n-1} d_i \cdot 2^i$	falls $s_n = 0$
die negative Zahl	$-\sum_{i=0}^{n-1} (1 - d_i) \cdot 2^i$	falls $s_n = 1$

Trick: Bits umdrehen, als Integer negieren!

Beispiele:

00110	im 5-stelligen Einerkomplement kodiert	$4 + 2 = 6$
10110	im 5-stelligen Einerkomplement kodiert	$-(8 + 1) = -9$
01111	größte darstellbare Zahl	$2^n - 1 = 15$
10000	kleinste darstellbare Zahl	$-(2^n - 1) = -15$

Redundante Darstellungen der 0_{10} sind $0 \ldots 0$ und $1 \ldots 1$.

Integers: Zweierkomplement

Zusätzliches Vorzeichen-Bit s_n wie beim Einerkomplement

Negative Zahlen $-x$ für $x \geq 0$ werden durch $2^{n+1} - x$ repräsentiert.

Die $(n+1)$-stellige Binärzahl $s_n d_{n-1} \ldots d_0$ im Zweierkomplement kodiert

die positive Zahl	$\sum_{i=0}^{n-1} d_i \cdot 2^i$	falls $s_n = 0$
die negative Zahl	$-(1 + \sum_{i=0}^{n-1} (1 - d_i) \cdot 2^i)$	falls $s_n = 1$

Trick: Bits umdrehen, Eins hinzuzählen, als Integer negieren!

Beispiele:

00110	im 5-stelligen Zweierkomplement kodiert	$4 + 2 = 6$
10110	im 5-stelligen Zweierkomplement kodiert	$-(1 + 8 + 1) = -10$
01111	größte darstellbare Zahl	$2^n - 1 = 15$
10000	kleinste darstellbare Zahl	$-2^n = -16$

Integers: Zweierkomplement

- Wir schreiben $[\![d_{n-1} \ldots d_0]\!]$ für die Zahl, die im Zweierkomplement durch $d_{n-1} \ldots d_0$ kodiert wird
- Lemma: $2^n = 1 + \sum_{i=0}^{n-1} 1 \cdot 2^i$
- Definition ohne Fallunterscheidung

$$[\![s_n d_{n-1} \ldots d_0]\!] = s_n \cdot -2^n + \sum_{i=0}^{n-1} d_i \cdot 2^i$$

Integers: Zweierkomplement

Beweis: Trivial für $s_n = 0$.

Für $s_n = 1$:

$$\begin{aligned}
[\![s_n d_{n-1} \ldots d_0]\!] &= 1 \cdot -2^n + \sum_{i=0}^{n-1} d_i \cdot 2^i \\
&= -(2^n - \sum_{i=0}^{n-1} d_i \cdot 2^i) \\
&= -(2^n + \sum_{i=0}^{n-1} -d_i \cdot 2^i) \\
&= -(1 + \sum_{i=0}^{n-1} 1 \cdot 2^i + \sum_{i=0}^{n-1} -d_i \cdot 2^i) \quad \text{(Lemma!)} \\
&= -\left(1 + \sum_{i=0}^{n-1} (1 \cdot 2^i - d_i \cdot 2^i)\right) \\
&= -(1 + \sum_{i=0}^{n-1} (1 - d_i) \cdot 2^i)
\end{aligned}$$

Festkommazahlen – Fixpoint Numbers

- Genauigkeit m der Nachkommastellen wird fixiert, z.B. $m = 2$ Binärstellen.
- Zurückrechnen wie Zweierkomplement, Multiplikation mit 2^{-m} zum Schluss.
- Umwandlung von Zahlen x in die Festkommadarstellung erfordert Runden:
  6.26_{10} bei zwei Nachkommastellen (1/4) wird zu $6.25_{10} = 110.01_2$
- Das Festkomma (oder der Punkt im Englischen) wird nicht gespeichert:

0	0	0	1	1	0	0	1

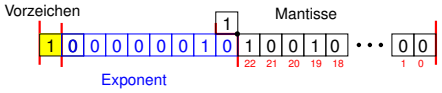
6.25 in einem Byte mit 2 Nachkommastellen

- Häufig verwendete Zahlendarstellung in Graphikhardware

Fließkommazahlen – Floating Point Numbers

Darstellen und Runden einer Zahl als $\boxed{(-1)^s \cdot 1.m \cdot 2^e}$

Separate Kodierung
des Exponenten e im Zweierkomplement,
des Vorzeichen-Bit s und
der positiven Mantisse m.



Einfache Genauigkeit (32 Bits):
1 Vorzeichen-Bit, 8 für Exponent, 23 für Mantisse.

IEEE Standard 754–1985: auch 64, 128 Bits.

Zusätzlich: Infinity und ungültige Zahlen (NaN = Not a Number).

Rechnen mit Binärzahlen

- Klar:

$$\underbrace{00000100}_{4} + \underbrace{00000110}_{6} = \underbrace{00001010}_{10}$$

- Aber was ist

$$\underbrace{11001000}_{200} + \underbrace{1100100}_{100} ?$$

- Exception werfen?
- Saturierung:

$$\underbrace{11001000}_{200} + \underbrace{1100100}_{100} = \underbrace{11111111}_{255}$$

Rechnen mit Binärzahlen

```
#include <stdio.h>

int main()
{
  char x;
  x=100;
  x=x+200;
  printf("%d\n", x);
}
```

Ausgabe: 44

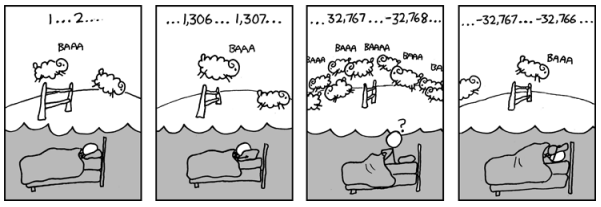
Rechnen mit Binärzahlen

- Überlauf: 300 = $\langle 100101100 \rangle$, also

$$\begin{array}{rl} & 11001000 \\ + & 01100100 \\ \hline = & 00101100 \quad = 44! \end{array}$$

