Stoffplan zur Vorlesung „Rechnerarithmetik“

Einführung.

Zahlenbereiche. Zahlendarstellungen. Natürliche Zahlen und Maschinenzahlen. Ganze Zahlen und Maschinenzahlen.

Repräsentationen. Residuenarithmetik.

Festkommasysteme. Gleitkommasysteme. Rundung. Standards IEEE-754/854. Implementierung von Gleitkomma-Operationen.

Logarithmische Zahlensysteme. Arithmetik variierender Genauigkeit.

Intervallarithmetik. Verifikationsnumerik.

(Komparatoren. Pipelining.)

Volladdierer.

Serieller Addierer. Von Neumann-Addierer. Ripple-Carry-Addierer. Carry-Skip-Addierer.

Carry-Lookahead-Addierer. Pyramiden-Addierer. Conditional-Sum-Addierer. Carry-Select-Addierer.

Subtraktion. Konversion. Mehr-Operanden-Addierer. SDNR-Arithmetik.

Serielle und sequenzielle Multiplikation. Beschleunigungstechniken für sequenzielle Multiplikation. Parallele Multiplikation.

Division. SRT-Division. Iterative Division. Quadratwurzel.

Berechnung von Standardfunktionen. CORDIC-Verfahren.

Gepackte Arithmetik. MMX. Asynchrone Arithmetik. Wave-Pipelining. Energiesparende Implementierung.

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Einführung in die Thematik

Einflussbereiche der Rechnerarithmetik: Prozessor-Implementierung

- Aufbau des Prozessors

- Registerorganisation
- Speicherorganisation
- Breite der Datenpfade

- Fließbandverarbeitung
- Taktfrequenz

- Platzbedarf
- Energiebedarf
- Wärmeabführung

Einflussbereiche der Rechnerarithmetik: Spezialprozessoren

- numerische Koprozessoren
- Signalprozessoren
- Grafikprozessoren
- Multimediaprozessoren
- Verschlüsselungstechnologie

Einflussbereiche der Rechnerarithmetik: Maschinenbefehlssatz

- Datenformate
- arithmetische Operationen

- Flags
- bedingte Sprünge

- Ausnahmebehandlung

- Dauer von Maschinenoperationen

Einflussbereiche der Rechnerarithmetik: höhere Programmiersprachen

- Datentypen
- Operatoren
- Standardfunktionen
- Ausnahmebehandlung

- Funktionsumfang

- Rundungsfehler
- Konvertierungsfehler
- Terminierungsprobleme

Einflussbereiche der Rechnerarithmetik: Anwendungen

- Numerik
- Grafik
- Multimedia
- Prozessdatenverarbeitung
- Echtzeitsysteme

- Rechengeschwindigkeit
- Rechengenauigkeit
- Stabilität

„Innenleben“ der Rechnerarithmetik

Welche Algorithmen/Schaltungen werden zur Berechnung benutzt?

- Grundrechnungsarten ($+$, $-$, $*$, $/$)
  - Verfahren der Schularithmetik (Berechnung „per Hand“)
  - zusätzlich auch leistungsfähigere Verfahren
- Standardfunktionen ($\sqrt{}$, log, exp, cos, arctan, sinh, artanh, ...)
  - kaum Hilfestellungen aus der Schularithmetik
  - Vielzahl von Verfahren unterschiedlicher Güte
  - Reihenentwicklung nur bedingt brauchbar

Kollidierende Optimierungsziele

Häufig stehen folgende Optimierungsziele in Konkurrenz zueinander:

- geringe Latenz
- großer Durchsatz
- hohe Genauigkeit

- geringer Energiebedarf

- geringer Platzbedarf
- kleiner Implementierungsaufwand

- geringer Entwicklungsaufwand

Unterschiede in Aufwand und Leistungsfähigkeit: Algorithmen

Beispiel: Auswertung von $f(x) = e^x, 0 < x < 1$ in n Bit Genauigkeit

verwendetes Verfahren	Implementierungsaufwand	Rechenaufwand
einstufige Wertetabelle	$2^n \times n$ Bit Speicher	1 Speicherzugriff
Taylor-Entwicklung $e^x \approx \sum_{i=0}^{k} \frac{x^i}{i!}$	Addition Multiplikation Division oder kleiner Speicher	stark abhängig von x konvergiert gut für $x \ll 1$
rationale Approximation (Quotient zweier Polynome in x, hier vom Grad 5)	Addition Multiplikation Division $11 \times n$ Bit Speicher	10 Additionen 10 Multiplikationen 1 Division 11 Speicherzugriffe
additive Normalisierung	Addition $n \times n$ Bit Speicher	n Speicherzugriffe $2n$ Additionen

Unterschiede in Aufwand und Leistungsfähigkeit: Schaltungen

Beispiel: Festkomma-Addierer der Wortlänge 64 Bit

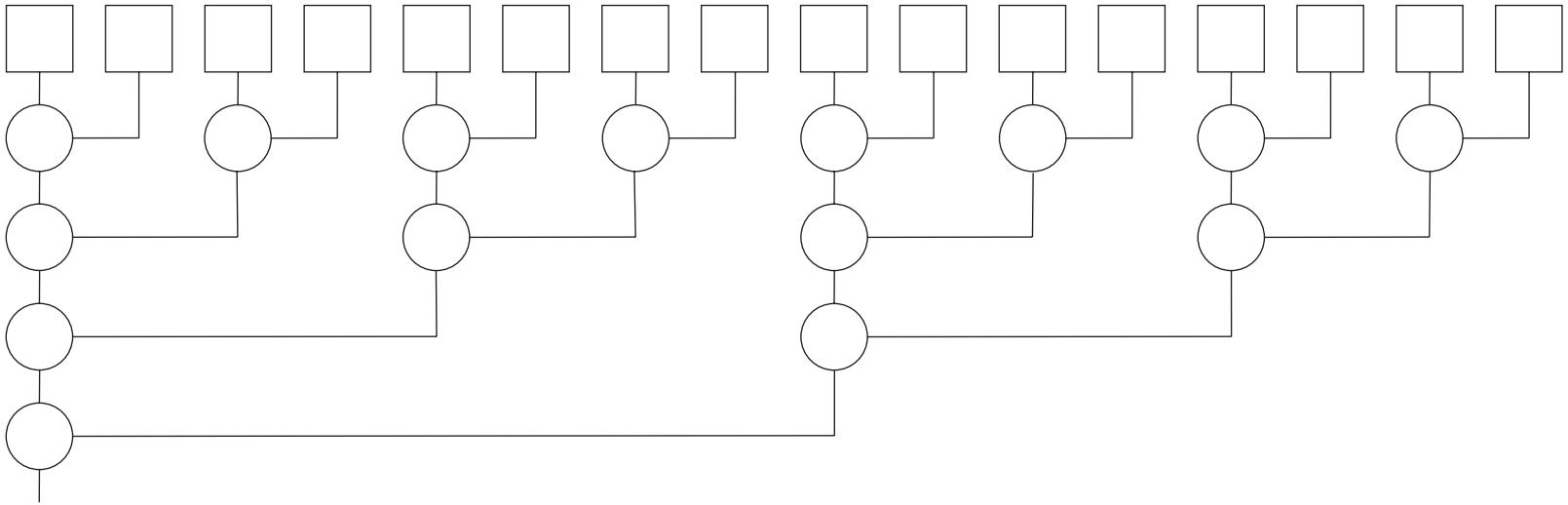
Bauweise	Latenz (Zyklen)	Aufwand (Transistoren)
Ripple-Carry	127	896
Carry-Select	6	2688
Carry-Lookahead	4	50624

(aus A. R. Omondi: Computer arithmetic systems, 1994, p. 99)

Hinweis: Die angegebenen Werte sind technologieabhängig!

Technologieabhängigkeit: Leitungslaufzeiten

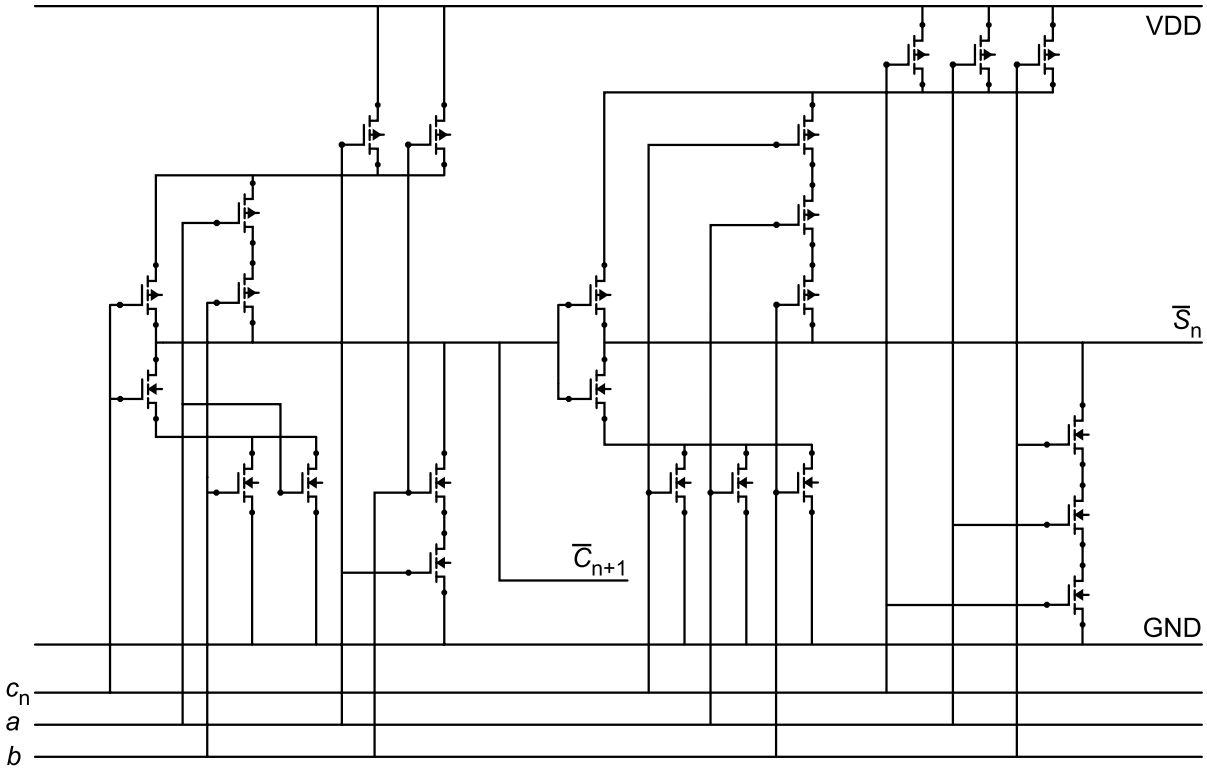
Beispiel: Baumartige Struktur mit n Eingängen (z. B. Komparator)



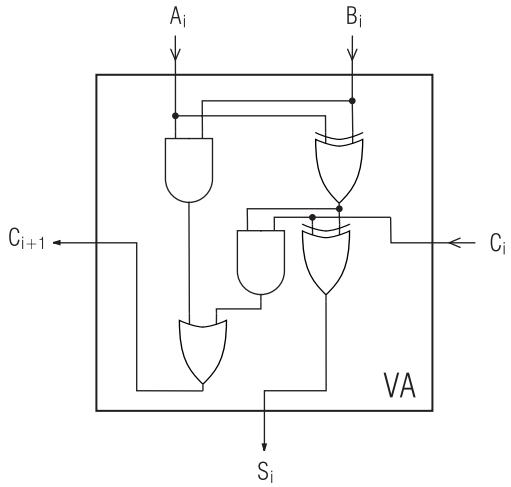
Falls Gatterschaltzeiten groß gegenüber Leitungslaufzeiten sind:
Latenz im wesentlichen proportional zur Baumhöhe $\log n$

Falls Gatterschaltzeiten klein gegenüber Leitungslaufzeiten sind:
Latenz im wesentlichen proportional zur Baumbreite n

Technologieabhängigkeit: Bauteile und Entwurfsstile



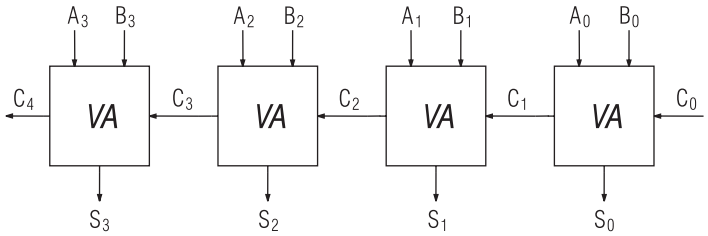
Volladdierer in CMOS-Implementierung
(aus N. Reifschneider: CAE-gestützte IC-Entwurfsmethoden, 1998, p. 125)



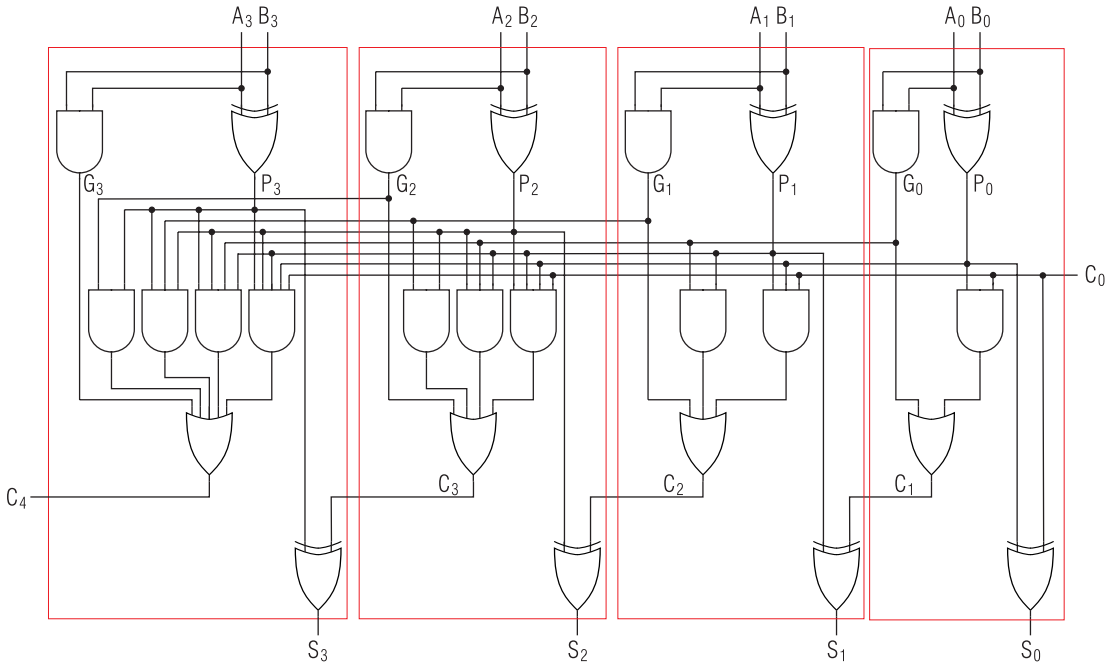
diskreter Aufbau (TTL o. ä.)

Berücksichtigung des Entwicklungsaufwands

Beispiel: Festkomma-Addierer der Wortlänge 4 Bit



Ripple-Carry-Addierer



Carry-Lookahead-Addierer

Ausnutzung von Synergie-Effekten

Werden mehrere arithmetische Funktionen zusammen implementiert, kann dies die Wahl der Algorithmen beeinflussen.

- Iterative Division als mehrstufiges Verfahren kann gut auf Multiplizierer kleiner Wortlänge aufgebaut werden.
- Sind mehrere gleichartige Standardfunktionen zu implementieren, bietet sich u. a. das CORDIC-Verfahren an.
- Sind die Geschwindigkeitsanforderungen gering und ist das CORDIC-Verfahren implementiert, können auch Multiplikation und Division damit realisiert werden.

„Außenwirkung“ der Rechnerarithmetik

Was kann über die berechneten Ergebnisse vorausgesetzt werden?

- Wird stets ein Ergebnis berechnet? (Bereichsüberschreitung; 1/0)
- Ist die einzelne Berechnung „korrekt“? (Rundungsfehler)
- Werden algebraische Gesetze eingehalten? $(a + (b + c) = (a + b) + c)$
- Was sagen die tatsächlich berechneten Ergebnisse einer Kette von Rechenoperationen über die eigentlich gesuchten Werte aus? (Inklusion, Anzahl gültiger Stellen)

Versagen der „klassischen“ Rechnerarithmetik

Beispiel: Auswertung arithmetischer Ausdrücke mit Microsoft Excel 2000

Ausdruck	Ergebnis	
10^20 - 10^20 + 17 - 10 + 130	137	(korrekt)
10^20 + 17 - 10 + 130 - 10^20	0	
10^20 + 17 - 10^20 - 10 + 130	120	
10^20 + 17 + 130 - 10^20 - 10	-10	
10^100 * 10^100 - 10^100 * 10^100 + 17 - 10 + 130	137	(korrekt)
10^200 * 10^200 - 10^200 * 10^200 + 17 - 10 + 130	—	
0,12345678912345678 9	0,1234567891234560 00	
1/3	0,3333333333333330 00	

Unabhängig von der Anzahl der angezeigten Ziffern speichert Excel Zahlen mit einer Genauigkeit von bis zu 15 Stellen hinter dem Komma. Besteht eine Zahl aus mehr als 15 signifikanten Ziffern, wandelt Excel die übrigen Stellen in Null (0) um. (aus: Microsoft Excel-Hilfe)

Auswirkungen von Entwurfsfehlern

Beispiel
Die Explosion der europäischen Trägerrakete „Ariane 5“ im Juni 1996 wurde verursacht durch einen Bereichsüberlauf bei der Umwandlung zwischen arithmetischen Zahlenformaten.
Der Unfall führte zur Zerstörung von vier Satelliten und verursachte einen Schaden von mindestens zwei Milliarden Mark.

Beispiel
Nachdem im Laufe des Jahres 1994 ein Fehler in der Gleitkomma-Einheit des Pentium-Prozessors entdeckt wurde (der sogenannte „Pentium-FDIV-Bug“), sagte Intel den kostenlosen Austausch aller fehlerhaften Pentium-Prozessoren gegen fehlerfreie Exemplare zu.

Vorlesungsinhalte: Gegenstände

- Zahlenbereiche
- Zahlendarstellungen
- Operationen auf Zahlen und Repräsentationen
- Implementierungen (Algorithmen, Logik, Hardware)
- Standards
- Nichtstandard-Zahlendarstellungen
- Einbettung in Programmiersprachen
- Aspekte der Handhabung

Vorlesungsinhalte: qualitative Betrachtungen

- algebraische Eigenschaften
- Zusammenhang dargestellter vs. approximierter Zahlenbereich
- Rundung
- Fehlersituationen
- Fehlerbehandlung

Vorlesungsinhalte: quantitative Betrachtungen

- Speicheraufwand von Zahlendarstellungen
- Genauigkeit bezüglich approximiertem Zahlenbereich

- Geschwindigkeit von Operationen
- Hardware-/Software-Aufwand von Operationen
- Genauigkeit von Operationen

- Konflikte und Kompromisse

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Vorzeichenlose ganze Zahlen

Zahlensysteme

Zahlenbereich S (Zahlenmenge)

Zahl $s \in S$

dargestellt durch

Menge R von *Repräsentationen*

Repräsentation $r \in R$

Zahlensystem $(S, R, I\colon R \to S)$

Beispiel

$(r_{l-1}, \ldots, r_0)$ mit $r_i \in \{0, 1\}$ stellt die Zahl $s = \sum_{i=0}^{l-1} r_i \times 2^i$ dar,

also $R = \{0, 1\}^l$ und $S = [0, 2^l - 1] \cap \mathbb{Z}$.

Eigenschaften der Interpretation

- Vollständigkeit der Zahlendarstellung

  Die Interpretation $I\colon R \to S$ ist surjektiv:
  Jede Zahl $s \in S$ wird durch mindestens eine Repräsentation $r \in R$ dargestellt.

- Eindeutigkeit

  Die Interpretation $I\colon R \to S$ ist immer eine wohldefinierte Abbildung:
  Aus jeder Repräsentation $r \in R$ ergibt sich eindeutig die dargestellte Zahl $s = I(r)$.

- Redundante Zahlendarstellung

  Die Umkehrung der Interpretation $I\colon R \to S$ braucht nicht eindeutig zu sein:
  Eine Zahl $s \in S$ kann mehrere Repräsentationen $r \in R$ besitzen.

  Beispiel: Vorzeichen/Betrag-Darstellung $\pm r_{l-1} \ldots r_0$
  Die Zahl Null besitzt die Repräsentationen $+0 \ldots 0$ und $-0 \ldots 0$

  Eine Zahlendarstellung (bzw. ein Zahlensystem) heißt *redundant*, wenn I nicht injektiv ist.

Approximation

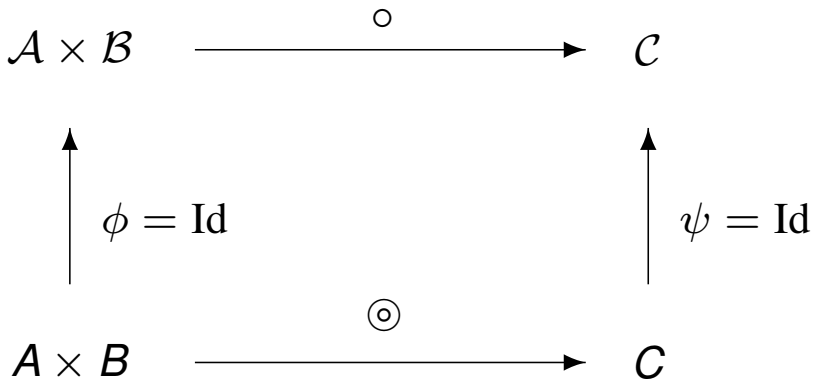
Meist sind Zahlensysteme *Approximationen* wohlbekannter mathematischer Strukturen.

Beispiel
Gleitkommasystem als Approximation des Körpers $\mathbb{Q}$ oder $\mathbb{R}$.

Konsequenz: Algebraische Gesetze gelten nur eingeschränkt!

Beispiel
$+$ und $\times$ sind auf Gleitkommazahlen nicht assoziativ.

Wertverlaufsgleichheit der Operationen



Zahlenbereiche A, B, C approximieren die Zahlenbereiche $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$.

$\circledcirc$ wertverlaufsgleich zu $\circ$, wenn $\forall a \in A, b \in B$:

(1) $a \circledcirc b$ definiert $\iff$ $a \circ b$ definiert,

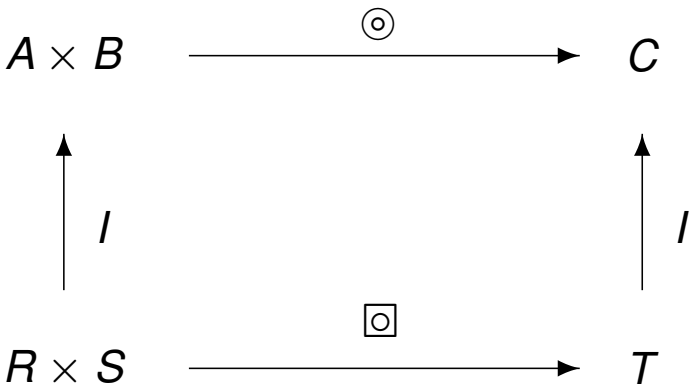
(2) $a \circledcirc b = a \circ b$, falls $a \circ b$ definiert.

Letzteres kann als Morphismus $\psi(a \circledcirc b) = \phi(a) \circ \phi(b)$ aufgefasst werden.

Approximationsanomalien

$a \circ b$ definiert?	$a \circledcirc b$ definiert?	Wertevergleich	Verlauf
undefiniert	undefiniert		korrekt
definiert	definiert	$a \circledcirc b = a \circ b$	korrekt
undefiniert	definiert		anomal
definiert	undefiniert		anomal
definiert	definiert	$a \circledcirc b \neq a \circ b$	anomal

Korrekte Implementierung der Operationen



Maschinenbereiche R, S, T implementieren Zahlenbereiche A, B, C.

Implementierung ist korrekt, wenn $\forall r \in R, s \in S$:

(1) $r \boxdot s$ definiert $\iff$ $I(r) \circledcirc I(s)$ definiert,

(2) $I(r \boxdot s) = I(r) \circledcirc I(s)$, falls $r \boxdot s$ definiert.

Einfluss der Darstellung auf den Zahlenbereich

Die prinzipielle Art der Zahlendarstellung impliziert strukturelle Eigenschaften des Zahlenbereichs.