- Überlauf entspricht modulo-Arithmetik

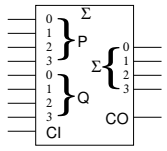
$$200 + 100 = 300 \bmod 256 = 44 \bmod 256$$



http://www.xkcd.org/571/

Addierer

- einfache arithmetische Basis-Operation
- Hardware Trade-Off: Zeit versus Platz
- Skalierbar durch Carry-Weiterleitung
  Carry In (CI) und Carry Out (CO)



IEEE Schaltsymbol
4-Bit Addierer

Additions-Algorithmus

a		1	1	0	1	1. Summand
b		0	1	0	1	2. Summand
c	1	1	0	1	0	Übertrag (engl. Carry)
s		0	0	1	0	Summe

1 = Carry Out 0 = Carry In

$$s_k \equiv (a_k \boxed{+} b_k \boxed{+} c_k) \bmod 2$$
$$c_{k+1} \equiv (a_k \boxed{+} b_k \boxed{+} c_k) \operatorname{div} 2$$

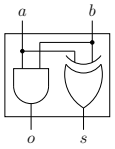
($\boxed{+}$ = Addition in den natürlichen Zahlen, nicht als $+$ = ODER)

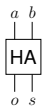
Halb-Addierer (Half Adder)

Addition von zwei Bits mit Übertragsgenerierung

$$s \equiv (a \boxed{+} b) \bmod 2 \equiv a \oplus b$$
$$o \equiv (a \boxed{+} b) \operatorname{div} 2 \equiv a \wedge b$$

Whitebox Model





Blackbox Model

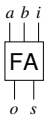
(Datenfluss von oben nach unten, Carry links, Summe rechts)

Volladdierer (Full Adder)

Addition von drei Bits mit Übertragsgenerierung

$$s \equiv (a \boxed{+} b \boxed{+} i) \bmod 2 \equiv a \oplus b \oplus i$$
$$o \equiv (a \boxed{+} b \boxed{+} i) \operatorname{div} 2 \equiv a \cdot b + a \cdot i + b \cdot i$$

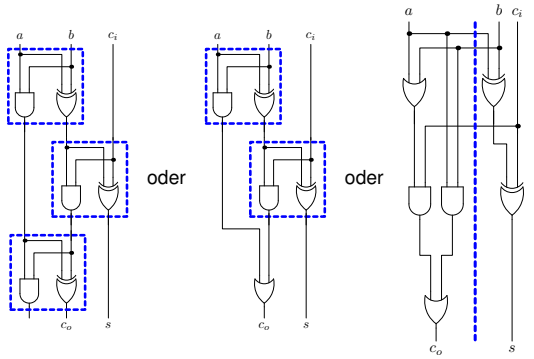
a	b	i	o	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



Blackbox Model

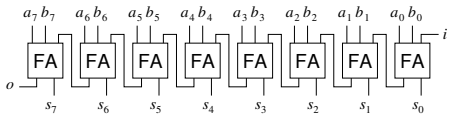
i = carry **i**n
o = carry **o**ut
s = **s**um

Volladdierer Whitebox Model



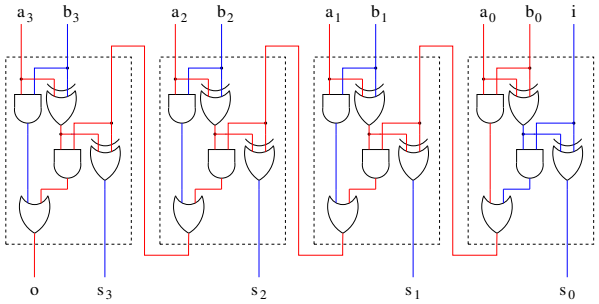
$$c_o = ab \oplus ((a \oplus b) \cdot c_i) \equiv ab + ((a \oplus b) \cdot c_i) \equiv ((a + b) \cdot c_i) + ab$$
$$s = (a \oplus b) \oplus c_i \equiv (a \oplus b) \oplus c_i \equiv (a \oplus b) \oplus c_i$$

Ripple-Carry Addierer



- Lässt sich einfach skalieren (Schema iterativ fortsetzen)
- **Platz:** $O(n)$
  n Volladdierer für Addition von zwei n-Bit Zahlen
- **Zeit:** $O(n)$
  längster Pfad (von i nach o) durchläuft n Volladdierer
  (bei einem 128 Bit Addierer kann das ziemlich lange dauern)

Carrypropagierung



Schritt 10

Korrektheit Addierer

Ein *Addierer* ist eine Schaltfunktion $add : \{0,1\}^n \times \{0,1\}^n \longrightarrow \{0,1\}^n$ mit

$$\langle x \rangle + \langle y \rangle = \langle add(x,y) \rangle \bmod 2^n$$

✔ Mit dieser Definition können wir die Korrektheit des Ripple-Carry Addierers per Induktion über n beweisen (siehe Buch).

Addition Zweierkomplement

Wie spezifizieren wir, was ein korrekter Addierer für vorzeichenbehaftete Zahlen tut?

Ein *Addierer* für Zahlen im Zweierkomplement ist eine Schaltfunktion $add^S : \{0,1\}^n \times \{0,1\}^n \longrightarrow \{0,1\}^n$ mit

$$[\![x]\!] + [\![y]\!] = [\![add^S(x,y)]\!] \bmod 2^n$$

Beispiel mit Überlauf:

$$\begin{array}{rcl} 100 & = & -4 \\ +111 & = & -1 \\ \hline 1\,011 & & -5 \\ \underbrace{}_{=3} & & \end{array} \qquad -5 = 3 \mod 8\,!$$

Zweierkomplement Addieren

Es sei $add : \{0,1\}^n \times \{0,1\}^n \longrightarrow \{0,1\}^n$ mit

$$\langle x \rangle + \langle y \rangle = \langle add(x,y) \rangle \bmod 2^n$$

(also ist add ein Addierer für Binärzahlen)

Behauptung

$$[\![x]\!] + [\![y]\!] = [\![add(x,y)]\!] \bmod 2^n$$

Beweis per Fallunterscheidung.

Konsequenz: $add = add^S$

✔ Wir können die selbe Schaltung für Binärzahlen und Zweierkomplement verwenden!

Subtrahieren

Idee: $a - b = a + (-b)$

Lemma:

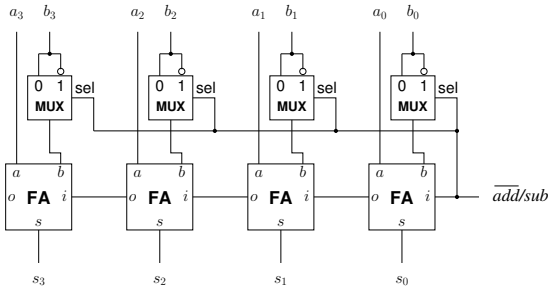
$$-[\![x_{n-1} \dots x_0]\!] = [\![\neg x_{n-1} \dots \neg x_0]\!] + 1 \bmod 2^n$$

Beweis per Fallunterscheidung.

Beispiele:

$$\begin{aligned} -0 &= -[\![000]\!] = [\![111]\!] + 1 = -1 + 1 \\ -1 &= -[\![001]\!] = [\![110]\!] + 1 = -2 + 1 \\ -(-1) &= -[\![111]\!] = [\![000]\!] + 1 = \ 0 + 1 \\ -(-4) &= -[\![100]\!] = [\![011]\!] + 1 = \ 3 + 1 \end{aligned}$$

Kombinierter 4-Bit Addierer/Subtrahierer



- A, B, S in 2er-Komplementdarstellung
- $\overline{add}/sub = 0$: $S = A + B$
- $\overline{add}/sub = 1$: $S = A - B$

Überlauferkennung

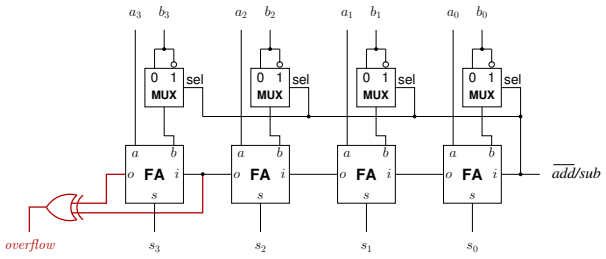
- Nützlich zur Fehlerbehandlung
- Intel x86: `into`, RISC: `ovf` Interrupt
- Binärzahlen: Klar, einfach das Carry Out

Theorem für das Zweierkomplement:
Es seien $c_0, \ldots, c_n$ die Carry Bits im Ripple-Carry Addierer.

$$[\![a]\!] + [\![b]\!] \notin \{-2^{n-1}, \ldots, 2^{n-1} - 1\} \quad \Longleftrightarrow \quad c_{n-1} \oplus c_n$$

Beweis: per Fallunterscheidung

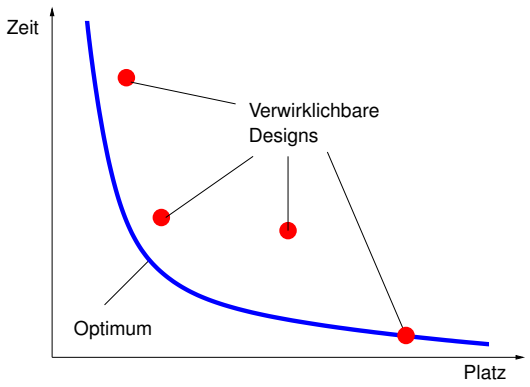
Kombinierter 4-Bit Addierer/Subtrahierer mit Überlauferkennung



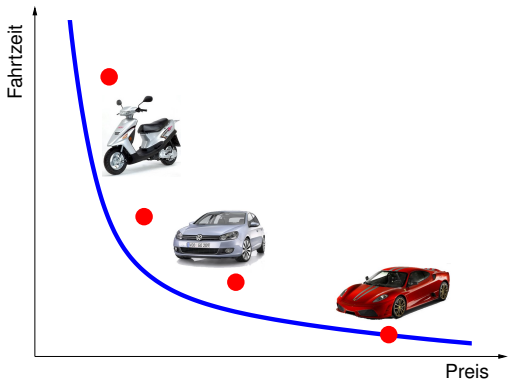
Addieren mit Verilog

- Selber zusammenbauen als kombinatorische Schaltung (Übung)
- Bibliothek verwenden (oft „handgemalt“)
- ✔ Einfach $a + b$ schreiben, wie in C/JAVA.
  Das Synthesetool wählt einen Addierer aus.

Zeit vs. Platz



Leistung vs. Kosten



Schnellere Addierer?

- Schnellere Addierer?
  (schneller als $O(n)$ beim n-Bit Ripple-Carry-Adder)
- Zeit vs. Platz:
  Platz begrenzt durch Chipfläche
  Zeit antiproportional zur erzielbaren Taktfrequenz (längster Pfad!)

Carry-Select-Addierer

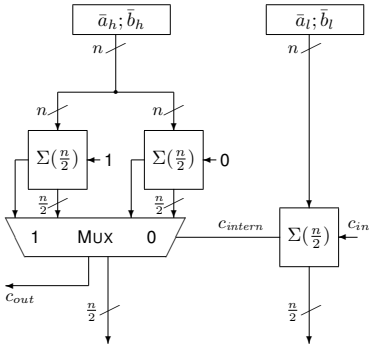
Wie vermeidet man die Carry-Propagation?