Darstellung (Beispiel)	Definierende Formeln	$\lvert S \rvert$
Binär ohne Vorzeichen	$s = \sum_{i=0}^{l-1} r_i \times 2^i$	2^l
Dezimal ohne Vorzeichen	$s = \sum_{i=0}^{l-1} r_i \times 10^i$	10^l
Binär mit Vorzeichen/Betrag	$s = (-1)^{r_{l-1}} \times \sum_{i=0}^{l-2} r_i \times 2^i$	$2^l - 1$
2-Komplement-Darstellung	$s = -r_{l-1} \times 2^{l-1} + \sum_{i=0}^{l-2} r_i \times 2^i$	2^l
Residuendarstellung	$s \equiv r_i \pmod{p_i}$	$\operatorname{kgV}_i\{p_i\}$

Vorzeichenlose ganze Zahlen: Zahlenbereich

Vorzeichenlose (oder nichtnegative) ganze Zahlen = *unsigned integer*

Approximierter Zahlenbereich: $\mathbb{N}$

Dargestellter Zahlenbereich: $\mathrm{UInt}(K) = [0, K-1] \cap \mathbb{Z}$ mit $K \geq 2$

Häufig $K = 2^l$ mit $l \in \mathbb{N}^*$

Beispiele

- C: `unsigned`, `unsigned long`, `unsigned short`, `unsigned char`
- Java: `char` (16-Bit-Unicode-Zeichen, vor arithmetischen Operationen „binary numeric promotion“ zu `int`)
- Modula-3: `Cardinal` und bestimmte Unterbereichstypen
- Pascal: Bestimmte Unterbereichstypen
- Ada: Modular-Typen; `Natural` und andere Integer-Untertypen

UInt(2^l): Repräsentation

Wenn nicht anders vereinbart, liege für UInt(2^l) folgende Repräsentation vor:

Eine Sequenz $(r_{l-1}, r_{l-2}, \ldots, r_1, r_0)$ mit $r_i \in \{0, 1\}$
repräsentiert die vorzeichenlose ganze Zahl $s = \sum_{i=0}^{l-1} r_i \times 2^i$.

Statt UInt(2^l) schreiben wir dann auch $\mathrm{UInt}_2(l)$

Zur Verdeutlichung der *Basis* (oder *Radix*) 2 auch geschrieben als
$(r_{l-1}, r_{l-2}, \ldots, r_1, r_0)_2$ oder $(r_{l-1} r_{l-2} \ldots r_1 r_0)_2$
Radix-2-Repräsentation; r_i heißt *Ziffer* oder (in diesem Fall) *Bit*.

Länge l der Darstellung ist fest.
Korrespondiert zu fester Registerlänge bzw. festem Speicherformat.

UInt(K): Prädikate

$=$	Gleichheitsrelation
$\neq$	Ungleichheitsrelation
$<$	Kleiner-Relation
$\leq$	Kleiner/Gleich-Relation
$>$	Größer-Relation
$\geq$	Größer/Gleich-Relation

$=_0$	Test auf Null
$\neq_0$	Test auf nicht Null
$>_0$	Test auf größer Null

Alle aufgezählten Prädikate in UInt(K) sind wertverlaufsgleich mit den entsprechenden Prädikaten in $\mathbb{N}$ (und damit total).

Zur Darstellung der aufgezählten Prädikate genügt eine geeignete Teilmenge, etwa $\{\leq\}$, zusammen mit logischen Verknüpfungen:

$$
\begin{aligned}
s \geq t &\Leftrightarrow t \leq s \\
s = t &\Leftrightarrow s \leq t \wedge t \leq s \\
s \neq t &\Leftrightarrow \neg\, s = t \\
s < t &\Leftrightarrow s \leq t \wedge s \neq t \\
s > t &\Leftrightarrow t < s
\end{aligned}
\qquad
\begin{aligned}
s =_0 &\Leftrightarrow s = 0 \\
s \neq_0 &\Leftrightarrow s \neq 0 \\
s >_0 &\Leftrightarrow s > 0 \quad \text{bzw.} \\
s >_0 &\Leftrightarrow s \neq_0
\end{aligned}
$$

UInt(K): Operationen und algebraische Eigenschaften

$\oplus_K$	Addition	$\oslash_K$	Division (ganzzahliger Anteil)
$\ominus_K$	Subtraktion	$\otimes_K$	Divisionsrest
$\otimes_K$	Multiplikation	$\text{conv}_{K',K}$	Konversion UInt(K') $\rightarrow$ UInt(K)

Subskripte K bzw. K' werden weggelassen, falls Bezug zu UInt(K) und UInt(K') klar.

Von der Intention her approximiert (UInt(K), $\oplus$, $\otimes$) meist ($\mathbb{N}$, $+$, $\times$).

Wesentlicher Unterschied: UInt(K) endlich, $\mathbb{N}$ unendlich.

Konsequenz: $\oplus$ und $\otimes$ in UInt(K) nicht identisch mit $+$ bzw. $\times$ in $\mathbb{N}$.

UInt(K) wird charakterisiert durch die Intervallgrenzen $s_{min} = 0$ und $s_{max} = K - 1$:

$\forall s, t \in \mathbb{Z}$: $\quad \text{UInt}(K) = [s, t] \cap \mathbb{Z} \quad \Leftrightarrow \quad s = s_{min} \wedge t = s_{max}$

Arithmetik in UInt(K): Drei gängige Varianten

Es gibt drei gebräuchliche Formen der Arithmetik in UInt(K):
Überlaufarithmetik, zirkuläre Arithmetik und Sättigungsarithmetik.

Alle drei Varianten liefern das exakte Ergebnis einer Operation in $\mathbb{N}$,
sofern dieses in $\mathbb{N}$ definiert und in UInt(K) darstellbar ist.

In allen anderen Fällen (Überlaufbereich) erfolgt eine unterschiedliche Behandlung:

- Überlaufarithmetik (Integer-Untertypen in Ada, evtl. Pascal oder Modula-3):
  Wert undefiniert, evtl. Unterbrechung oder Flag gesetzt.
- Zirkuläre Arithmetik (meist in C, Modular-Typen in Ada, alle gängigen Mikroprozessoren):
  Alle Berechnungen erfolgen modulo K.
- Sättigungsarithmetik (HP MAX-1, Intel MMX, einzelne Befehle im SPARC):
  Negative Werte werden durch 0 ersetzt, positive durch $K - 1$.

Werteverlauf der drei Varianten für UInt(K)

- Alle drei Varianten sind so weit wie möglich wertverlaufsgleich zur Arithmetik in $\mathbb{N}$.
- Falls die Überlaufarithmetik einen Wert liefert, ist dieser korrekt.
- Im Überlaufbereich liefern zirkuläre Arithmetik und Sättigungsarithmetik, bezogen auf eine einzelne Operation, meist verschiedene Ergebnisse, die beide „falsch“ im Sinne der Arithmetik in $\mathbb{N}$ sind.

  Beispiel, dass dies jedoch nicht immer gilt: $3 \otimes_8 5$ undefiniert in Überlaufarithmetik, aber $3 \otimes_8 5 = 7$ sowohl in zirkulärer Arithmetik als auch in Sättigungsarithmetik.
- Trat bei der Durchführung einer Folge von Operationen in zirkulärer Arithmetik oder Sättigungsarithmetik Überlauf auf, resultiert in Einzelfällen dennoch das korrekte Ergebnis aus $\mathbb{N}$.

  Beispiel: In zirkulärer Arithmetik gilt stets $(a \oplus b) \ominus b = a$.

UInt(K): Addition

Addition $+$ total in $\mathbb{N}$: $\forall a, b \in \mathbb{N} : \exists c \in \mathbb{N} : a + b = c$

Wertverlaufsgleichheit zu Addition in UInt(K) unmöglich, da $s_{max} + s_{max} \notin \text{UInt}(K)$.

Überlaufarithmetik: $\forall a, b, c \in \text{UInt}(K) : a \oplus b = c \Leftrightarrow a + b = c$ (in $\mathbb{N}$)

Nicht total: $s_{max} + s_{max} > s_{max}$, also $s_{max} \oplus s_{max}$ undefiniert.

Zirkuläre Arithmetik total: $\forall a, b \in \text{UInt}(K) : a \oplus b = (a + b) \bmod K$

Es gilt $(\text{UInt}(K), \oplus) \cong (\mathbb{Z}_K, +)$, Anordnungseigenschaften verletzt:
$\forall a, b \in \mathbb{N} : a > 0 \Rightarrow a + b > 0$, aber $1 \oplus s_{max} = 0$.

Sättigungsarithmetik total: $\forall a, b \in \text{UInt}(K) : a \oplus b = \min\{a + b, s_{max}\}$

Die Kürzungsregeln gelten nicht mehr:
$\forall a, b, c \in \mathbb{N} : a + c = b + c \Rightarrow a = b$, aber $0 \oplus s_{max} = s_{max} = 1 \oplus s_{max}$.

Strikte Anordnungseigenschaften werden verletzt:
$\forall a, b, c \in \mathbb{N} : a < b \Rightarrow a + c < b + c$, aber $0 \oplus s_{max} = s_{max} = 1 \oplus s_{max}$.

UInt(K): Subtraktion

Subtraktion $-$ partiell in $\mathbb{N}$: $\forall a, b \in \mathbb{N}\colon (\exists c \in \mathbb{N}\colon a - b = c) \Leftrightarrow a \geq b$

Überlaufarithmetik wertverlaufsgleich zu Arithmetik in $\mathbb{N}$:
$\forall a, b, c \in \text{UInt}(K)\colon a \ominus b = c \Leftrightarrow a - b = c \quad (\text{in } \mathbb{N})$
Beispiel: $3 \ominus 5$ korrekterweise undefiniert, da $3 < 5$.

Zirkuläre Arithmetik total: $\forall a, b \in \text{UInt}(K)\colon a \ominus b = (a - b) \bmod K$
Beispiel: $3 \ominus_{16} 5 = 14$, da $3 - 5 \equiv -2 \equiv 14 \pmod{16}$.

Sättigungsarithmetik total: $\forall a, b \in \text{UInt}(K)\colon a \ominus b = \max\{a - b, 0\}$
Beispiel: $3 \ominus 5 = 0$, da $\max\{3 - 5, 0\} = \max\{-2, 0\} = 0$.

UInt(K): Multiplikation

Multiplikation $\times$ total in $\mathbb{N}$: $\forall a, b \in \mathbb{N} : \exists c \in \mathbb{N} : a \times b = c$

Wertverlaufsgleichheit zu Multiplikation in UInt(K) für $K > 2$ unmöglich, da dann $s_{max} \times s_{max} \notin \text{UInt}(K)$; für $K = 2$ besteht immer Wertverlaufsgleichheit.

Überlaufarithmetik: $\forall a, b, c \in \text{UInt}(K) : a \otimes b = c \Leftrightarrow a \times b = c \quad (\text{in } \mathbb{N})$

Nicht total für $K > 2$: $s_{max} \times s_{max} > s_{max}$, also $s_{max} \otimes s_{max}$ undefiniert.

Zirkuläre Arithmetik total: $\forall a, b \in \text{UInt}(K) : a \otimes b = (a \times b) \bmod K$

$(\text{UInt}(K), \oplus, \otimes) \cong (\mathbb{Z}_K, +, \times)$, Anordnungseigenschaften verletzt für $K > 2$:
$\forall a, b, c \in \mathbb{N} : a \leq b \Rightarrow a \times c \leq b \times c$; $1 \otimes s_{max} > s_{max} - 1 = 2 \otimes s_{max}$.

Sättigungsarithmetik total: $\forall a, b \in \text{UInt}(K) : a \otimes b = \min\{a \times b, s_{max}\}$

Die Kürzungsregeln gelten nicht mehr für $K > 2$:
$\forall a, b \in \mathbb{N}, c \in \mathbb{N}^* : a \times c = b \times c \Rightarrow a = b$; $1 \otimes s_{max} = s_{max} = 2 \otimes s_{max}$.

Strikte Anordnungseigenschaften werden verletzt für $K > 2$:
$\forall a, b \in \mathbb{N}, c \in \mathbb{N}^* : a < b \Rightarrow a \times c < b \times c$; $1 \otimes s_{max} = s_{max} = 2 \otimes s_{max}$.

UInt(K): Ganzzahlige Division

Ganzzahlige Division $\div$ nur auf $\mathbb{N} \times \mathbb{N}^*$ definiert:

$$\forall a, b, c \in \mathbb{N}: \quad a \div b = c \;\Leftrightarrow\; (b \neq 0 \;\wedge\; \exists d \in \mathbb{N}: d < b \;\wedge\; a = d + c \times b)$$

Für $b \neq 0$ lassen sich stets entsprechende Werte c und d finden.

Wertverlaufsgleiche Division $\circledcirc$ ist in Überlaufarithmetik stets gegeben, in zirkulärer Arithmetik und Sättigungsarithmetik zumindest möglich (durch Anzeigen einer Ausnahme).

Null als zweites Argument kann zu folgenden Ereignissen führen:

- Ausnahme wegen Division durch Null (evtl. auch still)
- Ausnahme wegen Überlaufs (evtl. auch still)
- Berechnung eines in der Regel belanglosen Ergebnisses
- In Sättigungsarithmetik alternativ auch Ergebnis s_{max}

UInt(K): Divisionsrest

Der Divisionsrest $\oslash_K$ in UInt(K) ist in allen drei Varianten wertverlaufsgleich mit dem Divisionsrest in $\mathbb{N}$, jedenfalls soweit das zweite Argument von Null verschieden ist:

$$\forall a \in \mathbb{N}, b \in \mathbb{N}^* : a \oslash b = a \ominus b \otimes (a \oslash b)$$

Das Verhalten bezüglich Null als zweitem Argument folgt sinnvollerweise dem bei ganzzahliger Division:

- Bei undefiniertem Quotienten ist auch der Rest undefiniert.
- Bei definiertem Quotienten ist, um obige Identität fortzusetzen, der Rest gleich dem ersten Argument.

Konversion $\mathrm{UInt}(K') \to \mathrm{UInt}(K)$

- $K' < K$

  $\mathrm{conv}_{K',K}$ ist dann total und stimmt mit der Identität überein.

- $K' > K$

  In Überlaufarithmetik ist $\mathrm{conv}_{K',K}$ auf $\mathrm{UInt}(K)$ wertverlaufsgleich mit der Identität, und sonst undefiniert (Überlauf).

  In zirkulärer Arithmetik gilt $\mathrm{conv}_{K',K}(a) = a \bmod K$.

  In Sättigungsarithmetik ist $\mathrm{conv}_{K',K}(a) = \min\{a, K-1\}$.

Gestaltungsspielraum bei Fehlerbehandlung

- Erzwungene Ausnahmebehandlung nach Fehlerauftritt

  Operation partiell implementiert, löst bei Anwendung auf Operanden ohne definiertes Ergebnis Unterbrechung aus.

  Durchführung kann aufwendig sein.

- Stille Fehlerbehandlung (optionale Ausnahmebehandlung)

  Operation total implementiert, lässt nach Ausführung erkennen, ob Ergebnis falsch im Sinne exakter Arithmetik (meist an den Flags).

- Keine unmittelbare Fehlererkennung

  Operation wie bei stiller Fehlerbehandlung implementiert, hinterlässt jedoch in den Flags etc. keine auswertbare Fehlerinformation.

  Ausnahmebehandlung schwierig und aufwendig, evtl. müssen die Operanden auf Zulässigkeit geprüft werden.

  Beispiel: `MULX` im SPARC-V9.

  In bestimmten Fällen genügt der abgelieferte Wert zur systematischen Fehlererkennung.

  Beispiel: Addition zweier positiver Zahlen ergibt negative Zahl.

Äquivalenz arithmetischer Ausdrücke

Bestimmte arithmetische Äquivalenzen aus $\mathbb{N}$ gelten trotz modifizierter Operationen formal auch in zirkulärer Arithmetik, nicht aber in Überlauf- oder Sättigungsarithmetik.

Beispiele

- $\forall a, b, c \in \mathrm{UInt}(K) : a \otimes (b \ominus c) = (a \otimes b) \ominus (a \otimes c)$
- $\forall a, b, c \in \mathrm{UInt}(K) : a \oplus (b \ominus c) = (a \oplus b) \ominus c$
- $\forall a, b \in \mathrm{UInt}(K) : (a \oplus b) \ominus b = a$
- $\forall a, b \in \mathrm{UInt}(K) : (a \oplus b) \ominus a = b$
- $\forall a, b \in \mathrm{UInt}(K) : (a \ominus b) \oplus b = a$

Die Varianten verhalten sich deswegen unterschiedlich hinsichtlich Programmtransformationen und Optimierungen in Übersetzern.

Umkehrung additiver Operationen

Jede Addition in zirkulärer Arithmetik kann durch eine entsprechende Subtraktion in zirkulärer Arithmetik umgekehrt werden, es gilt dabei

$$\forall a, b \in \mathrm{UInt}(K) : (a \oplus b) \ominus b = a \qquad \text{und} \qquad \forall a, b \in \mathrm{UInt}(K) : (a \oplus b) \ominus a = b$$

Jede Subtraktion in zirkulärer Arithmetik kann durch eine entsprechende Addition in zirkulärer Arithmetik umgekehrt werden, es gilt dabei

$$\forall a, b \in \mathrm{UInt}(K) : (a \ominus b) \oplus b = a$$

Jede fehlerfreie Addition in Überlaufarithmetik kann durch eine entsprechende Subtraktion in zirkulärer, Überlauf- oder Sättigungsarithmetik umgekehrt werden, es gilt dabei

$$\forall a, b \in \mathrm{UInt}(K) : (a \oplus b) \ominus b = a \qquad \text{und} \qquad \forall a, b \in \mathrm{UInt}(K) : (a \oplus b) \ominus a = b$$

Jede fehlerfreie Subtraktion in Überlaufarithmetik kann durch eine entsprechende Addition in zirkulärer, Überlauf- oder Sättigungsarithmetik umgekehrt werden, es gilt dabei

$$\forall a, b \in \mathrm{UInt}(K) : (a \ominus b) \oplus b = a$$

Umkehrung multiplikativer Operationen

Für die Betrachtung der Umkehrbarkeit wird ein positives zweites Argument vorausgesetzt.

Für $K = 2$ sind Multiplikation bzw. Division in jeder der drei Varianten identisch und zueinander invers, es gilt $\forall a, b \in \mathrm{UInt}(K), b \neq 0 : (a \otimes b) \oslash b = a \ \wedge \ (a \oslash b) \otimes b = a$

Für $K > 2$ wird jede fehlerfreie Multiplikation in Überlaufarithmetik durch eine entsprechende Division in zirkulärer, Überlauf- oder Sättigungsarithmetik umgekehrt.

Für $K > 2$ kann die Division in keiner der drei Varianten umgekehrt werden.

Für $K > 2$ kann die Multiplikation in Sättigungsarithmetik nicht umgekehrt werden.

Für nicht primes K kann die Multiplikation in zirkulärer Arithmetik nicht umgekehrt werden.

Für primes $K > 2$ ist $(\mathbb{Z}_K, +, \times)$ ein Körper, in dem Multiplikation und Division zueinander invers sind; die Division im Körper $\mathbb{Z}_K$ ist allerdings nicht wertverlaufsgleich zu der in $\mathbb{N}$.

Damit wird die Multiplikation in zirkulärer Arithmetik weder durch die Division in zirkulärer Arithmetik noch in Überlaufarithmetik oder Sättigungsarithmetik umgekehrt.

Doppelt genaue Multiplikation

Die Multiplikation $\otimes$ kann in allen drei Varianten wertverlaufsgleich zur Arithmetik in $\mathbb{N}$ gehalten werden, wenn für das Ergebnis ein geeignet erweiterter Zahlenbereich benutzt wird.

Unter einer doppelt genauen Multiplikation versteht man üblicherweise die Abbildung

$\otimes\colon \mathrm{UInt}(2^l) \times \mathrm{UInt}(2^l) \to \mathrm{UInt}(2^{2l})$ mit $a \otimes b = a \times b$.

Beispiel: Multiplikation 32 Bit $\times$ 32 Bit $\to$ 64 Bit mittels `UMUL` im SPARC-V9,
im Gegensatz zu 64 Bit $\times$ 64 Bit $\to$ 64 Bit mittels `MULX`.

Bei einer dazu passenden Division $\oslash : \mathrm{UInt}(2^{2l}) \times \mathrm{UInt}(2^l) \to \mathrm{UInt}(2^l)$ kann es zu Überläufen kommen:

$\forall a \in \mathrm{UInt}(2^{2l}), b, c \in \mathrm{UInt}(2^l)\colon a \oslash b = c \Leftrightarrow a \div b = c \quad (\text{in } \mathbb{N})$

Der Überlaufbereich wird gemäß einer der drei Varianten von vorzeichenloser Arithmetik behandelt (Beispiel: Sättigungsarithmetik bei `UDIV` im SPARC).

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Vorzeichenbehaftete ganze Zahlen

Vorzeichenbehaftete ganze Zahlen: Zahlenbereich

(Vorzeichenbehaftete) ganze Zahlen = *(signed) integer*.

Approximierter Zahlenbereich: $\mathbb{Z}$

Dargestellter Zahlenbereich: $\mathrm{Int}(H, K) = [-H, K-1] \cap \mathbb{Z}$ mit $H, K \geq 1$

Häufig verwendete Zahlenbereiche:

- $\mathrm{Int}(2^{l-1}, 2^{l-1}) = [-2^{l-1}, 2^{l-1}-1] \cap \mathbb{Z}$ mit $l \in \mathbb{N}^*$, insbesondere $l = 8, 16, 32, 64$ bei 2-Komplement-Darstellung in gängigen Mikroprozessoren
- Desgleichen Datentypen in Java: `byte`, `short`, `int`, `long`
- $\mathrm{Int}(2^{l-1}-1, 2^{l-1}) = [1-2^{l-1}, 2^{l-1}-1] \cap \mathbb{Z}$ mit $l \geq 2$, insbesondere bei Vorzeichen/Betrag-, 1-Komplement- oder Signed-Binary-Darstellung
- $\mathrm{Int}(10^{18}-1, 10^{18}) = [1-10^{18}, 10^{18}-1] \cap \mathbb{Z}$ (Packed-Decimal-Format im Intel 8087 ff.)
- Modula-3: beliebige Unterbereichstypen von `Integer`

Prädikate und Operationen in Int(H, K)

$=$	Gleichheitsrelation	$=_0$	Test auf Null
$\neq$	Ungleichheitsrelation	$\neq_0$	Test auf nicht Null
$>$	Größer-Relation	$>_0$	Test auf größer Null
$\geq$	Größer/Gleich-Relation	$\geq_0$	Test auf größer oder gleich Null
$<$	Kleiner-Relation	$<_0$	Test auf kleiner Null
$\leq$	Kleiner/Gleich-Relation	$\leq_0$	Test auf kleiner oder gleich Null

$\oplus_{H,K}$	Addition	$\otimes_{H,K}$	Multiplikation
$\oslash_{H,K}$	Ganzzahliger Quotient	$\otimes\!\!\!/_{H,K}$	Divisionsrest

$\ominus_{H,K}$ Subtraktion; als einstellige Operation: Vorzeichenwechsel

$\mathrm{conv}_{H',K',H,K}$ Konversion Int$(H', K') \longrightarrow$ Int(H, K)

Ist der Bezug zu Int(H, K) bzw. Int(H', K') klar, werden die Subskripte weggelassen.

Wertverlaufsgleichheit und Darstellung der Prädikate in Int(H, K)

Alle aufgezählten Prädikate in Int(H, K) sind wertverlaufsgleich mit den entsprechenden Prädikaten in $\mathbb{Z}$ (und damit total).

Zur Darstellung der aufgezählten Prädikate genügt eine geeignete Teilmenge, etwa $\{\geq\}$, zusammen mit logischen Verknüpfungen:

$$
\begin{array}{ll}
s \leq t & \Leftrightarrow \; t \geq s \\
s = t & \Leftrightarrow \; s \geq t \wedge t \geq s \\
s \neq t & \Leftrightarrow \; \neg\, s = t \\
s > t & \Leftrightarrow \; s \geq t \wedge s \neq t \\
s < t & \Leftrightarrow \; t > s
\end{array}
\qquad\qquad
\begin{array}{ll}
s \geq_0 & \Leftrightarrow \; s \geq 0 \\
s \leq_0 & \Leftrightarrow \; s \leq 0 \\
s =_0 & \Leftrightarrow \; s = 0 \\
s \neq_0 & \Leftrightarrow \; s \neq 0 \\
s >_0 & \Leftrightarrow \; s > 0 \\
s <_0 & \Leftrightarrow \; s < 0
\end{array}
$$

Algebraische Eigenschaften von Int(H, K)

Von der Intention her approximiert $(\mathrm{Int}(H, K), \oplus, \otimes)$ meist $(\mathbb{Z}, +, \times)$.