✔ Mehrere Lösungskandidaten vorausberechnen!

Wir teilen die zu addierenden Vektoren a, b in eine obere und untere Hälfte auf:

$$\begin{aligned} a_h &:= a[n-1 : n/2] \\ a_l &:= a[n/2-1 : 0] \end{aligned}$$

Carry-Select-Addierer



Carry-Select-Addierer: Kosten und Zeit

Zeit:

$$\begin{aligned} T(1) &= t_{FA} \\ T(n) &= t(\frac{n}{2}) + t_{MUX} \end{aligned}$$

$\Rightarrow$ Tiefe $O(\log_2(n))$

Kosten:

$$C(n) = 2 \cdot 3^{\log_2 n} + \sum_{i=0}^{\log_2(n)-1} 3^i \cdot \left(\frac{n}{2^{i-1}} + 4\right)$$

$\Rightarrow$ Kosten $O(n)$

(siehe auch Aufgabe 4.9 im Buch)

Conditional-Sum-Addierer

✘ Problem: Kosten! Wir brauchen 3 Sub-Addierer!

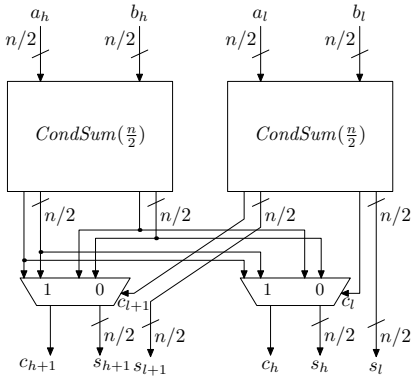
- Wir brauchen $a_h + b_h$ und $a_h + b_h + 1$

✔ Idee: Beide auf einmal berechnen!

$$\begin{aligned} s_h &= a_h + b_h \\ s_{h+1} &= a_h + b_h + 1 \end{aligned}$$

$$\begin{aligned} s_l &= a_l + b_l \\ s_{l+1} &= a_l + b_l + 1 \end{aligned}$$

Conditional-Sum-Addierer



Conditional-Sum-Addierer: Kosten und Zeit

Zeit:

$$\begin{aligned} T(1) &= 2 \cdot t_{FA} \\ T(n) &= t(\frac{n}{2}) + t_{MUX} \end{aligned}$$

$\Rightarrow$ Tiefe $O(\log_2(n))$

Kosten: $O(n)$

Carry-Look-Ahead Addierer

- Geht es noch schneller?
- Idee:
  Vorausberechnung des Carry
  Faktorisieren von Carry-Propagierung und Generierung

Carry-Generierung und Propagierung

Faktorisieren

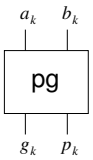
$$c_{k+1} \equiv a_k \cdot b_k + a_k \cdot c_k + b_k \cdot c_k \equiv \underbrace{a_k \cdot b_k}_{g_k} + \underbrace{(a_k + b_k)}_{p_k} \cdot c_k$$

$g_k \Leftrightarrow$ die Summanden-Bits an der k-ten Stelle **g**enerieren ein Carry
$p_k \Leftrightarrow$ die Summanden-Bits an der k-ten Stelle **p**ropagieren das Carry

Damit lässt sich wie folgt die Summe berechnen:

$$s_k \equiv a_k \oplus b_k \oplus c_k$$

Das pg Element



$$g_k \equiv a_k \cdot b_k$$
$$p_k \equiv a_k + b_k$$

Beobachtung: die Signale lassen sich auch aus dem Volladdierer abgreifen

Carry-Generierung und Propagierung

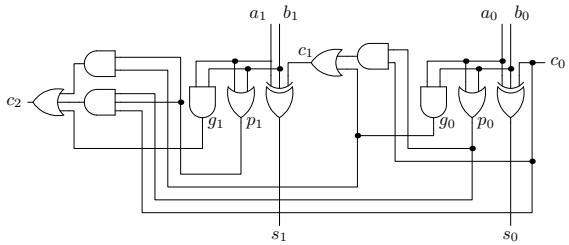
Beispiel für 4 Bits:

$$\begin{aligned}
c_1 &= g_0 \vee (p_0 \wedge c_0) \\
c_2 &= g_1 \vee (p_1 \wedge c_1) = g_1 \vee (g_0 \wedge p_1) \vee (c_0 \wedge p_0 \wedge p_1) \\
c_3 &= g_2 \vee (p_2 \wedge c_2) = g_2 \vee (g_1 \wedge p_2) \vee (g_0 \wedge p_1 \wedge p_2) \vee (c_0 \wedge p_0 \wedge p_1 \wedge p_2) \\
c_4 &= g_3 \vee (p_3 \wedge c_3) = g_3 \vee (g_2 \wedge p_3) \vee (g_1 \wedge p_2 \wedge p_3) \vee (g_0 \wedge p_1 \wedge p_2 \wedge p_3) \\
&\quad \vee (c_0 \wedge p_0 \wedge p_1 \wedge p_2 \wedge p_3)
\end{aligned}$$

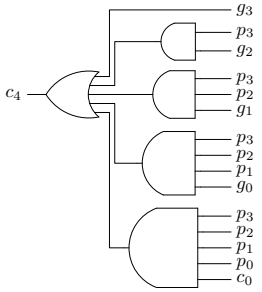
$\vdots$

Beobachtung: Das ist DNF!

Einfacher 2-Bit Carry Look-Ahead Addierer



Carry Out eines 4-Bit Carry Look-Ahead Addierers



Kosten & Geschwindigkeit

- **Platz:**
  insgesamt $O(n^2)$ Transistoren für n-Bit Addierer
  letztes ODER für c_k hat k Eingänge
  ODER mit k Eingängen braucht $O(k)$ Transistoren

$$\Sigma_{k=0}^{n} k \;=\; O(n^2)$$

- **Zeit:**
  $O(1)$
  4-stufige Logik (oder 5-stufig, wenn XOR nicht in der Basis)
  unabhängig von n
- die vielen Drähte erschweren Place & Route erheblich

Optimierter Carry Look-Ahead Addierer

- quadratischer Platz zu teuer für die Praxis,
  AND/OR Gatter mit beliebig vielen Eingängen gibt es nicht wirklich
- Ziel:
  weniger Platzverbrauch
  vergleichbarer Zeitaufwand
- Idee:
  Rekursive Generierung und Propagierung
  mit *Group-Propagate-* bzw. *Group-Generate-Signalen*
- damit erhält man ersten praktikablen Addierer

Group-Propagate und Group-Generate

- Idee: eine Gruppe von FAs kann propagieren und generieren
- Beispiel für 4 Bits:

$$PG = p_0 \wedge p_1 \wedge p_2 \wedge p_3$$
$$GG = g_3 \vee (g_2 \wedge p_3) \vee (g_1 \wedge p_3 \wedge p_2) \vee (g_0 \wedge p_3 \wedge p_2 \wedge p_1)$$

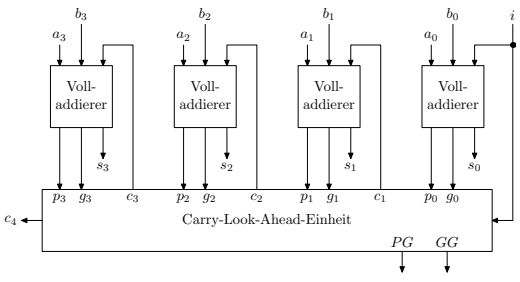
- Wir fügen diese Signale als neue Ausgabesignale des CLAs hinzu

Rekursionsansatz

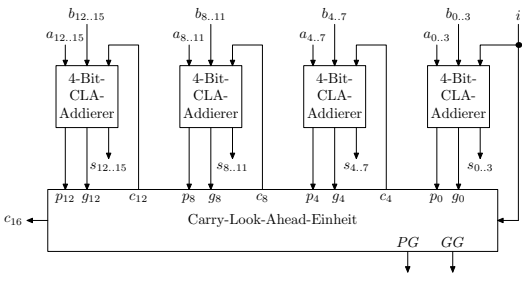
Rekursive Konstruktion von n-bit CLAs mit der
4-Bit „Carry-Look-Ahead Einheit“:

- Eingabe: 4x PG/GG, Carry-in
- Berechnet
  - 4 Carry Bits (siehe vorhergehende Folie)
  - Berechnet PG, GG (siehe vorhergehende Folie)

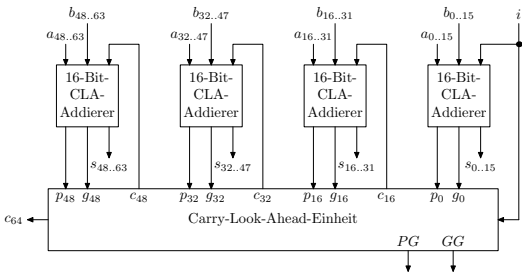
Ein 4-Bit Carry-Look-Ahead Addierer



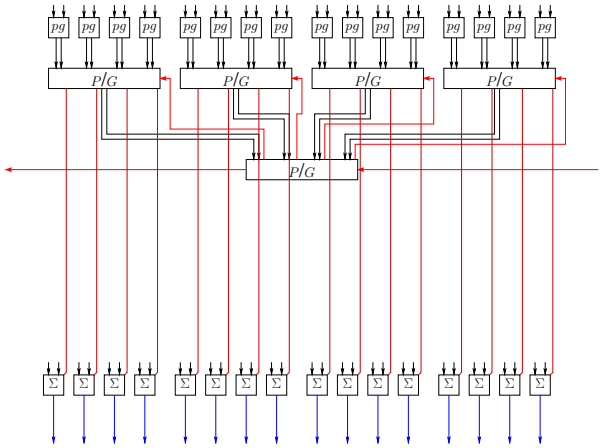
Ein 16-Bit Carry-Look-Ahead Addierer



Ein 64-Bit Carry-Look-Ahead Addierer



Ein 16-Bit Carry-Look-Ahead Addierer



Carry Look-Ahead Addierer: Längster Pfad

- Es sei $T(n)$ die *Tiefe* (Länge des längsten Pfades) im n-Bit CLA
- $T(1)$ ist die Tiefe des Volladdierers, t_{FA}
- Es sei t_{CLA} die Tiefe der „Carry-Look-Ahead Einheit“ (konstant!)
- Beobachtung: keine Pfade durch zwei unter-CLAs, da PG/GG *nicht* von dem Carry-in abhängt!

Also:

$$T(n) = T(\frac{n}{4}) + t_{CLA}$$

Carry Look-Ahead Addierer: Längster Pfad

- Geschlossene Form für $T(n) = T(\frac{n}{4}) + t_{CLA}$?
- $$\begin{array}{rcrcl} T(4) & = & T(1)+t_{CLA} & = & t_{FA}+\ t_{CLA} \\ T(16) & = & T(4)+t_{CLA} & = & t_{FA}+2{\cdot}t_{CLA} \\ T(64) & = & T(16)+t_{CLA} & = & t_{FA}+3{\cdot}t_{CLA} \\ T(256) & = & T(64)+t_{CLA} & = & t_{FA}+4{\cdot}t_{CLA} \end{array}$$
- Verallgemeinerung:

$$T(n) = t_{FA} + (\log_4 n) \cdot t_{CLA}$$

✔ Logarithmische Tiefe!