Wesentlicher Unterschied: $\mathrm{Int}(H, K)$ endlich, $\mathbb{Z}$ unendlich.
Konsequenz: $\oplus$, $\ominus$ und $\otimes$ in $\mathrm{Int}(H, K)$ nicht identisch mit $+$, $-$, $\times$ in $\mathbb{Z}$.

$\mathrm{Int}(H, K)$ wird charakterisiert durch die Intervallgrenzen $s_{\min} = -H$ und $s_{\max} = K - 1$:
$\forall s, t \in \mathbb{Z}\colon \quad \mathrm{Int}(H, K) = [s, t] \cap \mathbb{Z} \quad \Leftrightarrow \quad s = s_{\min} \wedge t = s_{\max}$

Arithmetik in Int(H, K): Drei gängige Varianten

Es gibt drei gebräuchliche Formen der Arithmetik in Int(H, K):
Überlaufarithmetik, zirkuläre Arithmetik und Sättigungsarithmetik.

Alle drei Varianten liefern das exakte Ergebnis einer Operation in $\mathbb{Z}$,
sofern ein solches in $\mathbb{Z}$ definiert und in Int(H, K) darstellbar ist.

In allen anderen Fällen (Überlaufbereich) erfolgt eine unterschiedliche Behandlung:

- Überlaufarithmetik (Ada, evtl. C, Pascal oder Modula-3)
  Undefiniertes Ergebnis; evtl. erfolgt Unterbrechung oder es wird ein Flag gesetzt.
- Zirkuläre Arithmetik (alle gängigen Mikroprozessoren, evtl. C)
  Alle Berechnungen erfolgen modulo $(H + K)$, mit H negativen Resten.
- Sättigungsarithmetik (HP MAX-1, Intel MMX)
  Negative Werte werden durch $-H$ ersetzt, positive durch $K - 1$.

Werteverlauf der drei Varianten für Int(H, K)

- Alle drei Varianten sind so weit wie möglich wertverlaufsgleich zur Arithmetik in $\mathbb{Z}$.
- Falls die Überlaufarithmetik einen Wert liefert, ist dieser korrekt.
- Im Überlaufbereich liefern zirkuläre Arithmetik und Sättigungsarithmetik, bezogen auf eine einzelne Operation, verschiedene Ergebnisse, die beide „falsch“ im Sinne der Arithmetik in $\mathbb{Z}$ sind.
- Trat bei der Durchführung einer Folge von Operationen in zirkulärer Arithmetik oder Sättigungsarithmetik Überlauf auf, resultiert in Einzelfällen dennoch das korrekte Ergebnis aus $\mathbb{Z}$.

  Beispiel: In zirkulärer Arithmetik gilt stets $(a \oplus b) \ominus b = a$.

Addition in $\mathrm{Int}(H, K)$

Addition $+$ ist total in $\mathbb{Z}$: $\forall a, b \in \mathbb{Z} : \exists c \in \mathbb{Z} : a + b = c$

Wertverlaufsgleichheit zu Addition in $\mathrm{Int}(H, K)$ unmöglich, da $s_{min} + s_{min} \notin \mathrm{Int}(H, K)$.

Überlaufarithmetik: $\forall a, b, c \in \mathrm{Int}(H, K) : a \oplus b = c \Leftrightarrow a + b = c$ (in $\mathbb{Z}$)

$\oplus$ ist partiell: In $\mathbb{Z}$ gilt $s_{\min} + s_{\min} < s_{\min}$, also $s_{\min} \oplus s_{\min}$ undefiniert.

Zirkuläre Arithmetik ist total: $\forall a, b \in \mathrm{Int}(H, K) : a \oplus b = ((a + b + H) \bmod (H + K)) - H$

Es gilt $(\mathrm{Int}(H, K), \oplus) \cong (\mathbb{Z}_{H+K}, +)$, Anordnungseigenschaften werden verletzt:
$\forall a, b \in \mathbb{Z} : a, b < 0 \Rightarrow a + b < 0$, aber $(-1) \oplus s_{\min} = s_{\max} \geq 0$.

Sättigungsarithmetik ist total: $\forall a, b \in \mathrm{Int}(H, K) : a \oplus b = \max\{s_{\min}, \min\{s_{\max}, a + b\}\}$

Die Kürzungsregeln gelten nicht mehr:
$\forall a, b, c \in \mathbb{Z} : a + c = b + c \Rightarrow a = b$, aber $s_{\min} \oplus s_{\min} = s_{\min} = 0 \oplus s_{\min}$, obwohl $s_{\min} \neq 0$.

Strikte Anordnungseigenschaften werden verletzt:
$\forall a, b, c \in \mathbb{Z} : a < b \Rightarrow a + c < b + c$, aber $s_{\min} \oplus s_{\min} = s_{\min} = 0 \oplus s_{\min}$ trotz $s_{\min} < 0$.

Subtraktion in $\mathrm{Int}(H, K)$

Subtraktion $-$ ist total in $\mathbb{Z}$: $\forall a, b \in \mathbb{Z} : \exists c \in \mathbb{Z} : a - b = c$

Überlaufarithmetik: $\forall a, b, c \in \mathrm{Int}(H, K) : a \ominus b = c \Leftrightarrow a - b = c$ (in $\mathbb{Z}$)

$\ominus$ ist partiell: In $\mathbb{Z}$ gilt $s_{\max} - s_{\min} > s_{\max}$, also $s_{\max} \ominus s_{\min}$ undefiniert.

Zirkuläre Arithmetik ist total: $\forall a, b \in \mathrm{Int}(H, K) : a \ominus b = ((a - b + H) \bmod (H + K)) - H$

Anordnungseigenschaften werden verletzt:
$\forall a, b, c \in \mathbb{Z} : a \leq b \Rightarrow a - c \leq b - c$, aber $(s_{\max} - 1) \ominus (-1) = s_{\max} > s_{\min} = s_{\max} \ominus (-1)$.

Sättigungsarithmetik ist total: $\forall a, b \in \mathrm{Int}(H, K) : a \ominus b = \max\{s_{\min}, \min\{s_{\max}, a - b\}\}$

Die Kürzungsregeln gelten nicht mehr:
$\forall a, b, c \in \mathbb{Z} : a - c = b - c \Rightarrow a = b$, aber $(s_{\max} - 1) \ominus (-1) = s_{\max} = s_{\max} \ominus (-1)$.

Strikte Anordnungseigenschaften werden verletzt:
$\forall a, b, c \in \mathbb{Z} : a < b \Rightarrow a - c < b - c$, aber $(s_{\max} - 1) \ominus (-1) = s_{\max} = s_{\max} \ominus (-1)$.

Multiplikation in Int(H, K)

Multiplikation $\times$ ist total in $\mathbb{Z}$: $\forall a, b \in \mathbb{Z} : \exists c \in \mathbb{Z} : a \times b = c$

In $\text{Int}(1, 2) = \{-1, 0, 1\}$ ist $\otimes$ wertverlaufsgleich mit $\times$ in $\mathbb{Z}$.

In allen anderen Fällen gilt:

Überlaufarithmetik ist partiell: $\forall a, b, c \in \text{Int}(H, K) : a \otimes b = c \Leftrightarrow a \times b = c$ (in $\mathbb{Z}$)
Für $K > 2$ ist $s_{\max} \times s_{\max} > s_{\max}$, also $s_{\max} \otimes s_{\max}$ undefiniert.
Für $K \leq 2$ ist $s_{\min} \times s_{\min} > s_{\max}$, also $s_{\min} \otimes s_{\min}$ undefiniert.

Zirkuläre Arithmetik ist total: $\forall a, b \in \text{Int}(H, K) : a \otimes b = ((a \times b + H) \bmod (H + K)) - H$

Es gilt $(\text{Int}(H, K), \oplus, \otimes) \cong (\mathbb{Z}_{H+K}, +, \times)$, Anordnungseigenschaften werden verletzt:

$\forall a, b \in \mathbb{Z}, c \in \mathbb{Z}^+ : a \leq b \Rightarrow a \times c \leq b \times c$, aber $1 \otimes s_{max} > 2 \otimes s_{max}$ für $K > 2$.

$\forall a, b \in \mathbb{Z}, c \in \mathbb{Z}^- : a \leq b \Rightarrow a \times c \geq b \times c$, aber $(-1 - s_{max}) \otimes (-1) < 0 \otimes (-1)$ sonst.

Sättigungsarithmetik ist total: $\forall a, b \in \text{Int}(H, K) : a \otimes b = \max\{s_{\min}, \min\{s_{\max}, a \times b\}\}$

Kürzungsregeln gelten nicht mehr, z. B. $\forall a, b \in \mathbb{Z}, c \in \mathbb{Z}^* : a \times c = b \times c \Rightarrow a = b$,

strikte Anordnungseigenschaften wie $\forall a, b \in \mathbb{Z}, c \in \mathbb{Z}^+ : a < b \Rightarrow a \times c < b \times c$
bzw. $\forall a, b \in \mathbb{Z}, c \in \mathbb{Z}^- : a < b \Rightarrow a \times c > b \times c$ werden verletzt:
$1 \otimes s_{max} = s_{max} = 2 \otimes s_{max}$ für $K > 2$, $s_{\min} \otimes s_{\min} = s_{max} = (1 + s_{\min}) \otimes s_{\min}$ sonst.

Ganzzahlige Division in $\mathrm{Int}(H, K)$

Die ganzzahlige Division $\div$ ist nur auf $\mathbb{Z} \times \mathbb{Z}^*$ definiert; die Ergebnisse unterliegen darüber hinaus nur folgender Einschränkung (entsprechende Werte q und r lassen sich stets finden):

$$\forall a, b, q \in \mathbb{Z}: \quad a \div b = q \;\Rightarrow\; (b \neq 0 \;\wedge\; \exists r \in \mathbb{Z}: |r| < |b| \;\wedge\; a = r + q \times b)$$

Häufig wird die Nebenbedingung $\mathrm{sgn}(r) = \mathrm{sgn}(a)$ verlangt, um $|q|$ zu minimieren;
eine Alternative wäre die Minimierung von $|r|$.

Nur bei symmetrischem Zahlenbereich, also $H = K - 1$, kann stets wertverlaufsgleich zu $\div$ gerechnet werden: $\forall a, b, q \in \mathrm{Int}(H, K): a \operatorname{\ominus} b = q \Leftrightarrow a \div b = q \quad (\text{in } \mathbb{Z})$

Andernfalls führt die Division durch -1 für mindestens einen Wert des Dividenden zu einem Überlauf oder zu einem im Sinne der Division in $\mathbb{Z}$ nicht korrekten Ergebnis.

Für den häufigsten Anwendungsfall $\mathrm{Int}(K, K)$ stellt $s_{\min} \operatorname{\ominus} (-1)$ neben der Division durch Null den einzigen Problemfall dar. Beispiele für die Behandlung:

- SDIVX-Operation im SPARC-V9, Java: Zirkuläre Arithmetik, keine Überlauferkennung.
- MC68020: Flag wird gesetzt.

Divisionsrest in Int(H, K)

Der Divisionsrest $\oslash_{H,K}$ in Int(H, K) ist wertverlaufsgleich mit dem Divisionsrest in $\mathbb{Z}$, sofern kein Überlauf auftrat:

$$\forall a \in \mathbb{Z}, b \in \mathbb{Z}^* : a \oslash_{H,K} b = a \ominus_{H,K} b \otimes_{H,K} (a \oslash_{H,K} b)$$

Das Verhalten bei Überlauf oder Division durch Null folgt sinnvollerweise dem bei ganzzahliger Division:

- Bei undefiniertem Quotienten ist auch der Rest undefiniert.
- Bei Division durch Null und definiertem Quotienten ist, um obige Identität fortzusetzen, der Rest gleich dem ersten Argument.
- In Sättigungsarithmetik kann obige Formel zur Berechnung eines Restes benutzt werden.

Vorzeichenwechsel in $\mathrm{Int}(H, K)$

Als einstellige Operation bewirkt $-$ einen Vorzeichenwechsel.

Mögliche Einschränkungen ergeben sich aus dem Zusammenhang $-a = a \div (-1)$:

Ist $-a \notin \mathrm{Int}(H, K)$, liefern weder $\ominus_{H,K} a$ noch $a \oslash_{H,K} (-1)$ den entsprechenden Wert aus $\mathbb{Z}$.

Dies impliziert im Allgemeinen noch nicht, dass $\ominus_{H,K} a$ durch $a \oslash_{H,K} (-1)$ (bzw. umgekehrt) implementiert werden muss, oder dass $\ominus_{H,K} a$ und $a \oslash_{H,K} (-1)$ auch nur wertverlaufsgleich sind.

In zirkulärer, Überlauf- und Sättigungsarithmetik sind allerdings $\ominus_{H,K} a$ und $a \oslash_{H,K} (-1)$ tatsächlich wertverlaufsgleich.

In zirkulärer und Sättigungsarithmetik gilt überdies der formale Zusammenhang
$\ominus_{H,K} a = a \oslash_{H,K} (-1)$.

Doppelt genaue Multiplikation für vorzeichenbehaftete ganze Zahlen

Die Multiplikation $\otimes$ kann wertverlaufsgleich zu $\times$ in $\mathbb{Z}$ gehalten werden, wenn für das Ergebnis ein geeignet erweiterter Zahlenbereich benutzt wird. (Weiterrechnen schwierig!)

Der Argumentbereich $\mathrm{Int}(H, K) \times \mathrm{Int}(H', K')$ erfordert einen Resultatbereich $\mathrm{Int}(H'', K'')$ mit $K'' > H \times H'$, $K'' > (K-1) \times (K'-1)$, $H'' \geq H \times (K'-1)$ und $H'' \geq H' \times (K-1)$.

In der Situation $\mathrm{Int}(K, K)$, $\mathrm{Int}(K', K')$ und $\mathrm{Int}(K'', K'')$ hat $K'' > K \times K'$ zu gelten.

Häufigster Anwendungsfall ist $K = K' = 2^{l-1}$ mit $K'' = 2^{2\times l-1}$.

Beispiele: iAPX 86, MC68000, SPARC-V9.

Die Bezeichnung „doppelt genaue Multiplikation“ leitet sich aus der Beziehung $|\mathrm{Int}(2^{2\times l-1}, 2^{2\times l-1})| = 2^{2\times l} = 2^l \times 2^l = |\mathrm{Int}(2^{l-1}, 2^{l-1})|^2$ ab.

Für symmetrische Zahlenbereiche $\mathrm{Int}(K-1, K)$, $\mathrm{Int}(K'-1, K')$ und $\mathrm{Int}(K''-1, K'')$ bedeuten obige Einschränkungen die Beziehung $K'' > (K-1) \times (K'-1)$.

Ein wichtiger Spezialfall ist $K = K' = 2^{l-1}$ und $K'' = 2^{2\times l-2}$.

Division mit gemischten Zahlenbereichen

Bei einer Division $\oslash : \mathrm{Int}(H, K) \times \mathrm{Int}(H', K') \to \mathrm{Int}(H'', K'')$ mit $H'' < H$ oder $K'' < K$ kann es zu Überläufen kommen:

$\forall a \in \mathrm{Int}(H, K), b \in \mathrm{Int}(H', K'), q \in \mathrm{Int}(H'', K'') : a \oslash b = q \Leftrightarrow a \div b = q \quad (\text{in } \mathbb{Z})$

Die Nebenbedingung $\mathrm{sgn}(r) = \mathrm{sgn}(a)$ kann zur Minimierung von $|q|$ benutzt werden; dadurch reduziert sich die Anzahl der Überlaufe.

Häufigster Anwendungsfall: $H = K = 2^{2 \times l - 1}$, $H' = K' = H'' = K'' = 2^{l-1}$.

Beispiele für die Behandlung des Überlaufbereichs:

- IDIV-Operation des iAPX 86: Unterbrechung, kein definiertes Ergebnis.
- DIVS-Operation des MC68000: Setzen eines Flags, kein definiertes Ergebnis, Argumente unverändert.
- SDIV-Operation des SPARC-V9: Sättigungsarithmetik. Kein Divisionsrest.
- SDIVcc-Operation des SPARC-V9: Sättigungsarithmetik, Setzen eines Flags.

Ganzzahlige Division im iAPX 86/88

Nicht immer wird der verfügbare Resultatbereich voll ausgeschöpft:

Der iAPX 86/88 benutzt Zahlenbereiche $\mathrm{Int}(2^{l-1}, 2^{l-1})$ mit $l = 8, 16, 32$.

Der Prozessor verfügt über Divisionsoperationen
`IDIV`: $\mathrm{Int}(2^{2\times l-1}, 2^{2\times l-1}) \times \mathrm{Int}(2^{l-1}, 2^{l-1}) \to \mathrm{Int}(2^{l-1}, 2^{l-1}) \times \mathrm{Int}(2^{l-1}, 2^{l-1})$
für $l = 8$ bzw. $l = 16$, die gleichzeitig einen ganzzahligen Quotienten und den dazu gehörenden Divisionsrest liefern; es wird die Vorschrift $\mathrm{sgn}(r) = \mathrm{sgn}(a)$ benutzt.

Liegt der zu berechnende Quotient nicht im Bereich $|q| < 2^{l-1}$, wird wie bei Division durch Null eine Unterbrechung ausgelöst; Quotient und Rest sind dann undefiniert.

Ein Quotient $q = -2^{l-1}$ kann also nicht berechnet werden, obwohl im Resultatbereich $\mathrm{Int}(2^{l-1}, 2^{l-1})$ vorhanden.

Nachfolgemodelle des iAPX 86/88 erlauben den vollen Resultatbereich $\mathrm{Int}(2^{l-1}, 2^{l-1})$.

Konversion $\mathrm{Int}(H', K') \to \mathrm{Int}(H, K)$

- $H' \leq H$ und $K' \leq K$:
  $\mathrm{conv}_{H',K',H,K}$ ist dann total und stimmt mit der Identität überein.

- $H' > H$ oder $K' > K$:
  In Überlaufarithmetik ist $\mathrm{conv}_{H',K',H,K}$ auf $\mathrm{Int}(H, K)$ wertverlaufsgleich mit der Identität, und sonst undefiniert (Überlauf).
  In zirkulärer Arithmetik gilt $\mathrm{conv}_{H',K',H,K}(a) = ((a + H) \bmod (H + K)) - H$.
  In Sättigungsarithmetik ist $\mathrm{conv}_{H',K',H,K}(a) = \max\{-H, \min\{K - 1, a\}\}$.

Ganzzahl-Arithmetik in Java

Typen, Zahlenbereiche und Repräsentation:

- `char`: $\mathrm{UInt}(2^{16})$ in kanonischer Radix-2-Repräsentation
- `byte`: $\mathrm{Int}(2^7, 2^7)$ in 2-Komplement-Darstellung
- `short`: $\mathrm{Int}(2^{15}, 2^{15})$ in 2-Komplement-Darstellung
- `int`: $\mathrm{Int}(2^{31}, 2^{31})$ in 2-Komplement-Darstellung
- `long`: $\mathrm{Int}(2^{63}, 2^{63})$ in 2-Komplement-Darstellung

Anpassung (numeric promotion) der Operanden arithmetischer Operationen:

- `char`, `byte`, `short` in unären Operationen wird zu `int` konvertiert
  (unary numeric promotion mit widening)
- In binären Operationen mit einem Operanden vom Typ `long` wird der andere ggf. zu `long` konvertiert (symmetrische Anpassung, binary numeric promotion mit widening)
- In allen anderen binären Operationen werden beide Operanden ggf. zu `int` konvertiert (symmetrische Anpassung, binary numeric promotion mit widening)

Durchführung arithmetischer Operationen:

- Operationen: $+$ (unär oder binär), $-$ (unär oder binär), $*$, $/$, %
- Zirkulär, ohne Überlaufinformation, Division durch Null erzwingt arithmetische Ausnahme
- Division bzw. Rest: $\mathrm{sgn}(r) = \mathrm{sgn}(a)$ sowie $((a \oslash b) \otimes b) \oplus (a \otimes_{H,K} b) = a$
- $+$ und $*$ nicht immer assoziativ, z. B. $(\texttt{long} + \texttt{int}) + \texttt{int} \stackrel{?}{=} \texttt{long} + (\texttt{int} + \texttt{int})$

Häufige Repräsentationen für Int(H, K)

- $\text{Int}_{VB}(l)$: $\text{Int}(2^{l-1}-1, 2^{l-1})$

  Eine Sequenz $(r_{l-1}, r_{l-2}, \ldots, r_1, r_0)$ mit $r_i \in \{0,1\}$ repräsentiert die ganze Zahl

  $$s = (-1)^{r_{l-1}} \times \sum_{i=0}^{l-2} r_i \times 2^i$$

  Repräsentation (schwach) redundant, da Null zwei Darstellungen besitzt.

- $\text{Int}_1(l)$: $\text{Int}(2^{l-1}-1, 2^{l-1})$

  Eine Sequenz $(r_{l-1}, r_{l-2}, \ldots, r_1, r_0)$ mit $r_i \in \{0,1\}$ repräsentiert die ganze Zahl

  $$s = (-1)^{r_{l-1}} \times \sum_{i=0}^{l-2} [r_{l-1} + (-1)^{r_{l-1}} \times r_i] \times 2^i = r_{l-1} \times (1 - 2^{l-1}) + \sum_{i=0}^{l-2} r_i \times 2^i$$

  Repräsentation (schwach) redundant, da Null zwei Darstellungen besitzt.

- $\text{Int}_2(l)$: $\text{Int}(2^{l-1}, 2^{l-1})$

  Eine Sequenz $(r_{l-1}, r_{l-2}, \ldots, r_1, r_0)$ mit $r_i \in \{0,1\}$ repräsentiert die ganze Zahl

  $$s = (-1)^{r_{l-1}} \times \{\sum_{i=0}^{l-2} [r_{l-1} + (-1)^{r_{l-1}} \times r_i] \times 2^i - r_{l-1}\} = -r_{l-1} \times 2^{l-1} + \sum_{i=0}^{l-2} r_i \times 2^i$$

  Wesentliches Problem: Asymmetrischer Zahlenbereich.

Weitere Repräsentationsschemata

In einem allgemeinen *Stellenwertsystem* repräsentiert eine Sequenz $(r_{l-1}, r_{l-2}, \ldots, r_1, r_0)$ mit $r_i \in S_i \subseteq S$ eine Zahl $s = \sum_{i=0}^{l-1} r_i \times b_i$ mit festen $b_i \in S$ und assoziativem Operator $+$ in der Struktur $(S, +, \times)$.

Sind $+$ und $\times$ monoton, so ist die Interpretationsfunktion bezüglich jeder Ziffer r_i monoton.

Häufig werden Systeme mit $b_i = R^{k+i}$, $R \in \mathbb{N}$, $R \geq 2$, $k \in \mathbb{Z}$ fest, verwendet (Zahlendarstellungen zur Basis R):

- Festkommazahlen: $S_i = [0, R-1] \cap \mathbb{Z}$
- Vorzeichenlose ganze Zahlen: $k = 0$, $S_i = [0, R-1] \cap \mathbb{Z}$ (besonders häufig für $R = 2, 10, 2^p$)
- Carry-Save-Darstellung: $k = 0$, $S_i = [0, R] \cap \mathbb{Z}$
- Signed-Digit-Darstellung: $k = 0$, $S_i = [-\alpha, \beta] \cap \mathbb{Z}$, $\alpha > 0$, $\beta > 0$, $\alpha + \beta \geq R$

Beispiele für weitere gebräuchliche Stellenwertsysteme

- Vorzeichenlose Festkommazahlen:
  $b_i = R^{k-i}$, $R \in \mathbb{N}$, $R \geq 2$, $k \in \mathbb{Z}$ fest, $S_i = [0, R-1] \cap \mathbb{Z}$
- 2-Komplement-Darstellung:
  $S_i = \{0, 1\}$, $b_{l-1} = -2^{l-1}$, $b_i = 2^i$ für $i = 0, \ldots, l-2$
- Allgemein: Basis-Komplement zur Basis R:
  $S_{l-1} = \{0, 1\}$, $b_{l-1} = -R^{l-1}$, $S_i = [0, R-1] \cap \mathbb{Z}$, $b_i = R^i$ für $i = 0, \ldots, l-2$
- 1-Komplement-Darstellung:
  $S_i = \{0, 1\}$, $b_{l-1} = 1 - 2^{l-1}$, $b_i = 2^i$ für $i = 0, \ldots, l-2$
- Allgemein: Vermindertes-Basis-Komplement zur Basis R:
  $S_{l-1} = \{0, 1\}$, $b_{l-1} = 1 - R^{l-1}$, $S_i = [0, R-1] \cap \mathbb{Z}$, $b_i = R^i$ für $i = 0, \ldots, l-2$
- Gemischtbasige Darstellung:
  $S_i = [0, \beta_i] \cap \mathbb{Z}$, $b_0 = 1$, $b_{i+1} = b_i \times (1 + \beta_i)$

NB: Die Vorzeichen/Betrag-Darstellung zur Basis R, $s = (-1)^{r_{l-1}} \times \sum_{i=0}^{l-2} r_i \times R^i$ mit $r_{l-1} \in \{0, 1\}$, ist kein Stellenwertsystem!

Residuen-Darstellung

$\mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_k}$, $m_i \in \mathbb{N}^*$, besitzt eine modulare Ringstruktur: Für $\circ \in \{+, -, \times\}$ gilt

$$x \circ y = (x_1, x_2, \ldots, x_k) \circ (y_1, y_2, \ldots, y_k) =$$

$$((x_1 \circ y_1) \bmod m_1, (x_2 \circ y_2) \bmod m_2, \ldots, (x_k \circ y_k) \bmod m_k)$$

Jeder Zahl $s \in \mathbb{Z}$ kann eindeutig das k-Tupel ($s \bmod m_1, s \bmod m_2, \ldots, s \bmod m_k$) zugeordnet werden.

Der Chinesische Restsatz zeigt, daß $M = \underset{i=1}{\overset{k}{\mathrm{kgV}}}\, m_i$ verschiedene Zahlen darstellbar sind.

Für $\mathrm{ggT}(m_i, m_j) = 1\ \forall i \neq j$ (relativ prime Moduli) gilt $M = \prod_{i=1}^{k} m_i$

Die Residuen-Darstellung ist kein Stellenwertsystem.

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Residuen-Arithmetik

Residuen-Arithmetik

Zweck der Residuen-Arithmetik

- Systeme zur Darstellung ganzer Zahlen.
- Schnelle Ausführung von Addition, Subtraktion, Multiplikation, Potenzierung.
- Fehlererkennung und -korrektur leicht zu implementieren.

Probleme der Residuen-Arithmetik

- Division, Größenvergleich, Vorzeichenbestimmung, Überlauferkennung, Skalierung, Konvertierung und Dekonvertierung sind sehr aufwändig.
- Die Residuen-Darstellung ist kein Stellenwertsystem, Mantissenangleich muss durch echte Multiplikation realisiert werden.
- Reelle Zahlen sind generell schlecht repräsentierbar.

Residuen-Arithmetik ist interessant, wenn eine große Anzahl von Additionen, Subtraktionen, Multiplikationen oder Potenzierungen einer kleinen Anzahl übriger Operationen gegenübersteht (z. B. in der Signalverarbeitung, in digitalen Filtern, Fourier-Transformation, Krypto-Equipment).

Grundidee der Residuen-Arithmetik

$\mathbb{Z}_{m_1} \times \mathbb{Z}_{m_2} \times \cdots \times \mathbb{Z}_{m_k}$, $m_i \in \mathbb{N}^*$, besitzt eine modulare Ringstruktur: Für $\circ \in \{+, -, \times\}$ gilt

$$x \circ y = (x_1, x_2, \ldots, x_k) \circ (y_1, y_2, \ldots, y_k) =$$

$$((x_1 \circ y_1) \bmod m_1, (x_2 \circ y_2) \bmod m_2, \ldots, (x_k \circ y_k) \bmod m_k)$$

Jeder Zahl $s \in \mathbb{Z}$ wird eindeutig das k-Tupel ($s \bmod m_1, s \bmod m_2, \ldots, s \bmod m_k$) zugeordnet. Die m_i heißen *Moduli*, die Werte $s_i \bmod m_i$ *Residuen*.

Der Chinesische Restsatz zeigt, dass damit $M = \operatorname{kgV}_{i=1}^{k} m_i$ beliebige, aufeinander folgende ganze Zahlen darstellbar sind.

Ein derartiges Residuensystem heiße daher im Folgenden ein *M-Residuensystem*.

Für $\operatorname{ggT}(m_i, m_j) = 1 \; \forall i \neq j$ (relativ prime Moduli, Annahme im Folgenden) gilt $M = \prod_{i=1}^{k} m_i$.

Statt „großer“ Zahlen (Größenordnung M) müssen nur „kleine“ Zahlen (Größenordnung m_i) verarbeitet werden.

Zahlenbereiche eines M-Residuensystems

Originärer Zahlenbereich: UInt(M)

Darstellung im Basis-Komplement: Int(H, K) mit $M = H + K$
Jede negative Zahl $-s$ wird dabei eindeutig durch $M - s$ dargestellt.
Residuen für $-s$ folgen aus Residuen für s: $(-x)_i = m_i - x_i$ für $x_i \neq 0$, $(-x)_i = 0$ für $x_i = 0$
Ein symmetrischer Zahlenbereich, $H = K - 1$, ist nur möglich, wenn alle Moduli ungerade sind.
An der üblichen Basis-Komplement-Darstellung angelehnte Zahlenbereiche, $H = K$, lassen sich realisieren, wenn (mindestens) ein Modul gerade ist.

Darstellung im verminderten Basis-Komplement: Int(H, K) mit $M = H + K + 1$
Negative Zahlen $-s$ eindeutig durch $(M - 1) - s$ dargestellt, Null redundant durch 0 und $M - 1$.
Residuen für $-s$ folgen aus Residuen für s: $(-x)_i = (m_i - 1) - x_i$
An der üblichen Basis-Komplement-Darstellung angelehnte Zahlenbereiche, $H = K$, sind nur möglich, wenn alle Moduli ungerade sind.
Symmetrische Zahlenbereiche, $H = K - 1$, realisierbar, wenn (mindestens) ein Modul gerade ist.

Vorzeichen/Betrag-Darstellungen sind schlecht verträglich mit Residuensystemen,
da das Vorzeichen zwar multiplikativ, aber nicht additiv ist.
Hilfsweise kann einer vorzeichenlosen Residuendarstellung ein separates Vorzeichenbit beigefügt werden, führt zu Int($M - 1, M$); die Moduli brauchen dann auch keiner Zusatzbedingung genügen.

Arithmetische Operationen auf einem Residuensystem

Addition, Subtraktion, Multiplikation und Negation erfolgen für jeden Modul getrennt (überlauffrei).

Dies ist die eigentliche (und meist einzige) Stärke von Residuensystemen.

Besagte Operationen können für kleine Moduli durch Wertetabellen implementiert werden.

Andernfalls werden die Residuen durch gewöhnliche Arithmetik verarbeitet;
Probleme bereitet dabei evtl. die abschließende Reduktion modulo m_i.

Für Moduli R^a kann direkt zirkuläre vorzeichenlose Radix-R-Ganzzahlarithmetik benutzt werden.

Negation im (verminderten) Basis-Komplement erfolgt ziffernweise wie zuvor beschrieben.

Die Subtraktion wird häufig auf Addition nach Negation zurückgeführt.

Gilt $x \times y \equiv 1 \pmod{M}$, so heißt y ein multiplikatives Inverses von x.
Multiplikative Inverse existieren nur für $\mathrm{ggT}(x, M) = 1$ und sind dann eindeutig.

Ganzzahlige Division mit Rest ist im Residuensystem nur unter großem Aufwand durchführbar.

Redundante Darstellung von Residuen

Ist m_i keine Potenz der Basis R, werden für die Residuen statt der ganzen Zahlen aus $[0, m_i - 1]$ häufig alle ganzen Zahlen aus $[0, R^{\lceil \log_R m_i \rceil} - 1]$ zugelassen.

Einige Residuen besitzen dann zwei Darstellungen.

Diese Technik vereinfacht die Implementierung von Addition und Multiplikation modulo m_i.

Beispiel

$m_i = R^a - 1$, das Residuum 0 besitzt die Darstellungen 0 und $R^a - 1$.

Nach einer gewöhlichen a-Ziffern-Addition wird das Übertragsbit zum Ergebnis addiert:

$x + y \equiv (x + y) \bmod R^a + R^a \times \lfloor (x + y)/R^a \rfloor \equiv (x + y) \bmod R^a + \lfloor (x + y)/R^a \rfloor \pmod{R^a - 1}$.

Naheliegende Implementierung: Addierer mit EAC-Technik (End-around-Carry).

Entsprechend die Multiplikation:
$x \times y \equiv (x \times y) \bmod R^a + R^a \times \lfloor (x \times y)/R^a \rfloor \equiv (x \times y) \bmod R^a + \lfloor (x \times y)/R^a \rfloor \pmod{R^a - 1}$.
Gewinnung der Operanden als Teilergebnisse einer doppelt genauen Multiplikation,
häufig zusätzlich EAC-Technik nötig, Beispiel $(R^a - R) \times (R^a - R)$ mit $a > 2$

Für Berechnungen modulo $R^a + 1$ lässt sich nutzen:
$z \equiv z \bmod R^a + R^a \times \lfloor z/R^a \rfloor \equiv z \bmod R^a - \lfloor z/R^a \rfloor \pmod{R^a + 1}$

Konvertierung aus einer Standarddarstellung in ein Residuensystem

Als Konstante, Eingabe oder skalares Zwischenergebnis anfallende Werte müssen in entsprechendes Tupel von Residuen umgerechnet werden:

Sei $\sum_{j=0}^{l-1} b_j \times R^j$ eine Stellenwertdarstellung der Länge l zur Basis R mit den Ziffern b_j.

Die Residuen werden mittels $x_i = (\sum_{j=0}^{l-1} b_j \times R^j) \bmod m_i = (\sum_{j=0}^{l-1} b_{ij} \times z_{ij}) \bmod m_i$ berechnet,

wobei die Gewichte $z_{ij} = R^j \bmod m_i$ ganzzahlig, klein und a priori bekannt sind,
die Faktoren $b_{ij} = b_j \bmod m_i$ ganzzahlig, klein und meist leicht herstellbar sind.

Die Berechnungen $(\sum_{j=0}^{l-1} b_{ij} \times z_{ij}) \bmod m_i$ können parallel und insbesondere bereits selbst in Residuen-Arithmetik ausgeführt werden.

Die Anzahl zu addierender Terme sinkt mit wachsender Basis R.

Anpassung auf vorzeichenbehaftete Zahlen in Stellenwertsystem durch andere Gewichte z_{ij}.

Konversion für spezielle Moduli

Für spezielle Moduli vereinfacht sich die Konversion erheblich. Beispiel: $m_i = R^a - 1$.

Wegen $R^a \equiv 1 \pmod{R^a - 1}$ gilt $R^{k+a\times h} \equiv R^k \times (R^a)^h \equiv R^k \pmod{R^a - 1}$ und damit

$$\sum_{j=0}^{r\times a-1} b_j \times R^j \equiv \sum_{h=0}^{r-1}\sum_{k=0}^{a-1} b_{k+a\times h} \times R^{k+a\times h} \equiv \sum_{h=0}^{r-1}\sum_{k=0}^{a-1} b_{k+a\times h} \times R^k \pmod{R^a - 1}$$

Es genügt also, Blöcke von jeweils a Ziffern der Eingabe modulo $(R^a - 1)$ zu addieren.

Ähnlich $m_i = R^a + 1$:

Wegen $R^a \equiv -1 \pmod{R^a + 1}$ gilt $R^{k+a\times h} \equiv R^k \times (R^a)^h \equiv (-1)^h \times R^k \pmod{R^a + 1}$ und damit

$$\sum_{j=0}^{r\times a-1} b_j \times R^j \equiv \sum_{h=0}^{r-1}\sum_{k=0}^{a-1} b_{k+a\times h} \times R^{k+a\times h} \equiv \sum_{h=0}^{r-1}(-1)^h \times \sum_{k=0}^{a-1} b_{k+a\times h} \times R^k \pmod{R^a + 1}$$

Blöcke von jeweils a Ziffern der Eingabe werden hier abwechselnd modulo $(R^a + 1)$ addiert bzw. subtrahiert.

Trivial dagegen der Fall $m_i = R^a$, es werden nur die niederwertigsten a Ziffern berücksichtigt:

$$\sum_{j=0}^{r\times a-1} b_j \times R^j \equiv \sum_{k=0}^{a-1} b_k \times R^k \pmod{R^a - 1}$$

Konvertierung aus einem Residuensystem in Standarddarstellung

Die Konvertierung erfolgt durch Berechnung des Ausdrucks $X = (\sum_{i=1}^{k} h_i \times x_i) \bmod M$,

wobei die Faktoren h_i ganzzahlig und a priori bekannt sind.

Es gilt insbesondere $(M/m_i)|h_i$ und $h_i \equiv 1 \pmod{m_i}$.

Beispiel
Zu den Moduli $m_i = 3, 5, 7, 8$ gehören die Faktoren $h_i = 280, 336, 120, 105$.

Wegen ggT$(\frac{M}{m_i}, m_i) = 1$ existiert $\alpha_i \in \mathbb{N}$ mit $\alpha_i < m_i$ und $\frac{M}{m_i} \times \alpha_i \equiv 1 \pmod{m_i}$.

Es gilt alternativ die Formel $X = (\sum_{i=1}^{k} \frac{M}{m_i} \times \gamma_i) \bmod M$ mit $\gamma_i = (\alpha_i \times x_i) \bmod m_i$.

Die Werte von γ_i werden mittels Residuenmultiplikation berechnet,

die Vielfachen $\frac{M}{m_i} \times \gamma_i$ aus einer Wertetabelle entnommen.

Konvertierung in ein gemischtbasiges Stellenwertsystem

Die durch ein k-Tupel von Residuen repräsentierte Zahl kann, statt in Standarddarstellung, auch eindeutig in einem gemischtbasigen Stellenwertsystem dargestellt werden:

$$X = a_k \times (m_{k-1} \times \cdots \times m_1) + \cdots + a_3 \times (m_2 \times m_1) + a_2 \times m_1 + a_1 \text{ mit } 0 \leq a_i < m_i$$

Die Konversion ist selbst in Residuenarithmetik ausführbar:

$$\begin{aligned} Y_1 &= X \\ a_i &= Y_i \bmod m_i \quad \text{(meist gleich } y_{ii} \text{ bzw. realisiert durch } y_{ii} \bmod m_i) \\ Y_{i+1} &= (Y_i - a_i) \times T_i \quad \text{(spezielle Skalierung, bewirkt Division durch } m_i) \end{aligned}$$

mit $T_i = (t_{i1}, \ldots, t_{ik})$, wobei $m_i \times t_{ij} \equiv 1 \pmod{m_j}\ \forall j \neq i$ (erfordert $\text{ggT}(m_i, m_j) = 1$).

Die Faktoren t_{ij} sind konstant und können deshalb vorab berechnet werden.

Anmerkung: y_{ij} ist irrelevant für $j < i$.

Es bietet sich an, die m_i aufsteigend zu ordnen, damit $a_i < m_j\ \forall j > i$;
ansonsten wird vor der Subtraktion eine Reduktion $a_i \bmod m_j$ nötig.

Anwendungen: Größenvergleich, Überlauferkennung, Vorzeichenbestimmung

Residuen-Arithmetik: Vergleichsoperationen

- Test auf Gleichheit erfolgt ziffernweise, in Systemen mit redundanter Darstellung der Residuen nach Entfernung der Redundanz durch Reduktion modulo m_i.
- Residuen nicht lexikalisch geordnet (kein Stellenwertsystem), Größenvergleich aufwendig
- Vorzeichen nicht direkt ablesbar (außer in künstlicher V/B-Darstellung)
- Größenvergleich vorzeichenloser Residuenzahlen kann lexikalisch im gemischtbasigen Stellenwertsystem erfolgen, da dieses nicht redundant ist.
- Vorzeichenbestimmung erfolgt z. B. durch Vergleich mit der größten darstellbaren Zahl.
- Größenvergleich vorzeichenbehafteter Residuenzahlen erfolgt z. B. durch Vorzeichentest, bei gleichem Vorzeichen durch Größenvergleich für vorzeichenlose Residuenzahlen.

Alternativ wird zum Vergleich eine Approximation von $\frac{X}{M} = (\sum_{i=1}^{k} \frac{\gamma_i}{m_i}) \bmod 1$ bestimmt.

Die approximierten Terme $\frac{\gamma_i}{m_i}$ werden aus einer Wertetabelle abgelesen,

die Reduktion modulo 1 bedeutet, dass nur Nachkommateile betrachtet werden.

Die Tabelleneinträge müssen so genau sein, dass anhand der Approximation zuverlässig entschieden werden kann; Fehler kleiner als $1/(2 \times k \times M)$ pro Eintrag genügen hierfür.

Weniger Genauigkeit reicht aus, wenn der Test nicht exakt zu sein braucht, z. B. in einer SRT-Division; SD-Quotientenziffern werden on-the-fly in Residuen umgewandelt.

Residuen-Arithmetik: Überlauf- und Fehlererkennung

Zur Überlauferkennung wird die Menge der Moduli $(m_1|\cdots|m_k)$ so zu einer Menge von Moduli $(m_1|\cdots|m_k|m_{k+1}|\cdots|m_r)$ erweitert, dass alle Ergebnisse in dem durch $M' = \operatorname{kgV}_{i=1}^{r} m_i$ festgelegten Bereich eindeutig sind.

Ein Überlauf liegt genau dann vor, wenn nach Konvertierung in das gemischtbasige Stellenwertsystem mit r Ziffern $a_i \neq 0$ für mindestens ein $i > k$.

Generell ist es von Vorteil, wenn der Ergebnisbereich a priori bekannt ist, da dann die prinzipiell aufwendige Überlauferkennung vermieden werden kann.

Zur Aufzudeckung von Übertragungsfehlern wird zu den Moduli $(m_1|\cdots|m_k)$ ein weiterer Modul $m_{k+1} > m_i \; \forall i \leq k$ hinzugefügt und mit $k+1$ Residuen gerechnet.

Ist nur ein Residuum von einem Fehler betroffen, so unterscheidet sich x_{k+1} von dem durch *Basiserweiterung* aus den Residuen $x_1, \ldots, x_k$ direkt berechenbaren Wert X mod m_{k+1}.

Die Verwendung weiterer zusätzlicher Moduli erlaubt die Erkennung von mehr als einem Fehler bzw. die Identifizierung der fehlerhaften Residuen und ihre anschließende Korrektur.

Residuen-Arithmetik: Aufwand für die Darstellung

Zur Darstellung von m_i Werten benötigen wir $\lceil \log_2 m_i \rceil$ Bit.

Insgesamt benötigen wir also $\sum_{i=1}^{k} \lceil \log_2 m_i \rceil$ statt $\lceil \log_2 M \rceil$ Bit.

Beispiel

Mit den Moduli (3|5|7|8) benötigen wir 11 Bit, um 840 Zahlen darzustellen.

In Binärcodierung bräuchten wir für 840 Zahlen nur 10 Bit.

Umgekehrt könnten wir mit 11 Bit sogar 2048 Werte darstellen.

Residuen-Arithmetik: Wahl der Moduli

- Kleine Moduli begünstigen tendenziell die einfache und schnelle Implementierung von Addition, Subtraktion, Multiplikation und Restbildung.
- Die Anzahl der Schritte bei Konvertierung aus dem Residuensystem entspricht der Anzahl der Moduli.
- Moduli von etwa gleicher Größenordnung minimieren tendenziell den größten Modul.
- Der Speichermehraufwand der Repräsentation wächst etwa mit $\lceil \log_2 m_i \rceil - \log_2 m_i$.
- Die Komplexität der Operationen ist nicht monoton in den Moduli (außer evtl. stückweise bei Verwendung von Wertetabellen).
- Wegen einfach durchzuführender arithmetischer Operationen sind Moduli der Form $(2^a - 1)$, 2^a (nur für einen Modul, meist den größten) sowie gelegentlich $(2^a + 1)$ besonders beliebt.

Nützlich für die Konstruktion ist die Beziehung $\mathrm{ggT}(a, b) = 1 \Leftrightarrow \mathrm{ggT}(2^a - 1, 2^b - 1) = 1$.

Anwendung:

Sind $a_1, a_2, \ldots, a_{k-1}$ relativ prim und a_{k-1} maximal, so benötigt das Residuensystem $(2^{a_1} - 1 | 2^{a_2} - 1 | \ldots | 2^{a_{k-1}} - 1 | 2^{a_{k-1}})$ höchstens ein Bit mehr als die Binärdarstellung.

Residuen-Arithmetik: Wahl der Moduli (Fallstudie)

Repräsentation von Zahlen des Bereichs [0..99.999] mit 17 Bit:

m_i	M	Bit	
(2|3|5|7|11|13|17)	510510	22	Die sieben kleinsten Primzahlen
(2|3|7|11|13|17)	102102	19	Bereichsreduktion unschädlich
(11|17|21|26)	102102	19	Anzahl der Moduli reduziert
(13|17|21|22)	102102	19	Moduli annähernd gleich groß
(5|7|8|9|11|13)	360360	21	Potenzen von Primzahlen zulassen
(3|5|7|8|11|13)	120120	19	Bereichsreduktion unschädlich
(7|8|11|13|15)	120120	18	Speicheraufwand verringert
(5|7|9|11|13|16)	720720	22	Größter Modul Potenz der Basis
(5|9|11|13|16)	102960	19	Bereichsreduktion unschädlich
(7|9|11|13|16)	144144	19	Spezielle Form eines Moduls
(7|15|31|32)	104160	17	Speicheraufwand minimal

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Fest- und Gleitkommasysteme

Gleitkommazahlen: Allgemeiner Zahlenbereich

Allgemeiner Zahlenbereich für Gleitkommazahlen (halblogarithmische Darstellung)

$\{s \times R^e | R \in \mathbb{N}, R > 1, (s, e) \in W \subset \mathbb{Z} \times \mathbb{Z}\}$

s heißt *Signifikant*, e *Exponent*, R *Basis*.

Spezieller: reguläre Kombination von Signifikanten und Mantissen

$\{s \times R^e | R \in \mathbb{N}, R > 1, s \in S \subset \mathbb{Z}, e \in E \subset \mathbb{Z}\}$

Zusätzlich: Intervallbereiche für Signifikanten und Mantissen

$\{s \times R^e | R \in \mathbb{N}, R > 1, s \in \mathbb{Z}, s = 0 \vee s^+_{\min} \leq s \leq s^+_{\max} \vee s^-_{\min} \leq -s \leq s^-_{\max}, e \in \mathbb{Z}, e_{\min} \leq e \leq e_{\max}\}$

Symmetrische Zahlenbereiche: $s^+_{\min} = s^-_{\min}, s^+_{\max} = s^-_{\max}$

Historisch: auch (leicht) unsymmetrische Zahlenbereiche verwendet

Basis R bestimmt den dynamischen Bereich, ist fest, braucht also nicht gespeichert zu werden

R meist 2, ergänzend auch 10 (iAPX87),

seltener 8 (Manchester University Atlas, 1962; Burroughs B5500, 1964)

oder 16 (IBM System/360-370, 1964/1970; Manchester University MU5, 1972; HEP, 1982),

andere Werte nur in Ausnahmefällen, z. B. 256 (MANIAC II, Los Alamos, 1956)

Typische Gleitkommasysteme mit symmetrischem Zahlenbereich

Heutzutage dominierende Gleitkommasysteme (IEEE-754, JAVA, ...):

System normalisierter Gleitkommazahlen $\quad \mathrm{Float}(R, l, e_1, e_2) =$

$$\{0\} \cup \{v \times m \times R^e | v \in \{-1, 1\}, m \in [R^{l-1}, R^l - 1] \cap \mathbb{N}, e \in [e_1 - l, e_2 - l] \cap \mathbb{Z}\}$$

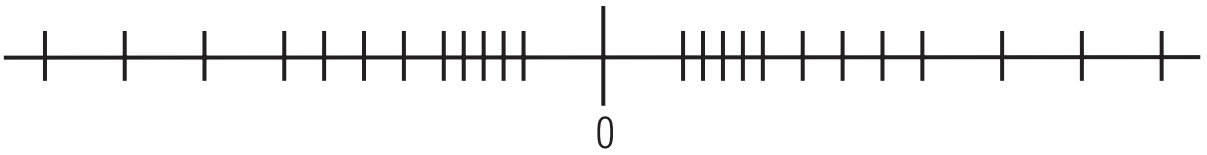
Erweitertes Gleitkommasystem $\quad \mathrm{Float}_e(R, l, e_1, e_2) =$

$$\{v \times m \times R^e | v \in \{-1, 1\}, m \in [0, R^l - 1] \cap \mathbb{N}, e \in [e_1 - l, e_2 - l] \cap \mathbb{Z}\}$$

Der Signifikant ist faktorisiert in das *Vorzeichen* v und die *Magnitude* m.