Carry Look-Ahead Addierer: Kosten

- Es sei c_{CLA} die Kosten der „Carry-Look-Ahead Einheit“, und c_{FA} die Kosten des Volladdierers
- Kosten für n-Bit CLA:

$$C(n) = 4 \cdot C(\frac{n}{4}) + c_{CLA}$$

- $$\begin{array}{rcrcl} C(4) & = & 4 \cdot C(1)+c_{CLA} & = & 4 \cdot c_{FA}+\ c_{CLA} \\ C(16) & = & 4 \cdot C(4)+c_{CLA} & = & 16 \cdot c_{FA}+\ 5{\cdot}c_{CLA} \\ C(64) & = & 4 \cdot C(16)+c_{CLA} & = & 64 \cdot c_{FA}+21{\cdot}c_{CLA} \\ C(256) & = & 4 \cdot C(64)+c_{CLA} & = & 256 \cdot c_{FA}+85{\cdot}c_{CLA} \end{array}$$

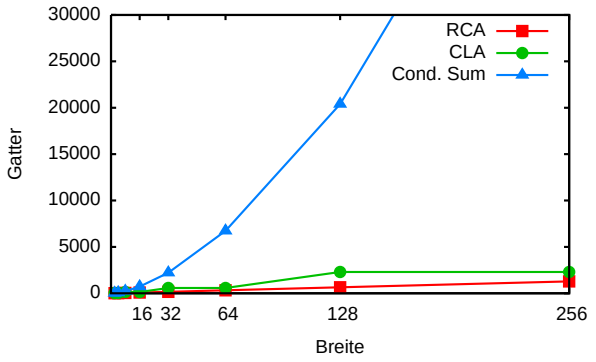
$\Longrightarrow$ $C(n) \approx O(n)$

Vergleich Addierer

Typ	Zeit	Platz	
		Gatter	Fläche
Carry Ripple Adder	$O(n)$	$O(n)$	$O(n)$
Carry Look-Ahead Adder	$O(\log n)$	$O(n)$	$O(n \cdot \log n)$
Carry Skip Adder Carry Select Adder	$O(\sqrt{n})$	$O(n)$	$O(n)$

- weitere Addierer mit dazwischenliegender Komplexität
- Anpassung an Technologie: manchmal ist Carry Ripple schnell
- Hybride Ansätze!

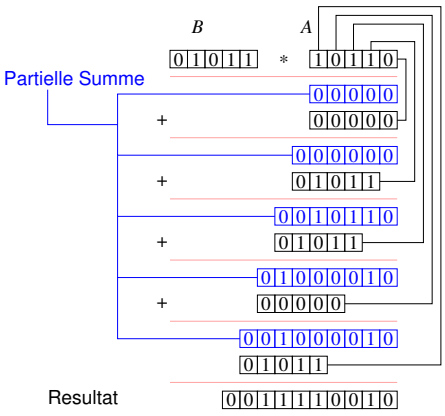
Vergleich Addierer



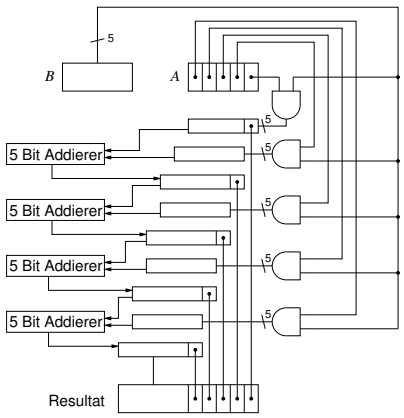
Multiplizierer

- Beispiel komplexer arithmetischer Schaltung
- Langsamer als Addition, braucht mehr Platz
- Übersicht:
  1. Sequentielle Multiplikation
  2. Kompakte kombinatorische Variante mit Carry-Save-Adders (CSA)
  3. Vorzeichenbehaftete Multiplikation mit Booth-Algorithmus

Schulalgorithmus zur Multiplikation



Einfacher Multiplizierer



Einfacher Multiplizierer

- $n \times n$ Multiplikation:
  Eingabe: zwei n-Bit Vektoren
  Ausgabe: $2 \cdot n$-Bit Vektor
- benötigt $n - 1$ seriell geschaltete Addierer
- Annahme: CLA Adder
- Zeit: $O(n \cdot \log n)$
- Platz: $O(n^2)$ Gatter, Fläche $O(n^2 \log n)$

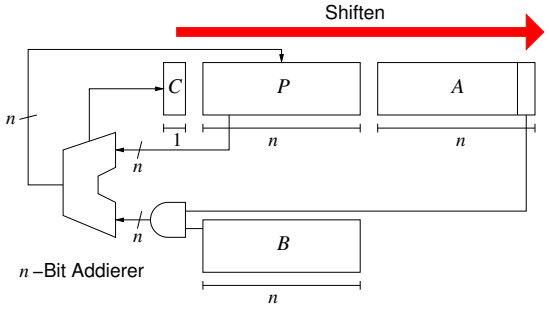
Sequentielle Version: Software

```
long long unsigned mult32(unsigned a, unsigned b)
{
  long long unsigned pa=a;
  unsigned i;

  for(i=0; i<32; i++) {
    if(pa&1)
      pa+=((unsigned long long)b)<<32;

    pa=pa>>1;
  }

  return pa;
}
```

Sequentielle Version: Software

- Benötigt eine Iteration pro Bit
- Pro Iteration:
  - eine Bitextrahierung
  - ein Test
  - zwei Shifts
- Q: Schleifeninvariante?

Sequentielle Version: Hardware



Sequentieller Algorithmus

1. Obere Hälfte P der partiellen Summe mit 0 initialisieren
2. Erster Operand ins B Register
3. Zweiter Operand ins A Register (untere Hälfte der partiellen Summe)
4. Für jeden der n Multiplikationsschritte:
   4.1 LSB von A gleich 1, dann addiere B zu P (ansonsten 0)
   4.2 Schiebe (C, P, A) nach rechts (C ist Carry des Addierers)
5. Result findet sich in (P, A)

Sequentielle Multiplikation von 7 und 5

	C	P	A	
		000	101	Schreibe $7 = 111_2$ nach B und $5 = 101_2$ nach A
+		111		$A_0 = 1$ also addiere $B = 111_2$
		0111	101	
→		011	110	Shiften (A_0 gebraucht, fällt also raus)
+		000		$A_1 = 0$ also addiere 0
		0011	110	
→		001	111	Shiften (A_1 gebraucht, fällt also raus)
+		111		$A_2 = 1$ also addiere $B = 111_2$
		1000	111	
→		100	011	Shiften, Resultat ist $100011_2 = 35$

Probleme mit Vorzeichen

- Multiplikation von -3 und -7 gibt natürlich 21
- 4-Bit Zweierkomplement: $-3 \mapsto 1101_2$, $-7 \mapsto 1001_2$
- als *unsigned* 4-Bit Zahlen sind das 13 bzw. 9
- Multiplikation von 13 und 9 ergibt 117
- 8-Bit Zweierkomplement: $21 \mapsto 00010101_2$, $117 \mapsto 01110101_2$
- Multiplikation gibt es mit oder ohne Vorzeichen (signed oder unsigned)!

Einfache Multiplikation mit Vorzeichen

1. Konvertierung der beiden Operanden in positive Zahlen
2. Speichern der ursprünglichen Vorzeichen
3. Unsigned-Multiplikation der konvertierten Zahlen
4. Berechnung des Resultats-Vorzeichen aus gespeicherten Vorzeichen

   (negativ gdw. ursprüngliche Operanden hatten komplementäres Vorzeichen)
5. eventuell Negation des Ergebnisses bei negativem Resultats-Vorzeichen

Booth Recoding

- Erste Beobachtung: PA ist auch signed!
- Verwende arithmetischen Shift statt logischem Shift
  (schiebe beim Shift Vorzeichen-Bit nach, statt dem Carry)
- Addiere $B \cdot (A_{i-1} - A_i)$ zu P (mit $A_{-1} = 0$):
  1. addiere B zu P wenn $A_i = 0$ und $A_{i-1} = 1$
  2. subtrahiere B von P wenn $A_i = 1$ und $A_{i-1} = 0$
  3. addiere 0 zu P wenn $A_i = A_{i-1}$

Booth Multiplikation von -6 und -5

P	A	
0000	1010	Schreibe $-6 = 1010_2$ nach A und $-5 = 1011$ nach B
0000	1010	$A_0 = A_{-1} = 0$ ergibt mit Regel 3 Addition von 0
0000	0101	Shiften
-1011		$A_1 = 1$, $A_0 = 0$ ergibt mit Regel 2 Subtraktion von B
$+0101$		Zweierkomplement von 1011_2 ist 0101_2
0101	0101	
0010	1010	Shiften
$+1011$		$A_2 = 0$, $A_1 = 1$ ergibt mit Regel 1 Addition von B
1101	1010	
1110	1101	Shiften (arithmetisch!)
-1011		$A_3 = 1$, $A_2 = 0$ ergibt mit Regel 2 Subtraktion von B
$+0101$		Zweierkomplement von 1011_2 ist 0101_2
0011	1101	Shiften (arithmetisch!)
0001	1110	Resultat ist $00011110_2 = 30$

Korrektheit Booth Multiplikation

Da jedes Mal

$$B \cdot (A_{i-1} - A_i)$$

zum partiellen Produkt addiert wird, erhält man die Teleskopsumme

$$\sum_{i=0}^{n-1} B \cdot (A_{i-1} - A_i) \cdot 2^i \equiv$$

$$B \cdot \left(-A_{n-1} \cdot 2^{n-1} + A_{n-2} \cdot 2^{n-2} + \cdots + A_1 2 + A_0\right) + B \cdot A_{-1} \equiv$$

$$B \cdot \left(-A_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} A_i \cdot 2^i\right)$$

Der Klammerausdruck ist genau der Integer-Wert einer n-Bit Zahl A im Zweierkomplement!

(z.B. $-3 = -0011_2 = 1101_2 = -2^3 + 5$)

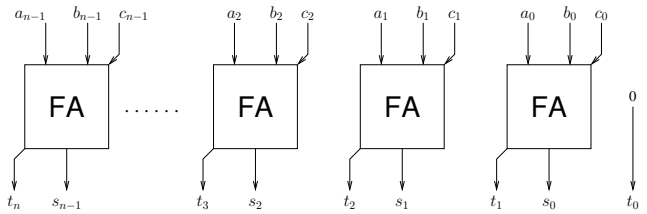
Carry-Save Adder (CSA)

- Problem: viele Zahlen sollen addiert werden
- Idee: Carry Propagierung vermeiden
- CSA für 3 Zahlen: Eingänge a, b, c, und zwei Ausgänge s, t.
  Der CSA wird auch als 3/2-Addierer bezeichnet.
- Berechne n-bit Vektoren s und t so dass

$$a + b + c = s + t$$

✔ Das kann mit Tiefe $O(1)$ geschehen!