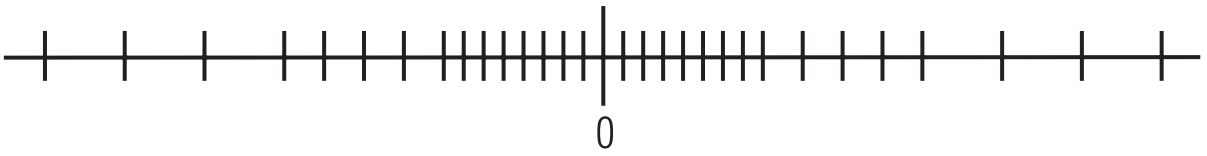
System normalisierter Gleitkommazahlen: Float(2, 3, 0, 2)

e \ m	0	1	2	3	4	5	6	7
-3	0				$\frac{4}{8}$	$\frac{5}{8}$	$\frac{6}{8}$	$\frac{7}{8}$
-2					$\frac{8}{8}$	$\frac{10}{8}$	$\frac{12}{8}$	$\frac{14}{8}$
-1					$\frac{16}{8}$	$\frac{20}{8}$	$\frac{24}{8}$	$\frac{28}{8}$



Erweitertes Gleitkommasystem: $\text{Float}_e(2, 3, 0, 2)$

e \ m	0	1	2	3	4	5	6	7
-3	0	$\frac{1}{8}$	$\frac{2}{8}$	$\frac{3}{8}$	$\frac{4}{8}$	$\frac{5}{8}$	$\frac{6}{8}$	$\frac{7}{8}$
-2	0	$\frac{2}{8}$	$\frac{4}{8}$	$\frac{6}{8}$	$\frac{8}{8}$	$\frac{10}{8}$	$\frac{12}{8}$	$\frac{14}{8}$
-1	0	$\frac{4}{8}$	$\frac{8}{8}$	$\frac{12}{8}$	$\frac{16}{8}$	$\frac{20}{8}$	$\frac{24}{8}$	$\frac{28}{8}$



Repräsentationen für Gleitkommazahlen

Es existiert eine Vielzahl unterschiedlicher Formate:

Signifikant und Exponent können jeweils in Vorzeichen-Betrag-Darstellung, Basis-Komplement oder vermindertem Basis-Komplement vorliegen; dies ergibt 9 verschiedene Grundformen.

Bei Vorzeichen-Betrag-Darstellung können Vorzeichen und Betrag jeweils separat oder in einem Feld zusammenhängend gespeichert werden.
In seltenen Fällen wurde auch das Vorzeichen von Komplement-Darstellungen abgetrennt.

Die verschiedenen Teile der Darstellung können beliebig angeordnet werden.

Die Basis R_s für den Signifikanten kann von der Basis R_e für den Exponenten abweichen, beide können wiederum von R verschieden sein (in der Praxis ist allerdings meist $R_e = 2$).

In Komplement-Darstellung vorliegende Exponenten können mit einem Bias versehen werden.

Schließlich sind noch Darstellungen für nicht normalisierte Werte (z. B. 0, ∞, $-\infty$) zu wählen.
In der Rechnerarithmetik wird häufig mit symbolischen unendlichen Elementen $\pm\infty$ operiert.
Der *Abschluss* einer Menge M ist definiert durch $M^{\pm\infty} := M \cup \{-\infty, +\infty\}$.

Vorzeichen-Betrag-Darstellung des Signifikanten

In der Vorzeichen-Betrag-Darstellung des Signifikanten wird das Vorzeichen $v \in \{-1, +1\}$ üblicherweise durch ein *Vorzeichenbit* dargestellt, mit der Codierung $0 \mathrel{\hat{=}} +1$ und $1 \mathrel{\hat{=}} -1$.

Eine Magnitude $m \in [R_s^{l-1}, R_s^l - 1] \cap \mathbb{N}$ wird codiert als Ziffernfolge $m_1\, m_2 \ldots m_l$ mit der Bedeutung $m = \sum_{i=0}^{l-1} m_{l-i} \times R_s^i$.

Für $R_s = 2$ kann wegen $m_1 = 1$ die Ziffer m_1 auch implizit sein, d. h. sie wird dann nicht in das Speicherformat aufgenommen, sondern bei der Verarbeitung der Zahlen je nach Bedarf ergänzt (*Hidden-Bit*, z. B. in IEEE-754, DEC/VAX).

Darstellung des Exponenten

Für den Exponenten $e \in \mathbb{Z}$ gibt es eine Reihe verschiedener Codierungen mit den unterschiedlichsten Eigenschaften und Intentionen:

Häufig wird zu Exponenten in einer Komplement-Darstellung ein sogenannter *Bias* addiert; dies ist eine positive Zahl mit der Eigenschaft, dass das Ergebnis der Addition nichtnegativ (in manchen Zahlensystemen auch echt positiv) ist.

Der Bias kann bei Bedarf so gewählt werden, dass unterhalb und/oder oberhalb der eigentlichen Darstellungen von Exponenten einige unbenutzte Werte auftreten, die der Kennzeichnung von Null, ∞, $-\infty$ oder dem Auftreten eines arithmetischen Fehlers, Über- oder Unterlaufs dienen.

Werden Anordnung und Darstellung der verschiedenen Teile einer Maschinenzahl sorgfältig aufeinander abgestimmt, lässt sich auf den Gleitkommazahlen ein arithmetischer Größenvergleich durch lexikalischen Vergleich (wie bei ganzen Zahlen im R-Komplement) durchführen.

Die Null kommt besonders häufig als Operand in Testoperationen vor und sollte deshalb eine leicht zu testende Darstellung besitzen (z. B. nur aus Null-Bits bestehen).

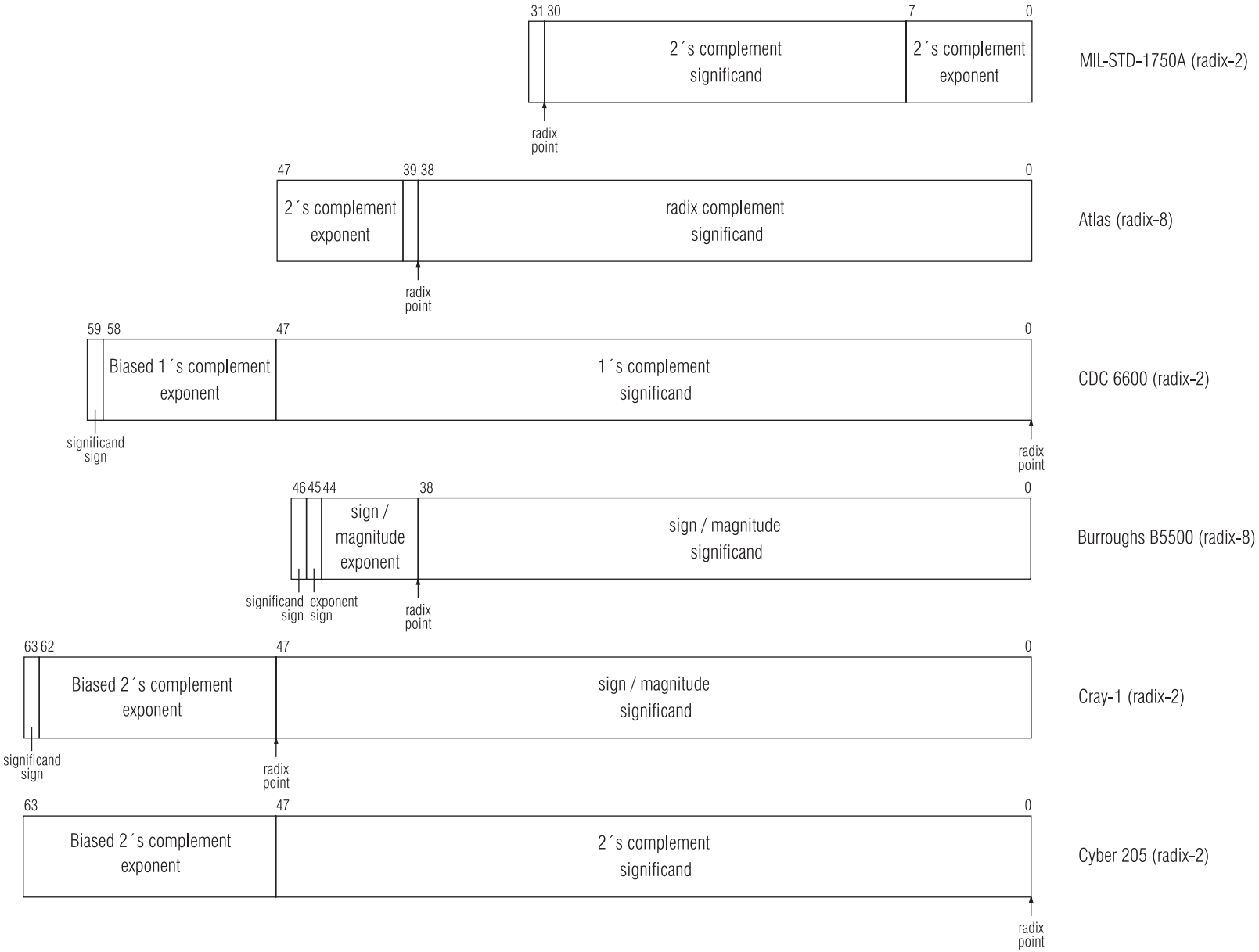
Darstellung erweiterter Gleitkommasysteme

Die Darstellung von Zahlen aus $\text{Float}_e(R, l, e1, e2)$ erfolgt im Prinzip nach dem gleichen Schema wie die der Zahlen aus $\text{Float}(R, l, e1, e2)$.

Abweichungen bestehen in folgenden Punkten:

- Für die Null ist in $\text{Float}_e(R, l, e1, e2)$ keine Sonderbehandlung nötig.
  Aus Gründen der Verträglichkeit wird die Null in $\text{Float}_e(R, l, e1, e2)$
  jedoch häufig genauso dargestellt wie in $\text{Float}(R, l, e1, e2)$.
- Die Hidden-Bit-Technik lässt sich hier nur nutzen, wenn normalisierte und denormalisierte Darstellungen unterscheidbar sind — etwa anhand ihrer abgespeicherten Exponenten.
  Für denormalisierte Zahlen gilt $m_1 = 0$, sodass für diese Zahlen ein Hidden-Bit
  mit dem Wert 0 benutzt werden kann.

Repräsentation von Gleitkommazahlen: Historische Beispiele



Festkommazahlen

Festkommasystem kann als spezielle Variante eines Gleitkommasystems gedeutet werden:

Zahlen $s \times R^e$

Basis R und Exponent e fest, Signifikant s Ganzzahl zur Basis R
R^e kann als Skalierungsfaktor gedeutet werden

$e > 0$ ist eher ungewöhnlich, $e = 0$ ergibt die Ganzzahlen, $e < 0$ typisch

Häufig $|e|$ klein (Währungen, Messungen, Anteile)
oder, wenn Mantissen l Stellen besitzen, $e = -l$ (Bruchteil) bzw. $e = 1 - l$

Festkommazahlen mit Nachkomma-Anteil

Spezifische Probleme, wenn $e < 0$:

Addition/Subtraktion von Festkommazahlen: $s \times R^e = s_1 \times R^e \pm s_2 \times R^e = (s_1 \pm s_2) \times R^e$
Möglichkeit des Überlaufs, wie bei Ganzzahlen; kein Genauigkeitsverlust

Multiplikation mit Ganzzahl: $s \times R^e = (s_1 \times R^e) \times s_2 = (s_1 \times s_2) \times R^e$
Möglichkeit des Überlaufs, wie bei Ganzzahlen; kein Genauigkeitsverlust

Multiplikation von Festkommazahlen: $s \times R^e = (s_1 \times R^e) \times (s_2 \times R^e) = (s_1 \times s_2 \times R^e) \times R^e$
$s = s_1 \times s_2 \times R^e$ im Allgemeinen keine Ganzzahl, Rundung nötig

Soll die Mantisse des Ergebnisses l Stellen besitzen und weist die Mantisse des Zwischenergebnisses der Ganzzahlmultiplikation l' Stellen auf, entstehen folgende Situationen:

$l' - e = l$: Rundung auf l Stellen; kein Überlauf möglich
$l' - e > l$: Rundung auf l Stellen; Überlauf möglich
$l' - e < l$: Rundung auf $l' - e$ Stellen, Genauigkeitsverlust; kein Überlauf möglich

Division durch Ganzzahl: $s \times R^e = (s_1 \times R^e)/s_2 = (s_1/s_2) \times R^e$
$s = s_1/s_2$ besitzt im Allgemeinen keine endliche Darstellung zur Basis R
Rundung nötig; kein Überlauf möglich

Division von Festkommazahlen: $s \times R^e = (s_1 \times R^e)/(s_2 \times R^e) = ((s_1/s_2) \times R^{-e}) \times R^e$
$s = (s_1/s_2) \times R^{-e}$ besitzt im Allgemeinen keine endliche Darstellung zur Basis R
Rundung nötig; Überlauf möglich

Anomalien

Jedes Gleitkommasystem ist endliche Approximation von $\mathbb{Q}$ bzw. $\mathbb{R}$

- Massiver Verlust der Abgeschlossenheit
- Massiver Verlust der Assoziativität der Addition
- Massiver Verlust der Assoziativität der Multiplikation
- Massiver Verlust der Distributivität
- Massives Fehlen multiplikativer Inverser

Bei echten Festkommasystemen (mit Nachkomma-Anteil) ähnlich

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Rundung

Rundung

Ist $S \subset \mathbb{R}$ ein Zahlenbereich, dann heißt eine Abbildung

$$\square\colon \mathbb{R} \to S$$

eine *Rundung*, wenn gilt:

(R1) $\quad \forall a \in S\colon \square a = a$

Die Rundung heißt *monoton*, wenn zusätzlich zu (R1) gilt:

(R2) $\quad \forall a, b \in \mathbb{R}\colon a \leq b \Rightarrow \square a \leq \square b$

Die Rundung heißt *antisymmetrisch*, wenn zusätzlich zu (R1) gilt:

(R4) $\quad \forall a \in \mathbb{R}\colon \square(-a) = -\square a$

Aus (R1) und (R4) folgt im Übrigen die *Symmetrie* von S: $-a \in S\ \forall a \in S$.

Semimorphismen

Ist $S \subset \mathbb{R}$ ein Zahlenbereich, so heißt eine monotone, antisymmetrische Rundung $\square\colon \mathbb{R} \to S$ ein *Semimorphismus*, wenn alle inneren (und äußeren) Verknüpfungen in S definiert sind durch die entsprechenden inneren (und äußeren) Verknüpfungen in $\mathbb{R}$:

(RG) $\quad \forall a, b \in S$ (bzw. $a \in S, b \in T$ oder $a \in T, b \in S$) $: a \boxdot b = \square(a \circ b)$

(RG), (R1) und (R2) bewirken, dass Operationen *maximal genau* sind, d. h. zwischen exaktem Ergebnis und Approximation einer Operation keine weitere Maschinenzahl liegt.

Ist $\square\colon \mathbb{R} \to S$ ein Semimorphismus und $1 \in S$, so bestehen folgende *Verträglichkeitsbeziehungen* zwischen $(\mathbb{R}, +, \times, \leq)$ und $(S, \boxplus, \boxtimes, \leq)$:

(RG1) $\quad \forall a, b \in S\colon a \circ b \in S \Rightarrow a \boxdot b = a \circ b$

(RG2) $\quad \forall a, b, c, d \in S\colon a \circ b \leq c \circ d \Rightarrow a \boxdot b \leq c \boxdot d$

(RG4) $\quad \forall a \in S\colon \boxminus a := (-1) \boxtimes a = -a$

Gerichtete Rundungen

Um gerichtete Rundungen einführen zu können, muss die Menge $\mathbb{R}$ beidseitig mit einem Abschluss versehen und der Begriff der monotonen Rundung über dieser abgeschlossenen Menge $\mathbb{R}^{\pm\infty}$ verstanden werden.

Seien $S \subset \mathbb{R}^{\pm\infty}$ ein Zahlenbereich mit $\pm\infty \in S$, $\bigtriangledown : \mathbb{R}^{\pm\infty} \to S$ und $\bigtriangleup : \mathbb{R}^{\pm\infty} \to S$ monotone Rundungen (die sogenannten *gerichteten Rundungen*) mit

(R3) $\quad \forall a \in \mathbb{R}^{\pm\infty} : \ \bigtriangledown a \leq a \leq \bigtriangleup a$

Dann sind $\bigtriangledown$ (*Rundung nach unten*) und $\bigtriangleup$ (*Rundung nach oben*) wohldefiniert.

Es gilt $\forall a, b \in \mathbb{R}^{\pm\infty} : \ \bigtriangledown a < b < \bigtriangleup a \Rightarrow b \notin S$

Seien S, D Zahlenbereiche mit $\pm\infty \in S \subset D \subset \mathbb{R}^{\pm\infty}$.

Seien $\bigtriangledown : \mathbb{R}^{\pm\infty} \to S$, $\bigtriangledown_1 : \mathbb{R}^{\pm\infty} \to D$, $\bigtriangledown_2 : D \to S$ Rundungen nach unten.
Dann gilt $\forall a \in \mathbb{R}^{\pm\infty} : \ \bigtriangledown a = \bigtriangledown_2 \bigtriangledown_1 a$.

Seien $\bigtriangleup : \mathbb{R}^{\pm\infty} \to S$, $\bigtriangleup_1 : \mathbb{R}^{\pm\infty} \to D$, $\bigtriangleup_2 : D \to S$ Rundungen nach oben.
Dann gilt $\forall a \in \mathbb{R}^{\pm\infty} : \ \bigtriangleup a = \bigtriangleup_2 \bigtriangleup_1 a$.

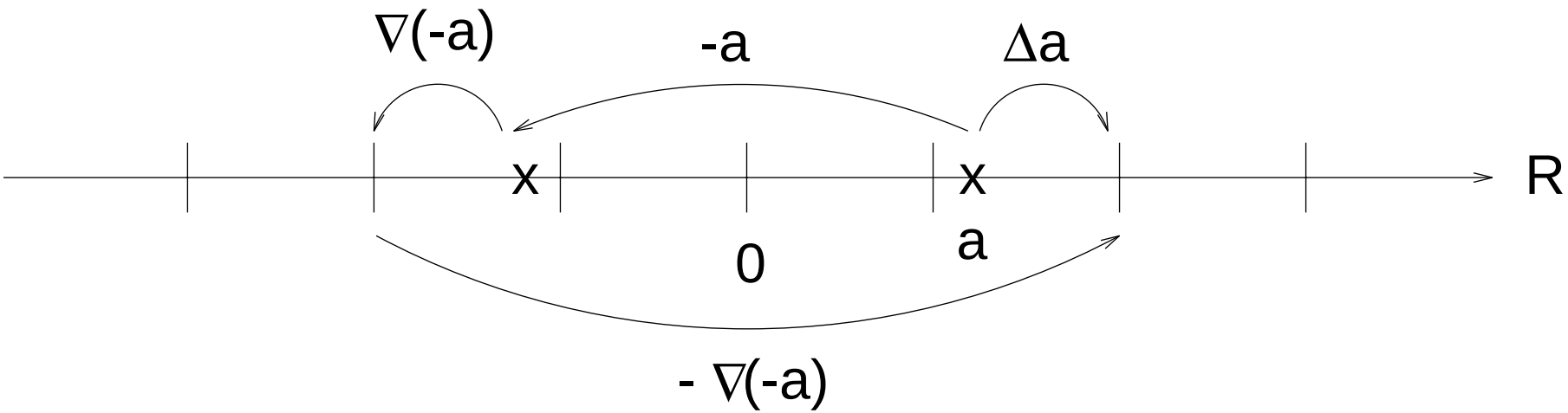
Abhängigkeit der gerichteten Rundungen

Sei $S \subset \mathbb{R}^{\pm\infty}$ ein symmetrischer Zahlenbereich mit $\pm\infty \in S$.

Dann gilt $\forall a \in \mathbb{R}^{\pm\infty}: \bigtriangledown a = -\bigtriangleup(-a) \quad \wedge \quad \bigtriangleup a = -\bigtriangledown(-a)$

Die gerichteten Rundungen sind also wechselseitig voneinander abhängig: Lediglich eine von beiden muss direkt implementiert werden, die andere folgt daraus mittels Vorzeichenwechsel.

Gerichtete Rundungen sind nicht antisymmetrisch: $\forall a \in \mathbb{R}^{\pm\infty} \setminus S: \bigtriangledown a < \bigtriangleup a = -\bigtriangledown(-a)$



Monotone Rundungen

Monotonie ist interessant, weil sie der intuitiven Vorstellung der Rundung entgegenkommt:

Für jedes Element a aus dem Wertebereich einer monotonen Rundung $\square$
ist die Urbildmenge $\square^{-1} a$ konvex.

Für alle $a \in \mathbb{R}^{\pm\infty}$ sei $I_a := [\bigtriangledown a, \bigtriangleup a]$.

Die Rundung $\square\colon \mathbb{R}^{\pm\infty} \to S$ ist genau dann monoton, wenn es für alle $a \in \mathbb{R}^{\pm\infty}$
eine Partition (I_a^1, I_a^2) von I_a gibt mit $I_a^1 < I_a^2$ und

$$\square a = \begin{cases} \bigtriangledown a & \text{für alle } a \in I_a^1 \\ \bigtriangleup a & \text{für alle } a \in I_a^2 \end{cases}$$

Bias einer Rundung

Zur Beurteilung einer Rundung wird oft betrachtet, wie sich die Rundung auf den Mittelwert der gerundeten Zahlen auswirkt:

Werden die Zahlen durch Rundung im Mittel kleiner, liegt *Downward-biased-Rounding* vor.

Werden sie im Mittel größer, heißt die Rundung *upward-biased*.

Eine Rundung, die den Mittelwert der Zahlen unverändert lässt, heißt *unbiased*.

Der *Bias* hängt von der Wahrscheinlichkeitsverteilung (der Signifikanten) ab, mit der die Häufigkeit des Auftretens der einzelnen Operanden der Rundung beschrieben wird.

$\bigtriangledown$ ist stets downward-biased (auch bereichsweise), es sei denn, alle zu rundenden Zahlen wären bereits Maschinenzahlen im korrekten Format; entsprechend ist $\bigtriangleup$ stets upward-biased.

Antisymmetrische Rundungen sind bei zur Null symmetrischer Verteilung unbiased.

Neben dem Bias für alle Zahlen kann auch getrennt jeweils ein Bias für die positiven und die negativen Zahlen berechnet werden.

Die häufig unterstellte Gleichverteilung der Signifikanten von Ergebnissen liegt in der Praxis oft nicht einmal annähernd vor, etwa bei der beliebten Division durch 2.

Genauigkeit berechneter Ergebnisse

- Genauigkeit eines Einzelergebnisses:
  Abweichung zwischen einer exakten reellen Zahl a und ihrer Approximation durch eine berechnete Maschinenzahl $\rho(a)$.

  Absoluter Fehler: $|a - \rho(a)|$

  Relativer Fehler: $\left|\frac{a - \rho(a)}{a}\right|$ für $a \neq 0$ (0 für $a = 0$, falls ρ eine Rundung ist)

  Anzahl gültiger (d. h. übereinstimmender) Stellen (in einer bestimmten Darstellung).

- Durchschnittlicher Fehler, abhängig von der Häufigkeit des Auftretens der einzelnen Zahlen a (Dichtefunktion).

- Maximaler absoluter oder relativer Fehler (über alle möglichen Eingaben).