Kombinatorischer Multiplizierer mit CSA



$$\langle t_{i+1} s_i \rangle = \langle a_i \rangle + \langle b_i \rangle + \langle c_i \rangle \quad \text{und} \quad t_0 = 0$$

Behauptung:

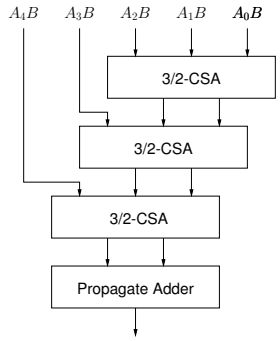
$$\langle a \rangle + \langle b \rangle + \langle c \rangle = \langle s \rangle + \langle t \rangle$$

Korrektheit CSA

Beweis:

$$\begin{aligned}
\langle a \rangle + \langle b \rangle + \langle c \rangle &= \sum_{i=0}^{n-1} (a_i + b_i + c_i) \cdot 2^i \\
&= \sum_{i=0}^{n-1} (2t_{i+1} + s_i) \cdot 2^i \\
&= \sum_{i=0}^{n-1} 2t_{i+1} \cdot 2^i + \sum_{i=0}^{n-1} s_i \cdot 2^i \\
&= \sum_{i=1}^{n} t_i \cdot 2^i + \sum_{i=0}^{n-1} s_i \cdot 2^i \\
&= \langle s \rangle + \langle t \rangle
\end{aligned}$$

Kombinatorischer Multiplizierer mit CSA

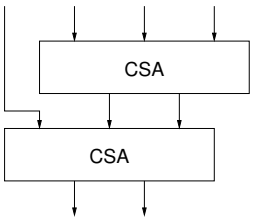


- Mehrere CSAs um alle partiellen Produkte aufzuaddieren
- Ein normaler Addierer für das Endergebnis (z.B. CLA)

Wallace Tree Multiplizierer

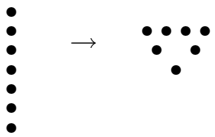
Offensichtliche Idee:
Baum balancieren (Wallace Tree)

Aus zwei 3/2-Addierern lässt sich ein 4/2-Addierer bauen:



Wallace Tree Multiplizierer

Damit erhält man einen gewöhnlichen Binärbaum:



Baum hat Tiefe $O(\log n)$

Optimierung Booth Multiplikation in Hardware

Erinnerung: Wir berechnen

$$\sum_{i=0}^{n-1} B \cdot (A_{i-1} - A_i) \cdot 2^i$$

Beispiel für 8 Bits:

$$\begin{array}{rcccccc} & B & (& A_{-1} & - & A_0 &) \; 2^0 \\ + & B & (& A_0 & - & A_1 &) \; 2^1 \\ + & B & (& A_1 & - & A_2 &) \; 2^2 \\ + & B & (& A_2 & - & A_3 &) \; 2^3 \\ + & B & (& A_3 & - & A_4 &) \; 2^4 \\ + & B & (& A_4 & - & A_5 &) \; 2^5 \\ + & B & (& A_5 & - & A_6 &) \; 2^6 \\ + & B & (& A_6 & - & A_7 &) \; 2^7 \end{array}$$

Booth Multiplikation in Hardware

Idee: Wir fassen jeweils zwei Summanden zusammen:

$$\begin{array}{rcccccc} & B & (& A_{-1} & - & A_0 &) \; 2^0 \\ + & B & (& A_0 & - & A_1 &) \; 2^1 \\ + & B & (& A_1 & - & A_2 &) \; 2^2 \\ + & B & (& A_2 & - & A_3 &) \; 2^3 \\ + & B & (& A_3 & - & A_4 &) \; 2^4 \\ + & B & (& A_4 & - & A_5 &) \; 2^5 \\ + & B & (& A_5 & - & A_6 &) \; 2^6 \\ + & B & (& A_6 & - & A_7 &) \; 2^7 \end{array}$$

✔ Dann hätte man die Anzahl der Summanden halbiert.

Booth Multiplikation in Hardware

Allgemein:

$$\begin{array}{rcccccc} & B & (& A_{i-1} & - & A_i &) \; 2^i \\ + & B & (& A_i & - & A_{i+1} &) \; 2^{i+1} \\ = & B & (& A_{i-1} - A_i + 2(A_i - A_{i+1}) & & &) \; 2^i \\ = & B & (& A_{i-1} + A_i - 2A_{i+1} & & &) \; 2^i \end{array}$$

Booth Multiplikation in Hardware

Wie berechen wir $B(A_{i-1} + A_i - 2A_{i+1})$?

Tabelle:

A_{i+1}	A_i	A_{i-1}	Summand
0	0	0	$+0B$
0	0	1	$+1B$
0	1	0	$+1B$
0	1	1	$+2B$
1	0	0	$-2B$
1	0	1	$-1B$
1	1	0	$-1B$
1	1	1	$-0B$

„Radix 4 Booth Encoding“

Wird auch „Sign-Magnitude“-Darstellung genannt

Booth Multiplikation in Hardware

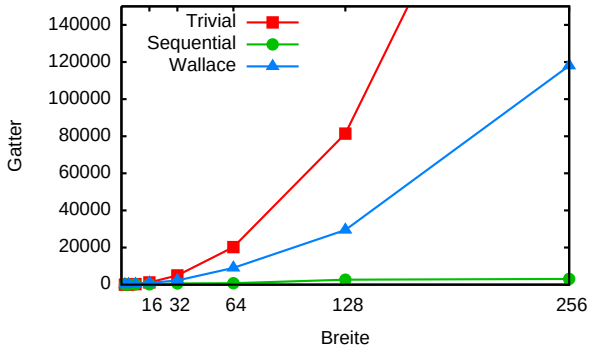
Auswahl aus: 0, B, $2B$, $-B$, $-2B$

Wir brauchen:

- einen Multiplexer zur Auswahl aus 0, B, $2B$
- einen Komplementierer für die untere Hälfte der Tabelle: bitweise invertieren (XOR); die +1 geht in die CSAs als Carry-in

→ In der Praxis ca. 25 % Einsparung bei Kosten und Tiefe

Vergleich Multiplizierer



Zusammenfassung Multiplizierer

- Multiplikation braucht Platz (Wallace Tree) oder Zeit (sequentielle Implementierung)
- Carry-Save-Addierer: keine Carry-Kette!
- Vorzeichen mit Booth-Recoding behandeln!
- Booth-Recoding reduziert auch die Anzahl der partiellen Produkte
- Weitere Operationen wie Division haben ähnliche Trade-Offs