Monotone, antisymmetrische Rundung nach Float$^{\pm\infty}(R, I, e_1, e_2)$

Mit $s_\mu(x) = \mu/R \times \triangle x + (R-\mu)/R \times \triangledown x$ sowie $s_0(x) = \triangledown x$ und $s_R(x) = \triangle x$ ergeben sich für $\mu = 0, 1, 2, \ldots, R$ monotone, antisymmetrische Rundungen $\square_\mu$ von $\mathbb{R}^{\pm\infty}$ nach Float$^{\pm\infty}(R, I, e_1, e_2)$ durch

$$\forall a \in [0, R^{e1-1}) : \square_\mu a = 0$$

$$\forall a \geq R^{e1-1} : \square_\mu a = \begin{cases} \triangledown a & \text{für} \quad a \in [\triangledown a, s_\mu(a)) \\ \triangle a & \text{für} \quad a \in [s_\mu(a), \triangle a] \end{cases}$$

$$\forall a < 0 : \square_\mu a = -\square_\mu(-a)$$

$\square_R$ heißt *Rundung zur Null*. Es gilt:

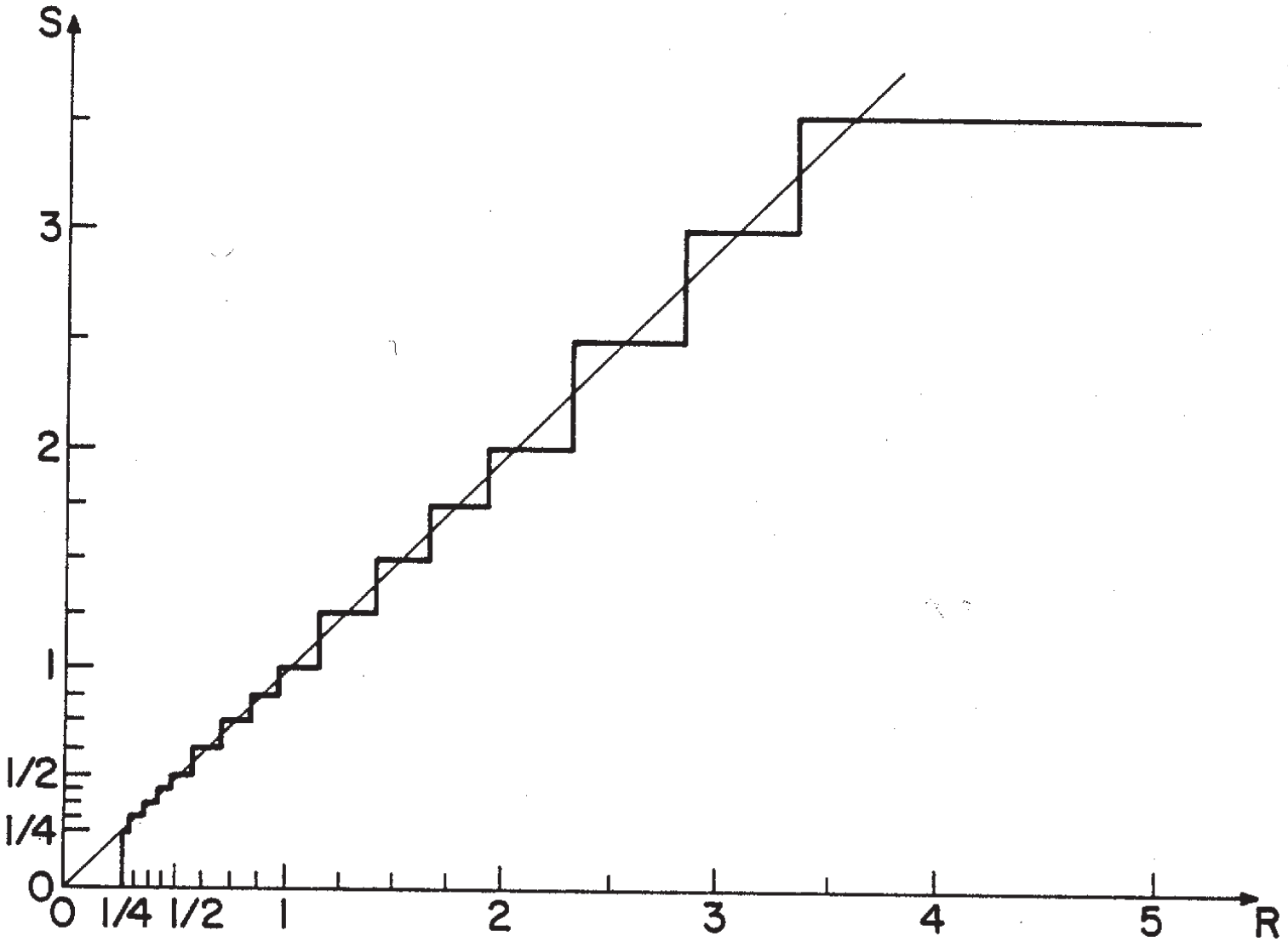
$$\forall a \geq 0 : \square_R a = \triangledown a$$
$$\forall a \leq 0 : \square_R a = \triangle a$$
$$\forall a : \square_R a = \mathrm{sgn}(a) \times \triangledown |a|$$

$\square_0$ ist bis auf den Unterlaufbereich identisch mit der *Rundung nach Unendlich*, $\tilde{\square}_0$, für die gilt:

$$\forall a \geq 0 : \tilde{\square}_0 a = \triangle a$$
$$\forall a \leq 0 : \tilde{\square}_0 a = \triangledown a$$
$$\forall a : \tilde{\square}_0 a = \mathrm{sgn}(a) \times \triangle |a|$$

Für gerades R ist $\square_{R/2}$ eine *Rundung zur nächsten Maschinenzahl*.

Die Rundung $\square_1$ nach Float$^{\pm\infty}(2, 3, -1, 2)$



Alternativen zu den beschriebenen Rundungen

Der Standard ANSI/IEEE-754 schreibt für Implementierungen der Rechnerarithmetik mit Basis $R = 2$ die Rundungen $\bigtriangledown$, $\bigtriangleup$ und $\square_2$ sowie eine geringfügig von $\square_1$ abweichende Rundung $\tilde{\square}_1$ vor. Intervallmitten im darstellbaren Bereich werden bei $\tilde{\square}_1$ abwechselnd nach unten bzw. oben gerundet, wobei die niederwertigste Ziffer stets den Wert Null bekommt (*Round-to-nearest-even*), bei $\square_1$ immer nach außen (Richtung Unendlich).

Außerdem gilt mit $B = (1 - 2^{-l}) \times 2^{e2}$, $B' = (1 - 2^{-l-1}) \times 2^{e2}$

$\square_1 a = B$ für $B < a < \infty$, aber $\tilde{\square}_1 a = B$ für $B < a < B'$, $\tilde{\square}_1 a = \infty$ für $B' \leq a$

$\square_1 a = -B$ für $-\infty < a < -B$, aber $\tilde{\square}_1 a = -B$ für $-B' < a < -B$, $\tilde{\square}_1 a = -\infty$ für $a \leq -B'$

Eine plausible Modifikation von $\square_\mu$ wäre auch

$\forall a \in [0, R^{e1-1}) : \bar{\square}_\mu a = 0$

$$\forall a \geq R^{e1-1} : \bar{\square}_\mu a = \begin{cases} \bigtriangledown a & \text{für} \quad a \in [\bigtriangledown a, s_\mu(a)] \\ \bigtriangleup a & \text{für} \quad a \in (s_\mu(a), \bigtriangleup a] \end{cases}$$

$\forall a < 0 : \bar{\square}_\mu a = -\bar{\square}_\mu(-a)$

Daneben kommen auch nicht antisymmetrische Rundungen zur nächsten Maschinenzahl vor, etwa diejenige, bei der die Intervallmitten stets nach oben gerundet werden (also $\square_{R/2}$ für positive, $\bar{\square}_{R/2}$ für negative Zahlen); die Wahl einer solchen (oder auch einer anderen) Rundung ist in der Regel in ihrer besseren Implementierbarkeit begründet.

Rundung durch Abschneiden

In der Praxis sehr beliebt ist *Rundung durch Abschneiden* (*Chopping* oder *Truncation*).
Dabei werden die nach der l-ten Stelle folgenden Ziffern des Signifikanten einfach abgeschnitten.

Für Signifikanten in Vorzeichen-Betrag-Darstellung oder im Verminderten-Basis-Komplement bewirkt dies eine Rundung zur Null $\square_R$.
Die Rundung ist im Positiven downward-biased, im Negativen upward-biased.

Liegt der Signifikant dagegen im R-Komplement vor, implementiert die Rundung durch Abschneiden die Rundung nach unten $\bigtriangledown$. Dies liegt an der Eigenschaft

$$a = (-a_1 \times R^{l-1} + \sum_{i=0}^{l-2} a_{l-i} \times R^i) \times R^{e-l}$$

Die sogenannte *Von-Neumann-Rundung* (*Jamming*, *Force-1-Rounding*) ist ein Verfahren für $R = 2$, das praktisch so schnell wie Rundung durch Abschneiden durchgeführt werden kann.

Dabei wird der Signifikant nach der l-ten Stelle abgeschnitten, das niederwertigste Bit erhält danach den Wert 1; die Eigenschaft (R1) einer Rundung geht allerdings verloren.

Eine Modifikation, die aus dem Verfahren wieder eine Rundung macht, besteht darin, das l-te Bit nur dann auf 1 zu setzen, wenn sich der Wert der Zahl beim Abschneiden geändert hat.

Rundung zur nächsten Maschinenzahl

Ebenfalls relativ leicht zu implementieren ist *Half-adjust-Rounding* für gerade Basen R.
Dabei wird der Signifikant des exakten Ergebnisses an der $(l+1)$-ten Signifikantenstelle um $R/2$ Einheiten erhöht und dann nach der l-ten Stelle abgeschnitten.

Für Signifikanten im R-Komplement resultiert eine nicht antisymmetrische Rundung zur nächsten Maschinenzahl mit Aufrundung der Intervallmitten (*Round-to-nearest-up*).
Für gleichverteilte Signifikanten ist die Rundung upward-biased, auch bereichsweise.

Für Signifikanten in Vorzeichen-Betrag-Darstellung ergibt sich dagegen die Rundung $\square_{R/2}$.
In der Praxis missbräuchlich ebenfalls als *Round-to-nearest-up* bezeichnet,
sollte diese Variante besser *Round-to-nearest-out* heißen.
Bei Gleichverteilung ist die Rundung im Positiven upward-biased, im Negativen downward-biased.

Obwohl vom Ergebnis her mit der aus der Schule bekannten Rundungsmethode (Ansehen der nächsten Ziffer hinter der beabsichtigten letzten Stelle) identisch, ist Half-adjust-Rounding implementierungstechnisch günstiger, da die möglicherweise auftretende Carry-Propagierung bereits während der Berechnung des Zwischenergebnisses erfolgt.

Weitere Rundungen zur nächsten Maschinenzahl

Komplizierter zu implementieren sind die Verfahren des *Round-to-nearest-even* bzw. *Round-to-nearest-odd*; beide sind Varianten der Rundung zur nächsten Maschinenzahl.

Bei *Round-to-nearest-even* wird die Intervallmitte zu der benachbarten Maschinenzahl gerundet, deren letzte Ziffer gerade ist.

Bei *Round-to-nearest-odd* wird die Intervallmitte dagegen zu der benachbarten Maschinenzahl gerundet, deren letzte Ziffer ungerade ist.

Beide Verfahren wirken tendenziell dem bereichsweisen Biasing entgegen.

Welche Variante verwendet werden sollte, hängt von der benutzten Zahlenbasis R ab.
Für $R \equiv 2 \pmod 4$ wird Round-to-nearest-even empfohlen, bei $R \equiv 0 \pmod 4$ Round-to-nearest-odd.

Explizite Darstellung der Rundung $\triangledown$ nach Float$^{\pm\infty}(R, l, e_1, e_2)$

Für $a = \pm a_1 a_2 \ldots a_l, a_{l+1} a_{l+2} \ldots \times R^{e-l}$ mit Signifikant in Vorzeichen/Betrag-Darstellung gilt

$$\triangledown a = \begin{cases} +\infty & : a = +\infty \\ B & : B \leq a < +\infty \quad \text{mit } B = (1 - R^{-l}) \times R^{e2} \\ a_1 a_2 \ldots a_l \times R^{e-l} & : R^{e1-1} \leq a < B \\ 00 \ldots 0 \times R^{e1-l} & : 0 \leq a < R^{e1-1} \\ -100 \ldots 0 \times R^{e1-l} & : -R^{e1-1} \leq a < 0 \\ -a_1 a_2 \ldots a_l \times R^{e-l} & : -B \leq a < -R^{e1-1}, \bigwedge\limits_{i \geq 1} a_{l+i} = 0 \\ -100 \ldots 0 \times R^{e-l+1} & : -B < a < -R^{e1-1}, \bigwedge\limits_{1 \leq i \leq l} a_i = R - 1, \bigvee\limits_{i \geq 1} a_{l+i} \neq 0 \\ -(a_1 a_2 \ldots a_l + 1) \times R^{e-l} & : -B < a < -R^{e1-1}, \bigvee\limits_{1 \leq i \leq l} a_i \neq R - 1, \bigvee\limits_{i \geq 1} a_{l+i} \neq 0 \\ -\infty & : -\infty \leq a < -B \end{cases}$$

Mit Hilfe der Gauß-Funktion $\lfloor \cdot \rfloor$ kann diese Beschreibung (und damit auch die Implementierung) wesentlich verkürzt werden:

$$\triangledown a = \begin{cases} +\infty & : a = +\infty \\ B & : B \leq a < +\infty \\ \lfloor a \times R^{l-e} \rfloor \times R^{e-l} & : R^{e1-1} \leq |a| < B \\ 00 \ldots 0 \times R^{e1-l} & : 0 \leq a < R^{e1-1} \\ -100 \ldots 0 \times R^{e1-l} & : -R^{e1-1} \leq a < 0 \\ -\infty & : -\infty \leq a < -B \end{cases}$$

Approximation reeller Zahlen durch Maschinenzahlen

Jede reelle Zahl ist beliebig genau approximierbar durch eine Folge von Maschinenzahlen

$a^{(k)} = s \times \sum_{i=-k}^{n} a_i \times R^i$, n fest, $k \to \infty$

Wegen der Vertauschbarkeit der Reihenfolge von Limesbildung und Anwendung einer stetigen Operation in $\mathbb{R}$ gilt für $\circ \in \{+, -, \times, /\}$

$$c = a \circ b = (\lim_{k\to\infty} a^{(k)}) \circ (\lim_{k\to\infty} b^{(k)}) = \lim_{k\to\infty} (a^{(k)} \circ b^{(k)})$$

Das Ergebnis jeder arithmetischen Grundoperation auf reellen Zahlen kann deshalb beliebig genau aus immer genauer werdenden Approximationen der Operanden berechnet werden.

Die dabei berechneten Ziffern des approximierenden Ergebnisses stimmen jedoch nicht unbedingt mit den entsprechenden Ziffern des exakten Ergebnisses überein.

Beispiele für Approximationsanomalien

$$\frac{1}{29} + \frac{2}{29} = \frac{3}{29} \quad \text{bzw.}$$

$$0{,}03448\ldots + 0{,}06896\ldots = 0{,}10344\ldots$$

k	$a^{(k)} + b^{(k)}$	$c^{(k)}$
1	0,0	0,1
2	0,09	0,10
3	0,102	0,103
4	0,1033	0,1034

$$\frac{1}{3} \times \frac{1}{3} = \frac{1}{9} \quad \text{bzw.}$$

$$0{,}\bar{3} \times 0{,}\bar{3} = 0{,}\bar{1}$$

k	$a^{(k)} \times b^{(k)}$	$c^{(k)}$
1	0,09	0,1
2	0,1089	0,11
3	0,110889	0,111
4	0,11108889	0,1111

Das Phänomen, dass Rundung zur nächsten Maschinenzahl hier die gewünschten Ziffern liefert, ist der Grund für die Berechnung von Produkten in doppelter Genauigkeit.

Approximationsanomalie bei Multiplikation in doppelter Genauigkeit

Dass die Auswertung der Multiplikation in doppelter Genauigkeit mit anschließender Rundung zur nächsten Maschinenzahl (oder irgendeiner anderen Rundung) nicht unbedingt lauter gültige Stellen produziert, zeigt folgendes Beispiel:

$$1{,}9999 \times 1{,}9999 = 3{,}99960001$$

k	$a^{(k)} \times b^{(k)}$	$c^{(k)}$
1	1	3
2	3,61	3,9
3	3,9601	3,99
4	3,996001	3,999

Verwendung unterschiedlicher Genauigkeiten

- In Programmiersprachen bzw. in Hardware meist mehrere Systeme von Maschinenzahlen unterschiedlicher Genauigkeit und evtl. unterschiedlichen Zahlenbereichs implementiert.
- Ausnahmen: z. B. PASCAL (nur ein Typ `real` vorhanden).
- Klassisch: *Single-Precision* SP, *Double-Precision* DP; `real(4)`, `real(8)` in FORTRAN.
- Double-Precision liefert genauere Ergebnisse bei höherem Zeitaufwand
  (Addition bis etwa doppelt so hoch, Multiplikation bis etwa vierfach).
- Grober Zusammenhang: log (relativer Fehler DP) $\approx 2 \times$ log (relativer Fehler SP)
- Der Zusammenhang SP $\subset$ DP wird nicht generell garantiert!
- Viele (insbesondere neuere) DP-Zahlensysteme haben auch einen größeren dynamischen Zahlenbereich als die entsprechenden SP-Zahlensysteme.
- Heute häufig weitere Zahlensysteme höherer Genauigkeit, z. B. *Quadruple-Precision*
  Zahlensysteme in C: `float` $\subset$ `double` $\subset$ `long double`
  Standard ANSI/IEEE-754: *Single* $\subset$ *Double* $\subset$ *Extended*
  Revision IEEE-754R: *binary16* $\subset$ *binary32* $\subset$ *binary64* $\subset$ *binary128*
  sowie *decimal32* $\subset$ *decimal64* $\subset$ *decimal128*
- In der Signalverarbeitung stellenweise höhere Genauigkeit (typischerweise dreifach).

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Die Standards ANSI/IEEE 754 und 854

Bestehende und künftige Standards der Rechnerarithmetik

ANSI/IEEE Standard 754-1985 (siehe z.B. http://754r.ucbtest.org/standards/754.pdf)

- Titel: IEEE Standard for Binary Floating-Point Arithmetic
- Verabschiedet (approved) am 21. März 1985 vom IEEE Standards Board
  (IEEE = Institute of Electrical and Electronics Engineers)
- Verabschiedet am 26. Juli 1985 vom American National Standards Institute (ANSI)
- Bestätigt (reaffirmed) am 6. Dezember 1990 vom IEEE Standards Board
- Internationaler Standard IEC 60559:1989
  Binary floating-point arithmetic for microprocessor systems
- Deutsch als DIN IEC 60559:1992-01 bzw. HD 592 S1:1991
  Binäre Gleitpunkt-Arithmetik für Mikroprozessor-Systeme

ANSI/IEEE Standard 854-1987 (siehe z.B. http://754r.ucbtest.org/standards/854.pdf)

- Titel: IEEE Standard for Radix-Independent Floating-Point Arithmetic
- Verabschiedet am 12. März 1987 vom IEEE Standards Board
- Verabschiedet am 10. September 1987 vom American National Standards Institute
- Bestätigt am 17. März 1994 vom IEEE Standards Board
- Bestätigt am 23. August 1994 vom American National Standards Institute

IEEE 754R (siehe z.B. http://754r.ucbtest.org)

- Revision, die u.a. ANSI/IEEE-754 und ANSI/IEEE-854 zusammenführen soll
- Bemühungen seit dem Jahr 2000
- Seit Ende 2006 etliche Abstimmungen über sukzessive Entwürfe
- Draft 1.9.0 bisher letzter Stand, Abstimmung lief bis 5. Mai 2008, wird wohl Standard

Zweck und Einsatzbedingungen der Standards

Allgemeines:

- Die Anwendung von IEEE Standards ist freiwillig.
- Die Standards bestehen aus strengen Vorschriften sowie dringenden Empfehlungen.
- IEEE-754/854 ist systemzentriert: Als konform zum Standard wird immer ein Programmier- oder Anwendungssystem verstanden, nicht eine einzelne Hardware-Komponente.
- IEEE-854 soll Verallgemeinerung von IEEE-754 sein: Bis auf marginale Abweichungen ist jede zu IEEE-754 konforme Implementierung auch zu IEEE-854 konform.

Gemeinsame Ziele:

- Erleichterte Portierung existierender Programme von einer Vielzahl von Rechnern auf zum Standard konforme Systeme.
- Einfache, sichere Anfertigung numerisch anspruchsvoller Programme durch Nicht-Numeriker.
- Ermutigung zu Entwicklung und Verteilung robuster und effizienter numerischer Programme, die ohne größere Änderungen zwischen konformen Systemen ausgetauscht werden können. (Nur IEEE-754: Auf eine spezifizierte Teilmenge des Standards eingeschränkt, sollten diese Programme auf allen konformen Systemen identische Resultate produzieren.)
- Direkte Unterstützung für Laufzeitfehleranalyse, abgestufte Ausnahmebehandlung und Intervallarithmetik zu vernünftigen Kosten.
- Vorbereitung der Entwicklung von Standardfunktionen (exp, cos, . . .), hochgenauer Arithmetik und der Verbindung von numerischer mit symbolischer Berechnung.
- Verfeinerungen und Erweiterungen sollen ermöglicht, nicht verhindert werden.

Der Standard ANSI/IEEE 754-1985

Titel: IEEE Standard for Binary Floating-Point Arithmetic

Der Standard spezifiziert

- Klassen erweiterter Gleitkommasysteme (sog. Grunddatenformate bzw. erweiterte Formate)
- Repräsentationen für die Grunddatenformate
- Additions-, Subtraktions-, Multiplikations-, Divisions-, Modulo-, Quadratwurzel-, Vergleichsoperationen für Gleitkommasysteme
- Operationen für die Umwandlung zwischen Ganzzahl- und Gleitkommasystemen
- Operationen für die Umwandlung zwischen verschiedenen Gleitkommasystemen
- Konversionsoperationen zwischen Grunddatenformaten und Dezimalzeichenreihen (die Dezimalbrüche codieren)
- Gleitkomma-Ausnahmen und ihre Behandlung, einschließlich NANs (Formate, die keine Zahlen im eigentlichen Sinne sind)

Der Standard spezifiziert nicht

- Zahlenbereiche und Repräsentation von Dezimalzeichenreihen und Ganzzahlen
- Interpretation des Signifikanten von NANs
- Operationen für die Umwandlung zwischen erweiterten Formaten und Dezimalzeichenreihen

754: Spezifizierte Gleitkommasysteme und Repräsentationsparameter

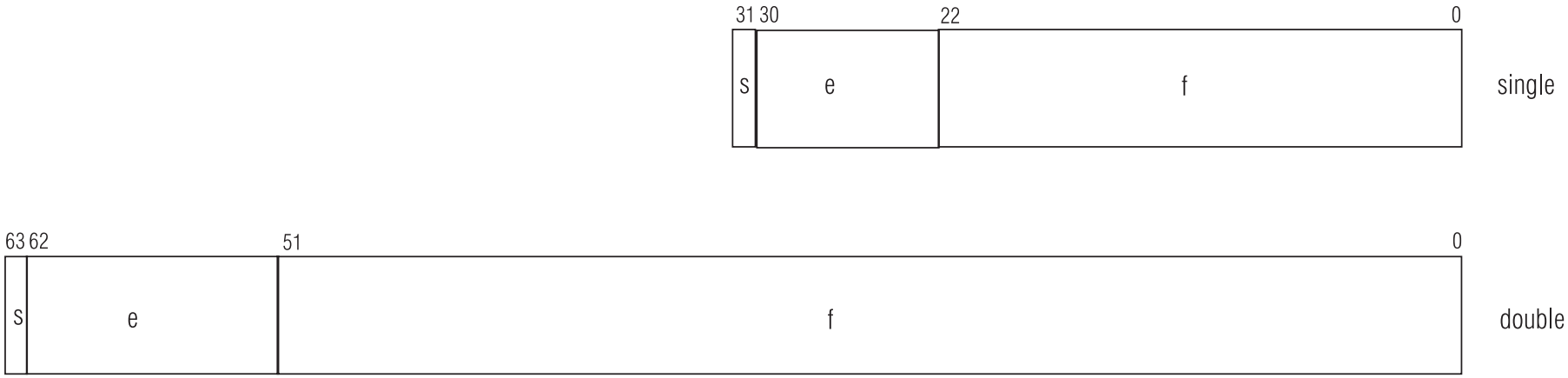
Der Standard legt die wichtigsten Parameter von zwei Grunddatenformaten (*single*, *double*) und zwei erweiterten Formaten (*single extended*, *double extended*) fest;
alle implementieren Erweiterungen von Gleitkommasystemen $\mathrm{Float}_e(2, l, e1, e2)$:

	single	*single extended*	*double*	*double extended*
l	24	≥ 32	53	≥ 64
$e1$	-125	≤ -1021	-1021	≤ -16381
$e2$	128	≥ 1024	1024	≥ 16384
Bias	126	unspezifiziert	1022	unspezifiziert
Exponentenfeld	8	≥ 11	11	≥ 15
Formatbreite	32	≥ 43	64	≥ 79

Das Format *single* ist obligatorisch; Implementierungen mit Format *double* sollten auch das Format *double extended* bereitstellen, die übrigen zumindest das Format *single extended*.

All diese Gleitkommasysteme besitzen eine positive und eine negative Null, die als Vertreter für positive bzw. negative Zahlen mit sehr kleinem Betrag aufgefasst werden können; außerdem die symbolischen Größen $+\infty$ und $-\infty$ sowie mindestens ein Signaling-NAN und ein Quiet-NAN.

754: Vorgeschriebene Repräsentation für Grunddatenformate



Das Vorzeichen besitzt die übliche Codierung (0 für positiv, 1 für negativ);
positive und negative Null werden durch $00\ldots0$ bzw. $10\ldots0$ codiert.

Die Magnitude f liegt in Hidden-Bit-Darstellung vor.

Die Repräsentation ist nicht redundant;
denormalisierte Darstellungen kennzeichnet ein Biased Exponent $0\ldots0$,
der für den kleinsten vorkommenden Exponenten steht (-125 bei *single*, -1021 bei *double*).

Wenn der Biased Exponent den Wert $e_2 + \mathit{Bias} + 1$ besitzt, liegt entweder ein NAN ($f \neq 0$)
oder — abhängig vom Vorzeichen s — eine der Zahlen $-\infty$ oder ∞ vor.

754: Erweiterte Formate

Die Codierung der Zahlen in den erweiterten Formaten bleibt im wesentlichen unspezifiziert.

Es muss jedoch das gesamte zugrundeliegende erweiterte Gleitkommasystem $\mathrm{Float}_e(2, l, e1, e2)$ mit mindestens einer positiven Null und mindestens einer negativen Null implementiert werden; hinzu kommen Darstellungen für $-\infty$ und ∞, sowie mindestens ein Signaling-NAN und mindestens ein Quiet-NAN.

Von der Hidden-Bit-Technik darf Gebrauch gemacht werden.

Die erweiterten Formate dürfen auch Felder enthalten, die im Standard nicht aufgeführt sind, oder spezielle Repräsentationen für im Standard nicht genannte Zwecke vorsehen.

Die Implementierung hat bei Vorliegen von Redundanz so zu erfolgen, dass zwischen alternativen Darstellungen echter, von Null verschiedener Zahlen in der Wirkung kein Unterschied besteht; dasselbe gilt in Bezug auf die Darstellungen von $-\infty$, ∞, $+0$ und -0.

754: Rundungen

Weicht das abgespeicherte Ergebnis einer Operation vom exakten Ergebnis ab, wird die Inexact-Ausnahme angezeigt (außer evtl. bei Umwandlungen zwischen Gleitkommazahlen und Dezimalzeichenreihen).

Vergleichs- und Modulo-Operationen sind unabhängig von der verwendeten Rundung;
alle übrigen Operationen werden entsprechend eines der folgenden Rundungsmodi ausgeführt:

- Gerichtete Rundung: $\bigtriangledown$, $\bigtriangleup$ oder $\square_2$ (Rundung zur Null, implementiert durch *Chopping*).
- Rundung zur nächsten Maschinenzahl: $\tilde{\square}_1$ für erweitertes Gleitkommasystem.

Alle vier Rundungsmodi müssen implementiert werden und sich vom Benutzer einstellen lassen;
Voreinstellung ist die Rundung zur nächsten Maschinenzahl.

Die verwendete Rundungsgenauigkeit kann vom Speicherformat des Ergebnisses abweichend gewählt werden, um eine geringere Anzahl von Stellen zu simulieren.

So kann etwa in Systemen, die nur Operanden im Format *double* verarbeiten,
die Rundungsgenauigkeit entsprechend dem Format *single* festgelegt werden.

Die Implementierung hat sicherzustellen, dass durch die abweichende Rundungsgenauigkeit kein zusätzlicher Fehler in Bezug auf die tatsächliche Rundungsgenauigkeit entsteht.

754: Operationen

Der Standard fordert die Implementierung mindestens der folgenden Typen von Operationen:

- $+$, $-$, $\times$ und $/$ für beliebige Operanden gleichen Formats
  (empfohlen auch für Operanden mit gemischtem Format)
- Eine Modulo-Operation $\%$ für Operanden gleichen Formats (gemischt empfohlen):
  $A \% B = A - N \times B$ mit der Ganzzahl $N = \mathit{Round\text{-}to\text{-}nearest\text{-}even}(A/B)$
- Quadratwurzel für einen Operanden beliebigen Formats
- Vergleichsoperationen zwischen Gleitkommazahlen
- Rundung von Gleitkommazahlen zu Ganzzahlen (dargestellt im gleichen Gleitkommaformat) entsprechend dem eingestellten Rundungsmodus
- Umwandlung von Gleitkommazahlen in Ganzzahlen (dargestellt in einem der unterstützten Ganzzahlenformate), und umgekehrt (Anmerkung: Die Ganzzahlen sind meist um Größenordnungen kleiner als die größte darstellbare Gleitkommazahl)
- Umwandlungen zwischen verschiedenen Gleitkommaformaten: Zur höheren Genauigkeit hin exakt, ansonsten mit Rundung entsprechend dem eingestellten Rundungsmodus
- Umwandlungen zwischen Gleitkommazahlen und Dezimalzeichenreihen

Interessanterweise wird kein unäres Minus gefordert! Eine Empfehlung, die allerdings nicht expliziter Teil des Standards ist, bevorzugt die Implementierung durch reinen Vorzeichenwechsel gegenüber der Hilfskonstruktion $0 \ominus x$. Unterschiede bestehen, wenn x Null oder ein NAN ist.

754: Umwandlung von Gleitkommazahlen und Dezimalzeichenreihen

Für jedes Grunddatenformat müssen Umwandlungsoperationen von und nach Dezimalzeichenreihen der Form $\pm M \times 10^{\pm N}$ mit folgender Charakteristik implementiert werden:

	dezimal zu binär		*binär zu dezimal*	
	max M	max N	max M	max N
single	$10^9 - 1$	99	$10^9 - 1$	53
double	$10^{17} - 1$	999	$10^{17} - 1$	340

N ist dabei, solange ohne Genauigkeitsverlust möglich, minimal zu wählen.

Wird der Bereich von M überschritten, können Ziffern nach der 9. bzw. 17. Ziffer beliebig gewählt werden (typischerweise Null).

754: Rundung bei Umwandlung von/nach Dezimalzeichenreihen

Das Ergebnis der Umwandlung muss korrekt gerundet werden, falls die Dezimalzeichenreihe in folgendem reduzierten Bereich liegt:

	dezimal zu binär		*binär zu dezimal*	
	max M	max N	max M	max N
single	$10^9 - 1$	13	$10^9 - 1$	13
double	$10^{17} - 1$	27	$10^{17} - 1$	27

Falls die Dezimalzeichenreihe zwar im zulässigen, nicht aber im reduzierten Bereich liegt, müssen folgende Vorschriften eingehalten werden:

- Rundung zur nächsten Maschinenzahl:
  Falls weder Überlauf noch Unterlauf eintritt, darf der Fehler gegenüber exakter Rundung nicht größer als 0,47*ulp* (*units in the last place*) des Ergebnisses sein;
  die sukzessive Umwandlung einer Gleitkommazahl in eine Dezimalzeichenreihe und deren Umwandlung in eine Gleitkommazahl muss die Identität ergeben.
- Gerichtete Rundungen:
  Der Fehler muss korrektes Vorzeichen besitzen und darf nicht größer als 1,47*ulp* des Ergebnisses sein.

In allen Fällen muss die Umwandlungsfunktion monoton sein.

754: Vergleichsoperationen

Vergleiche müssen zwischen Operanden beliebiger (auch gemischter) Formate möglich sein; sie sind immer exakt.

Bei Vergleichen wird kein Unterschied zwischen negativer und positiver Null gemacht.

Es stehen 4 primäre Prädikate zur Verfügung:

- $<, =, >$:
  Für Zahlen (auch $-\infty$ und ∞) die übliche Bedeutung.
- ? (*unordered*):
  Besitzt genau dann den Wert ‘wahr’, wenn mindestens einer der Operanden ein NAN ist;
  ? liefert selbst dann den Wert ‘wahr’, wenn die beiden Operanden dasselbe NAN darstellen.

Das Ergebnis eines Vergleichs kann je nach Implementierung sein:

- ein *Condition Code*, der gleichzeitig den Wert aller 4 primären Prädikate wiedergibt
- ein Wahrheitswert, der für ein aus primären Prädikaten zusammengesetztes Prädikat steht

754: NANs

Ein *NAN* ist die Codierung einer symbolischen Einheit, die keine Maschinenzahl aus $\mathbb{R}^{\pm\infty}$ bedeutet; es wird unterschieden:

- *Signaling-NANs* zeigen an, dass keine zahlartige Größe vorliegt.
  Wird ein Signaling-NAN als Operand benutzt, erfolgt eine *Invalid-Operation-Ausnahme*.
- *Quiet-NANs* werden in arithmetischen Operationen wie Zahlen behandelt,
  produzieren also wieder zahlartige Resultate (meist erneut NANs).
  Quiet-NANs geben nach Ablauf einer Folge von Rechenschritten Aufschluss über dabei eventuell aufgetretene anomale Situationen.

Wird eine Invalid-Operation nicht durch einen *Trap* aufgefangen, und soll ein Gleitkomma-Ergebnis berechnet werden, so wird ein Quiet-NAN erzeugt.

Operationen, in denen nur Quiet-NANs vorkommen, jedoch keine Signaling-NANs, und die ein Gleitkomma-Ergebnis erzeugen sollen, liefern ein Quiet-NAN.

754: Vorzeichen von Ergebnissen

Vorzeichen von Resultaten — auch die von Null, Unendlich und NANs — ergeben sich (meist) systematisch aus den Vorzeichen der Operanden:

- Addition:
  Vorzeichen der Operanden bei übereinstimmendem Vorzeichen;
  für Resultat Null ergibt sich andernfalls -0 bei Rundung nach unten, $+0$ sonst.
- Subtraktion:
  Vorzeichen des linken Operanden bei verschiedenem Vorzeichen;
  für Resultat Null ergibt sich andernfalls -0 bei Rundung nach unten, $+0$ sonst.
- Multiplikation und Division:
  Plus bei gleichem Vorzeichen der Operanden, ansonsten Minus.
- Modulo-Operation:
  Vorzeichen des linken Operanden bei Resultat Null.
- Quadratwurzel:
  $\sqrt{-0} = -0$;
  positives Vorzeichen für alle anderen zulässigen Operanden.
- Rundung einer Gleitkommazahl zu einer Ganzzahl:
  Vorzeichen des Operanden.

Vorzeichen von NANs werden durch den Standard nicht interpretiert!

754: Ausnahmen

Der Standard gibt fünf Ausnahmen an, deren Auftreten angezeigt werden muss.
Dieses Anzeigen erfolgt über Setzen eines *Flags* und/oder Durchführung eines *Traps*.

Ausnahmen treten nur einzeln auf, abgesehen von der Inexact-Ausnahme,
die zusammen mit Überlauf oder Unterlauf vorkommen kann.

Der Standard legt nahe, dass zu jeder der vorgeschriebenen Ausnahmen die Zulassung
bzw. Unterdrückung eines Traps durch Benutzer ermöglicht werden sollte.

Benutzer sollten auch eigene *Trap-Handler* zur Verfügung stellen können.

Das Ergebnis nach einem Trap kann sich von dem ohne Benutzung des Traps unterscheiden.

Jeder Ausnahme entspricht ein eigenes Flag, das bei Auftreten der Ausnahme gesetzt wird,
falls kein Trap erfolgt; dieses wird nur auf Anforderung zurückgesetzt.

Flags können einzeln getestet bzw. zurückgesetzt werden,
und alle Flags können gemeinsam gesichert oder geladen werden.

754: Invalid-Operation-Ausnahme

Die Invalid-Operation-Ausnahme wird angezeigt für folgende unzulässigen Operationen:

- Jede Operation mit einem Signaling-NAN als Operanden.
- Addition und Subtraktion mit Ergebnis $\infty - \infty$.
- Multiplikation $0 \times \pm\infty$ oder $\pm\infty \times 0$.
- Division $0/0$ oder $\pm\infty / \pm\infty$.
- Modulo-Operation $x \otimes y$ mit $x = \pm\infty$ oder $y = 0$.
- Quadratwurzel mit Operanden echt kleiner als Null.
- Umwandlung einer Gleitkommazahl in eine Ganzzahl oder eine Dezimalzeichenreihe, wenn Überlauf, Unendlich oder NAN eine sinnvolle Repräsentation des Ergebnisses verhindern und dies nicht anders angezeigt werden kann.
- Vergleich durch Prädikate, die auf $<$ oder $>$, nicht aber auf ? aufbauen, wenn die Operanden ungeordnet sind.

Sattt einer Gleitkommazahl ergibt sich ein Quiet-NAN, falls ein Trap nicht zugelassen ist.

754: Überlauf-Ausnahme

Die Überlauf-Ausnahme tritt auf, wenn die Operanden einer Operation endliche Zahlen sind und das Ergebnis einen nicht im darstellbaren Bereich des Zielformats liegenden Wert hat.

Ist kein Trap zugelassen, bestimmt sich das Ergebnis folgendermaßen:

- Rundung zur nächsten Maschinenzahl liefert Unendlich mit dem Vorzeichen des korrekten Ergebnisses.
- Rundung zur Null liefert die betragsgrößte darstellbare Zahl mit dem Vorzeichen des korrekten Ergebnisses.
- Rundung nach unten liefert bei positivem exakten Ergebnis die größte darstellbare Zahl, bei negativem exakten Ergebnis $-\infty$.
- Rundung nach oben liefert bei positivem exakten Ergebnis $+\infty$, bei negativem exakten Ergebnis die kleinste darstellbare Zahl.

754: Unterlauf-Ausnahme

Die Unterlauf-Ausnahme kann zwei verschiedene Ursachen haben; für beide sind jeweils zwei alternative Implementierungsmethoden vorgesehen (die gewählte Methode muss aber für alle Operationen gleich sein):

(A) *Tininess*, wird nach folgenden alternativen Verfahren bestimmt:

- Nach der Rundung:
  Bei Rundung des exakten Ergebnisses in Float(2, l) entsteht ein von Null verschiedenes Ergebnis, das echt zwischen der größten negativen und der kleinsten positiven Zahl in Float(2, l, $e1$, $e2$) liegt.
- Vor der Rundung:
  Das exakte Ergebnis ist von Null verschieden und liegt echt zwischen der größten negativen und der kleinsten positiven Zahl in Float(2, l, $e1$, $e2$).

(B) *Loss-of-Accuracy*, wird nach folgenden alternativen Verfahren bestimmt:

- Denormalization-Loss:
  Das gerundete Ergebnis liegt in $\text{Float}_e(2, l, e1, e2) \setminus \text{Float}(2, l, e1, e2)$.
- Inexact-Result:
  Das gerundete Ergebnis weicht vom exakten Ergebnis ab.

Ohne Trap wird Unterlauf nur angezeigt, wenn sowohl (A) als auch (B) eintreten; bei aktiviertem Trap genügt dagegen die Bedingung (A) — unabhängig von der Bedingung (B).

754: Weitere Ausnahmen

Die *Divison-durch-Null-Ausnahme* wird ausgelöst, wenn in einer Division der Divisor Null zusammen mit einem Dividenden auftritt, der eine endliche, von Null verschiedene Zahl darstellt.

Wird kein Trap zugelassen, ergibt sich Unendlich (mit korrektem Vorzeichen) als Ergebnis.

Die *Inexact-Ausnahme* wird angezeigt,
wenn das gerundete Ergebnis einer Operation sich vom exakten Ergebnis unterscheidet,
oder wenn ein Überlauf auftritt, der nicht durch einen Überlauf-Trap abgefangen wird.

754: Trap-Handler

Ist ein Trap zugelassen, so wird bei einer Ausnahme das hierbei erzeugte Ergebnis an den Trap-Handler übergeben; dieser wirkt wie ein Unterprogramm, das statt des Ausnahme-Operanden einen Wert zurückgibt, der dann als Ergebnis der Operation dient.

Der Trap-Handler sollte nach seiner Aktivierung feststellen können,

- welche Ausnahmen in der Operation auftraten;
- die Art der durchgeführten Operation;
- das Format des Resultats der Operation;
- bei Überlauf, Unterlauf oder Inexact-Operation: das korrekt gerundete Ergebnis sowie zusätzliche Information, die im Format der Ergebnisvariablen keinen Platz findet;
- bei Invalid-Operation- bzw. Division-durch-Null-Ausnahme: die Werte der Operanden.

Wenn zugelassen, wird ein Überlauf- oder Unterlauf-Trap vor einem gleichzeitig auftretenden Inexact-Trap behandelt.

Der Standard ANSI/IEEE 854-1987

Titel: IEEE Standard for Radix-Independent Floating-Point Arithmetic

Der Standard spezifiziert

- Einschränkungen der Parameter von Gleitkommasystemen
- Additions-, Subtraktions-, Multiplikations-, Divisions-, Modulo-, Quadratwurzel-, Vergleichsoperationen für Gleitkommasysteme
- Operationen für die Umwandlung zwischen Ganzzahl- und Gleitkommasystemen
- Operationen für die Umwandlung zwischen verschieden genauen Gleitkommasystemen
- Operationen für die Umwandlung zwischen Grunddatenformaten und Dezimalzeichenreihen
- Gleitkomma-Ausnahmen und ihre Behandlung, einschließlich NANs (Formate, die keine Zahlen im eigentlichen Sinne sind)

Der Standard spezifiziert nicht

- Speicherformate von Gleitkommazahlen
- Formate von Dezimalzeichenreihen und Ganzzahlen
- Interpretation des Signifikanten von NANs
- Operationen für die Umwandlung zwischen erweiterten Gleitkommasystemen und Dezimalzeichenreihen

854: Gleitkommasysteme

Der Standard formuliert Einschränkungen für die Parameter von zwei Grunddatenformaten (*single*, *double*) und zwei erweiterten Formaten (*single extended*, *double extended*);
alle implementieren Erweiterungen von Gleitkommasystemen $\text{Float}_e(R, l, e1, e2)$, wobei R für alle zusammen implementierten Formate einheitlich einen der Werte 2 oder 10 besitzt.

Das Format *single* ist obligatorisch; Implementierungen mit Format *double* sollten auch das Format *double extended* bereitstellen, die übrigen zumindest das Format *single extended*.

All diese Gleitkommasysteme besitzen eine positive und eine negative Null, außerdem die symbolischen Größen $+\infty$ und $-\infty$ sowie mindestens ein Signaling-NAN und ein Quiet-NAN.

Hinsichtlich Vorzeichenbehandlung, Ausnahmen, Ausnahmebehandlung und Traps folgt IEEE-854 im wesentlichen IEEE-754.
Bei Rundung einer Gleitkommazahl zu einer Ganzzahl bzw. Konversion wird das Vorzeichen des Operanden beibehalten, jedoch darf -0 in +0 umgewandelt werden, falls nicht anders darstellbar.

Zur Repräsentation legt der Standard lediglich fest, dass redundante Codierungen jeder von Null verschiedenen Zahl sich in ihrer Wirkung nicht unterscheiden dürfen.

854: Einschränkungen für Gleitkommasysteme

Für jedes einzelne der zu implementierenden Zahlensysteme wird festgelegt:

- Es muss $e2 - e1 > 5 \times l$ gelten, empfohlen wird $e2 - e1 > 10 \times l$.
- Es muss $R^{l-1} \geq 10^5$ gelten (dies bedeutet $l \geq 18$ für $R = 2$ bzw. $l \geq 6$ für $R = 10$).
- Für $R = 2$ sollte $e1 + e2 = 3$ gelten; für $R = 10$ sollte $e1 + e2 = 2$ gelten.

Der Zusammenhang zwischen den zu implementierenden Zahlensystemen ist:

	single	*single extended*	*double*	*double extended*
l	l_s	$l_{se} \geq 1{,}2 \times l_s$	$l_d \geq 2 \times l_s + \log_R 10$	$l_{de} \geq 1{,}2 \times l_d$
$e1$	$e1_s$	$e1_{se} \leq 8 \times e1_s - 7$	$e1_d \leq 8 \times e1_s - 7$	$e1_{de} \leq 8 \times e1_d - 7$
$e2$	$e2_s$	$e2_{se} \geq 8 \times e2_s$	$e2_d \geq 8 \times e2_s$	$e2_{de} \geq 8 \times e2_d$

Für $R = 2$ hat zusätzlich zu gelten:
$l_{se} \geq l_s + \lceil \log_2(e2_s - e1_s) \rceil, \quad l_{de} \geq l_d + \lceil \log_2(e2_d - e1_d) \rceil$

Außerdem sollte noch gelten:
$l_{se} > 1 + l_s + \ln(3 \times e2_s \times \ln R)/\ln R, \quad l_{de} > 1 + l_d + \ln(3 \times e2_d \times \ln R)/\ln R$

854: Rundungen

Weicht das abgespeicherte Ergebnis einer Operation vom exakten Ergebnis ab, wird die Inexact-Ausnahme angezeigt (außer evtl. bei Umwandlungen zwischen Gleitkommazahlen und Dezimalzeichenreihen).

Vergleichs- und Modulo-Operationen sind unabhängig von der verwendeten Rundung;
alle übrigen Operationen werden entsprechend eines der folgenden Rundungsmodi ausgeführt:

- Gerichtete Rundung: $\bigtriangledown$, $\bigtriangleup$ oder $\square_R$ (Rundung zur Null).
- Rundung zur nächsten Maschinenzahl: $\tilde{\square}_{R/2}$ (*Round-to-nearest-even*).

Alle vier Rundungsmodi müssen implementiert werden und sich vom Benutzer einstellen lassen;
Voreinstellung ist die Rundung zur nächsten Maschinenzahl.

Die verwendete Rundungsgenauigkeit kann vom Speicherformat des Ergebnisses abweichend gewählt werden, um eine geringere Anzahl von Stellen zu simulieren.

So kann etwa in Systemen, die nur Operanden im Format *double* verarbeiten,
die Rundungsgenauigkeit entsprechend dem Format *single* festgelegt werden.

Die Implementierung hat sicherzustellen, dass durch die abweichende Rundungsgenauigkeit kein zusätzlicher Fehler in Bezug auf die tatsächliche Rundungsgenauigkeit entsteht.

854: Operationen

Der Standard fordert die Implementierung mindestens der folgenden Typen von Operationen:

- $+$, $-$, $\times$ und $/$ für beliebige Operanden gleichen Formats
  (empfohlen auch für Operanden mit gemischtem Format)
- Eine Modulo-Operation $\otimes\!\!\!/$ für Operanden gleichen Formats (gemischt empfohlen):
  $A \otimes\!\!\!/ B = A - N \times B$ mit der Ganzzahl $N = \mathit{Round\text{-}to\text{-}nearest\text{-}even}(A/B)$
- Quadratwurzel für einen Operanden beliebigen Formats
- Vergleichsoperationen zwischen Gleitkommazahlen (identisch zu IEEE-754)
- Rundung von Gleitkommazahlen zu Ganzzahlen (dargestellt im gleichen Gleitkommaformat) entsprechend dem eingestellten Rundungsmodus
- Umwandlung von Gleitkommazahlen in Ganzzahlen (dargestellt in einem der unterstützten Ganzzahlenformate), und umgekehrt (Anmerkung: Die Ganzzahlen sind meist um Größenordnungen kleiner als die größte darstellbare Gleitkommazahl)
- Umwandlungen zwischen verschiedenen Gleitkommaformaten: Zur höheren Genauigkeit hin exakt, ansonsten mit Rundung entsprechend dem eingestellten Rundungsmodus
- Umwandlungen zwischen Gleitkommazahlen und Dezimalzeichenreihen

Interessanterweise wird kein unäres Minus gefordert! Eine Empfehlung, die allerdings nicht expliziter Teil des Standards ist, bevorzugt die Implementierung durch reinen Vorzeichenwechsel gegenüber der Hilfskonstruktion $0 \ominus x$. Unterschiede bestehen, wenn x Null oder ein NAN ist.

854: Umwandlung von Gleitkommazahlen und Dezimalzeichenreihen

Für jedes Grunddatenformat sind Umwandlungsoperationen von/nach Dezimalzeichenreihen der Form $\pm M \times 10^{\pm N}$, $0 \leq M \leq 10^D - 1$ mit folgender Charakteristik zu implementieren:

	max D	max N
$R = 2$	$\lceil 1 + I \times \log_{10} 2 \rceil$	$10^{\lfloor \log_{10} E \rfloor + 1} - 1$ mit $E = \max\{D + (I - e1) \times \log_{10} R, 1 - D + e2 \times \log_{10} R\}$
$R = 10$	I	

N ist dabei, solange ohne Genauigkeitsverlust möglich, minimal zu wählen.

Wird der Bereich von M überschritten, können Ziffern hinter der Stelle max D beliebig gewählt werden (typischerweise Null).

854: Rundung bei Umwandlung von/nach Dezimalzeichenreihen

Das Ergebnis der Umwandlung muss korrekt gerundet werden, falls die Dezimalzeichenreihe in folgendem Bereich liegt:

	max D	max N
$R = 2$	$\lceil 1 + l \times \log_{10} 2 \rceil$	$\min l_{se} \times \log_5 2,$
$R = 10$	l	$10^{\lfloor \log_{10} E \rfloor + 1} - 1$ mit $E = \max\{D + (l - e1) \times \log_{10} R, 1 - D + e2 \times \log_{10} R\}$

Andernfalls muss gelten:

- Rundung zur nächsten Maschinenzahl für $R = 2$:
  Falls weder Überlauf noch Unterlauf eintritt, muss der Fehler gegenüber exakter Rundung kleiner als 0,5*ulp* des Ergebnisses sein;
  die sukzessive Umwandlung einer Gleitkommazahl in eine Dezimalzeichenreihe und deren Umwandlung in eine Gleitkommazahl muss für alle R die Identität ergeben.
- Gerichtete Rundungen für $R = 2$:
  Der Fehler muss korrektes Vorzeichen besitzen und kleiner als 1,5*ulp* des Ergebnisses sein.

In allen Fällen muss die Umwandlungsfunktion monoton sein.

Wesentliche Entscheidungen für IEEE-754R

Leitlinie der Revision war das Streben nach stärkerer Reproduzierbarkeit von Resultaten.

- Zusammenführung der Standards ANSI/IEEE-754 und ANSI/IEEE-854
- Keine weiteren Basen außer 2 und 10
- Ergänzung um externe Zahlensysteme in *quadruple precision*
- Ergänzung um interne Zahlensysteme (Speicherformate)
- Spezifikation von Repräsentationen auch für Basis 10
- Hinzunahme der Operation *fused multiply-add*
- Erweiterung um eine zusätzliche Rundung
- Einschränkung der Implementierungsfreiheiten
- Klarere Formulierung des Standardisierungstextes
- Saubere Trennung der verschiedenen Ebenen der Standardisierung
- Inhaltliche Begründung wird teilweise in den Standard übernommen

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Implementierung von Gleitkomma-Operationen

Gleitkomma-Multiplikation

Allgemeine Aufgabe: $(s_1, e_1) \otimes (s_2, e_2) \rightarrow (s_1 \otimes s_2, e_1 \oplus e_2)$

Speziell für Signifikanten in Vorzeichen/Betrag-Darstellung:
$(v_1, m_1, e_1) \otimes (v_2, m_2, e_2) \rightarrow (v_1 \oplus v_2, m_1 \otimes m_2, e_1 \oplus e_2)$

Addition der Exponenten, Multiplikation der Signifikanten und ggf. Vorzeichenbildung können gleichzeitig erfolgen.

Liegen die Exponenten in Biased-Darstellung vor, muss von ihrer formalen Summe der Bias abgezogen werden: $((e_1 + b) + (e_2 + b)) - b = (e_1 + e_2) + b$

Für den Bias $2^{e-1} = 10\ldots0$ geschieht dies durch Invertieren des höchstwertigen Exponentenbits.

Für den Bias $2^{e-1} - 1$ wird die zusätzliche 1 als Eingangsübertrag der Addition bereitgestellt.

Bei der Multiplikation der Signifikanten entsteht aus Operanden der Länge l ein Ergebnis der Länge $2 \times l$; grundsätzlich bilden dabei die höherwertigen l Ziffern das Resultat.

Mit Hilfe der niederwertigen l Ziffern werden Rundung und Normalisierung durchgeführt.

Es werden meist nicht alle Ziffern des exakten Ergebnisses benötigt.

Die über die eigentliche Darstellungsgenauigkeit hinausgehenden Ziffern heißen *Schutzziffern* (*Guard-Digits*).

Rundung nach Gleitkomma-Multiplikation

Die Anzahl der Rundungsschutzziffern hängt vom gewählten Rundungsmodus ab.

Dies soll hier an Hand von Signifikanten in Vorzeichen/Betrag-Darstellung gezeigt werden:

Rundung zur Null erfolgt durch Abschneiden, benötigt also keine Schutzziffern.

Für Rundung nach Unendlich muss feststellbar sein, ob alle abgeschnittenen Ziffern den Wert 0 besitzen; diese Information kann in einem einzigen Bit, dem *Sticky-Bit*, gespeichert werden.

Für gerade Basen genügt beim Rundungsmodus Round-to-nearest-out ein *Round-Digit*.

Alle übrigen Round-to-nearest-Rundungen benötigen zusätzlich ein Sticky-Bit bezüglich aller nach dem Round-Digit folgenden Ziffern.

Zur Berechnung des Sticky-Bits brauchen im Falle der Produktbildung nicht alle Ziffern des Ergebnisses berechnet werden, wenn mit binärer Darstellung gearbeitet wird.

Die Anzahl der niederwertigen Nullziffern eines Produkts binärcodierter Zahlen ist nämlich gleich der Summe der Anzahlen der niederwertigen Nullziffern von Multiplikator und Multiplikand.

Dieser Zusammenhang gilt so auch für andere Basen, die prim sind;
solche kommen allerdings normalerweise nicht vor.

Rundung durch Injektion

Statt den gerundeten Wert einer Gleitkomma-Multiplikation aus den Schutzziffern abzulesen, können alle Rundungsarten auf Abschneiden des Signifikanten zurückgeführt werden.

Je nach auszuführender Rundungsart wird ein bestimmter positiver Wert zum eigentlichen Multiplikationsergebnis hinzuaddiert und das Ergebnis dieser Aktion abgeschnitten.

Der Korrekturterm wird effizient bereits während der Ausführung der Multiplikation addiert.

In Vorzeichen/Betrag-Darstellung ist der Korrekturterm $R^l/2$ für Round-to-nearest-out, $R^l - 1$ für Rundung nach Unendlich.

Die Rundung nach oben bzw. nach unten entspricht je nach Vorzeichen der Rundung zur Null oder der Rundung nach Unendlich.

Andere Round-to-nearest-Rundungen werden durch spezielle Regeln zur Behandlung des Intervallmittelpunkts auf Round-to-nearest-out zurückgeführt.

Besonders einfach ist dies für Round-to-nearest-even.

Postnormalisierung

Liegt das ungerundete Ergebnis im Normalisierungsbereich und besitzt die berechnete Magnitude $m = m_1 \otimes m_2$ führende Nullen, muss sie vor der Rundung durch Linksverschiebung normalisiert werden (*Postnormalisierung*).

Wird von normalisierten Operanden m_1 und m_2 ausgegangen, tritt höchstens eine führende Null in m auf.

Dies erfordert dann eine einzelne Schutzziffer G (*Guard-Digit*).

Wird G nicht benötigt, dient G als Round-Digit; das Sticky-Bit wird entsprechend neu berechnet.

Beim Runden eines postnormalisierten Signifikanten kann sich, außer bei Rundung zur Null, wiederum ein Übertrag ergeben, der durch Rechtsverschiebung ausgeglichen werden muss.

Normalisierungsoperationen auf dem Signifikanten sind bei der Berechnung des Exponenten zu berücksichtigen.

Schnelle Gleitkomma-Multiplikation

Even und Seidel beschreiben in IEEE Computer, Juli 2000, einen sehr schnellen Gleitkomma-Multiplizierer. Folgende Techniken kommen dabei zur Anwendung:

- Die Berechnung des Signifikanten erfolgt mittels Carry-Save-Techniken.
- Die Rundung wird durch Injektion und Abschneiden durchgeführt.
- Während der Konvertierung des Signifikanten in Binärdarstellung werden die nächstgelegenen Maschinenzahlen parallel berechnet.
- Es ist pro Berechnungspfad nur eine Verschiebeoperation erforderlich.
- Die Herstellung des korrekt gerundeten Ergebnisses erfolgt abschließend durch Selektion.

Gleitkomma-Division

Allgemeine Aufgabe: $(s_1, e_1) \oslash (s_2, e_2) \rightarrow (s_1 \oslash s_2, e_1 \ominus e_2)$

Speziell für Signifikanten in Vorzeichen/Betrag-Darstellung:
$(v_1, m_1, e_1) \oslash (v_2, m_2, e_2) \rightarrow (v_1 \oplus v_2, m_1 \oslash m_2, e_1 \ominus e_2)$

Subtraktion der Exponenten, Division der Signifikanten und ggf. Vorzeichenbildung können gleichzeitig erfolgen.

Liegen die Exponenten in Biased-Darstellung vor, muss zu ihrer formalen Summe der Bias addiert werden: $((e_1 + b) - (e_2 + b)) + b = (e_1 - e_2) + b$

Für den Bias $2^{e-1} = 10\ldots0$ geschieht dies durch Invertieren des höchstwertigen Exponentenbits.

Für den Bias $2^{e-1} - 1$ wird die zusätzliche 1 als Eingangsübertrag der Subtraktion bereitgestellt.

Im Ablauf stimmen Gleitkomma-Multiplikation und Gleitkomma-Division weitgehend überein.

Eine aus prenormalisierten Operanden berechnete Magnitude kann ebenfalls höchstens eine führende Null aufweisen.

Es werden ein Guard-Digit und je nach Rundungsmodus ein Round-Digit und ein Sticky-Bit benötigt; das Sticky-Bit wird direkt aus dem Divisionsrest berechnet.

Gleitkomma-Addition/Subtraktion

Die Gleitkomma-Addition/Subtraktion ist komplizierter als die Gleitkomma-Multiplikation oder die Gleitkomma-Division; sie verläuft prinzipiell in mehreren Schritten:

(1) Berechnung von $d = |e_1 - e_2|$.

(2) Mantissenanpassung: Verschieben des Signifikanten des Operanden mit dem kleineren Exponenten um d Stellen nach rechts.

(3) Addition/Subtraktion des angepassten Signifikanten zum/vom Signifikanten des anderen Operanden; Exponent des Ergebnisses wird $\max(e_1, e_2)$.

(4) Postnormalisierung: Bei Addition der Signifikanten höchstens ein Rechtsshift; bei Subtraktion bis zu l Linksshifts (Auslöschung).

Zur korrekten Rundung genügen wieder ein Guard-Digit, ein Round-Digit und ein Sticky-Bit, die beim Mantissenangleich gewonnen werden.

Mantissenanpassung und Postnormalisierung benötigen relativ viel Zeit und sollten nach Möglichkeit vermieden werden; die Wahrscheinlichkeit der Notwendigkeit dieser Anpassungen nimmt mit wachsender Basis R ab (die Postnormalisierung ist generell recht selten).

Schnelle Gleitkomma-Addition/Subtraktion

Wird vor Subtraktion der Signifikanten eine Mantissenanpassung um mehr als eine Stelle vorgenommen, ist die Postnormalisierungsdistanz höchstens Eins.

Mantissenanpassungen um $l + 2$ Stellen genügen,
größere Exponentendifferenzen brauchen nicht vollständig ausgerechnet werden.

Um die Latenz gepipelineter Addierer/Subtrahierer klein zu halten, können alternative Berechnungspfade abhängig von der Weite der Mantissenanpassung gewählt werden.

Die Postnormalisierungsdistanz kann frühzeitig abgeschätzt werden:

Wird statt einer Subtraktion von vorzeichenlosen Zahlen eine Addition von 2-Komplement-Zahlen implementiert, berechnet man die Addiererhilfssignale $G_i = A_i \wedge B_i$, $P_i = A_i \oplus B_i$ und $K_i = \overline{A}_i \wedge \overline{B}_i$.

Sei $P_1 = P_2 = \ldots = P_i = G_{i+1} = K_{i+2} = \ldots = K_j = 1, K_{j+1} = 0$
oder $P_1 = P_2 = \ldots = P_i = K_{i+1} = G_{i+2} = \ldots = G_j = 1, G_{j+1} = 0$.

Dann beträgt die Postnormalisierungsdistanz entweder j oder $j - 1$.

Gleitkomma-Ausnahmen

Wird der Exponent eines Gleitkomma-Ergebnisses zu klein, tritt *Unterlauf* auf.

Als Abhilfe kann der Exponent erhöht und der Signifikant nach rechts verschoben werden (*Denormalisierung*).

Dies kann zum Verlust signifikanter Ziffern führen (*Inexact-Exception*).

Es ist auch hier auf korrekte Rundung bezüglich des exakten Ergebnisses zu achten!

Wird der Exponent eines Gleitkomma-Ergebnisses zu groß, tritt *Überlauf* auf.

Hierfür gibt es keine Abhilfe, da durch Linksshift höchstwertige Ziffern des Signifikanten verloren gehen würden.

Beide Ausnahmen können auch das Ergebnis einer Postnormalisierung sein.

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Logarithmische Zahlensysteme

Logarithmische Zahlensysteme: Einsatzbereich

Logarithmische Zahlensysteme (logarithmic number systems LNS) dienen der Vereinfachung von

- Multiplikation (wird zu gewöhnlicher Addition)
- Division (wird zu gewöhnlicher Subtraktion)
- Potenzierung (wird zu gewöhnlicher Multiplikation)
- Radizieren (wird zu gewöhnlicher Division)

Dagegen sind Addition und Subtraktion in logarithmischer Darstellung komplizierter als in gewöhnlicher Darstellung.

Logarithmische Zahlensysteme sind insbesondere gut geeignet für Signalverarbeitung mit geringen Genauigkeitsanforderungen und beabsichtigtem reduziertem Energieaufwand

Typische Verteilungen von Filter-Koeffizienten passen empirisch zu logarithmischer Darstellung

Laufendes Projekt: European Logarithmic Microprocessor (ESPRIT-Projekt, Beginn 1999)

European Logarithmic Microprocessor

- Coleman, J.N.; Softley, C.I.; Kadlec, J.; Matousek, R.; Licko, M.; Pohl, Z.; Hermanek, A.:
  The European Logarithmic Microprocessor - a QR RLS application.
  Conference Record of the Thirty-Fifth Asilomar Conference on Signals, Systems and Computers, Volume 1, 2001 pp. 155–159.

  Summary: In contrast to all other microprocessors, which use floating-point for their real arithmetic, the European Logarithmic Microprocessor is the world's first device to use the logarithmic number system for this purpose. Simulation work has already suggested that this can deliver approximately twofold improvements in speed and accuracy. This paper describes the ELM device, and illustrates its operation using an example from a class of RLS algorithms.
- Coleman, J.N.; Softley, C.I.; Kadlec, J.; Matousek, R.; Tichy, M.; Pohl, Z.; Hermanek, A.; Benschop, N.F.:
  The European Logarithmic Microprocesor.
  IEEE Transactions on Computers, April 2008 (Vol. 57, No. 4) pp. 532–546.

  Abstract: In 2000 we described a proposal for a logarithmic arithmetic unit, which we suggested would offer a faster, more accurate alternative to floating-point procedures. Would it in fact do so, and could it feasibly be integrated into a microprocessor so that the intended benefits might be realised? Herein we describe the European Logarithmic Microprocessor, a device designed around that unit, and compare its performance with that of a commercial superscalar pipelined floating-point processor. We conclude that the experiment has been successful; that for 32-bit work logarithmic arithmetic may now be the technique of choice.

Logarithmische Zahlensysteme: Konversion

Die Konversion zwischen logarithmischen Zahlensystemen und Standarddarstellungen erfordert die Berechnung von Logarithmen und Antilogarithmen.

Wegen der dabei auftretenden Approximationsfehler sind die Operanden bzw. Ergebnisse in logarithmischer Darstellung ungenau.

Typische Anwendungsfelder (z. B. digitale Filter) erfordern wenige Konversionen, aber viele Multiplikationen oder Divisionen.

Von der Darstellung alleine her gesehen, sind logarithmische Zahlensysteme sogar etwas genauer als Gleitkommasysteme annähernd gleichen Zahlenbereichs.

Vorzeichen-Logarithmus-Darstellung (SL)

Die *Vorzeichen-Logarithmus-Darstellung* (sign logarithm SL) einer zu codierenden reellen Zahl X besteht aus dem Vorzeichenbit S_X und dem Logarithmus L_X des Betrags von X,
$X = (-1)^{S_X} \times R^{L_X}$.

Falls $|X| < 1$ zugelassen ist, muss L_X auch negative Werte annehmen können.

Da $\log 0$ nicht definiert ist, muss außerdem die Null eine spezielle Darstellung besitzen.

Meist wird $R = 2$ gewählt und L_X in einer binären Festkommadarstellung angegeben:

$$S_X\, L_X = S_X\, x_{k-1}\, x_{k-2} \ldots x_1\, x_0 . x_{-1}\, x_{-2} \ldots x_{-n}$$

Der Nachkommateil besitzt hier n Bits, der ganzzahlige Anteil k Bits (einschließlich eines eventuellen Vorzeichenbits des Logarithmus).

Beispiel: SL mit $k = 4$, $n = 3$

Mit L_X in 2-Komplement-Darstellung ohne Bias gilt z. B.

$00111.111 = +2^{(8-\frac{1}{8})}$	$\approx +234.753_{10}$	(größte positive Zahl)
$00001.010 = +2^{(1+\frac{1}{4})}$	$\approx +2.37841_{10}$	
$01110.100 = +2^{-(1+\frac{1}{2})}$	$\approx +0.35355_{10}$	
$01000.000 = +2^{-8}$	$\approx +0.003906_{10}$	(kleinste positive Zahl)
spezielle Darstellung		(Null)
$11000.000 = -2^{-8}$	$\approx -0.003906_{10}$	(betragskleinste negative Zahl)
$11110.100 = -2^{-(1+\frac{1}{2})}$	$\approx -0.35355_{10}$	
$10001.010 = -2^{(1+\frac{1}{4})}$	$\approx -2.37841_{10}$	
$10111.111 = -2^{(8-\frac{1}{8})}$	$\approx -234.753_{10}$	(betragsgrößte negative Zahl)

Vereinfachung arithmetischer Operationen

Operationen ohne Rundungsfehler:

Multiplikation	$(S_X, L_X) \times (S_Y, L_Y) = (S_X \oplus S_Y, L_X + L_Y)$	(Festkomma-Addition)
Division	$(S_X, L_X)/(S_Y, L_Y) = (S_X \oplus S_Y, L_X - L_Y)$	(Festkomma-Subtraktion)
Kehrwert	$1/(S_X, L_X) = (S_X, -L_X)$	(Komplement)
Quadrat	$(S_X, L_X)^2 = (0, 2 \times L_X)$	(Linksverschiebung)
ganze Potenz	$(S_X, L_X)^Z = (S_X \wedge (Z \bmod 2), Z \times L_X)$	(Festkomma-Multiplikation)

Operationen, bei denen ein Rundungsfehler auftreten kann:

Wurzel	$\sqrt{(0, L_X)} = (0, L_X/2)$	(Rechtsverschiebung)
Potenz	$(0, L_X)^Y = (0, Y \times L_X)$	(Festkomma-Multiplikation)

Überlauf und Unterlauf können in allen Fällen leicht erkannt werden.

Addition und Subtraktion von SL-Zahlen

Für Addition und Subtraktion in logarithmischen Zahlensystemen existieren verschiedene alternative Ansätze:

1. Wertetabelle der Größe $l \times 2^{2 \times l}$ bit (mit $l = k + n$), nicht praktikabel für übliche Werte von l.
2. Die Operanden werden delogarithmiert und mit gewöhnlicher Addition behandelt, das Ergebnis wird logarithmiert (Rückgriff auf Wertetabellen der Größe $l \times 2^l$ bit, die für die Konvertierung in bzw. aus Standarddarstellungen ohnehin gebraucht werden).
3. Direkte Berechnung einer approximativen Summe oder Differenz (wegen kleinerer Wertetabellen der bevorzugte Ansatz).

Approximative Summen und Differenzen

Seien o. B. d. A. $X, Y > 0$.

$X > Y$:	$Z = X + Y = X \times (1 + \frac{Y}{X})$,		$Z = X - Y = X \times (1 - \frac{Y}{X})$,
		also	
	$L_Z = \log_2 X + \log_2(1 + \frac{Y}{X})$		$L_Z = \log_2 X + \log_2(1 - \frac{Y}{X})$
	$= L_X + \Phi^+(L_X - L_Y)$		$= L_X + \Phi^-(L_X - L_Y)$
		mit	
	$\Phi^+(h) = \log_2(1 + 2^{-h})$, $h > 0$		$\Phi^-(h) = \log_2(1 - 2^{-h})$, $h > 0$
$X < Y$:	$L_Z = L_Y + \Phi^+(L_Y - L_X)$		$L_Z = L_Y + \Phi^-(L_Y - L_X)$
$X = Y$:	$L_Z = L_X + 1$		$Z = 0$

Implementierung von Φ^+ und Φ^-

Φ^+ und Φ^- können durch Wertetabellen der Größe $l \times 2^l$ bit implementiert werden.
Um eine möglichst hohe Genauigkeit der Addition/Subtraktion zu garantieren, werden die exakten Werte von $\log_2(1 \pm 2^{-h})$ zum Eintrag in die Wertetabelle Round-to-nearest gerundet.

Wegen $0 < \Phi^+(h) < 1$ braucht bei der Addition nur ein Nachkommateil erzeugt werden.
Statt einer $(l \times 2^l)$-bit Wertetabelle genügt also eine $(n \times 2^l)$-bit Wertetabelle.

Für große h ist $\log_2(1 \pm 2^{-h}) \approx \pm 2^{-h}$, und damit fast Null; wegen der beschränkten Genauigkeit werden diese Werte durch die Rundung zu Null und es macht keinen Sinn, sie explizit zu speichern.

Die Wertetabellen können auch in eine Reihe kleinerer Tabellen zerlegt werden, in denen jeweils ein ähnlicher Effekt ausgenutzt werden kann.

Bei Kombination von Wertetabellen und Interpolation müssen weniger Einträge in der Wertetabelle gespeichert werden; hierzu gibt es ziemlich raffinierte, schnelle Verfahren.

Konversionen

Festkommaformat $\leftrightarrow$ logarithmische Darstellung:

$$v \times m = v \times R^L$$

Gleitkommaformat $\leftrightarrow$ logarithmische Darstellung:

$$v \times m \times R^e = v \times R^{L+e}$$

Es wird also im Prinzip nur eine Logarithmentafel für $1 \leq m < R$ und eine Antilogarithmentafel für $0 \leq L < 1$ benötigt.

Berechnung der Logarithmen

Sei $m = m_u m_{u-1} \ldots m_0 . m_{-1} m_{-2} \ldots m_{-w}$ und $t = \max\{i : m_i = 1\}$.

Es gilt $m = 2^t + \sum_{i=-w}^{t-1} 2^i \times m_i = 2^t \times \left(1 + \sum_{i=-w}^{t-1} 2^{i-t} \times m_i\right) = 2^t \times (1+h)$ mit $0 \leq h < 1$.

Also $\log_2 m = t + \log_2(1+h)$,
wobei t der ganzzahlige Anteil des Logarithmus ist, $\log_2(1+h)$ der Nachkommateil.

Meist wird $\log_2(1+h) \approx h$ ausgenutzt.

Eine Verbesserung ergibt sich durch geschickte Unterteilung des Intervalls $[0, 1)$ für h.

Die Implementierung erfolgt mittels eines Zählers und eines Schieberegisters.

Verallgemeinerung

Index Calculus Double-Base Number System (IDBNS), $y = v \times 2^a \times 3^b, a, b \in \mathbb{Z}$

Eigenschaft: $\forall \varepsilon > 0, x \geq 0\ \exists a, b \in \mathbb{Z} : |x - 2^a \times 3^b| < \varepsilon$

Andere Basen, mehr Ziffern (n digit two-dimensional logarithmic representation):

$$y = \sum_{i=1}^{n} v_i \times 2^{a_i} \times p^{b_i}, \quad p \text{ ungerade}$$

Beispiel: Darstellung mit Fehler $\leq 0{,}5ulp$

Standard-Darstellung: $x \in \mathrm{Int}_2(10)$ (10-Bit-Darstellung)

SL-Repräsentation erfordert 12 Bit Logarithmus und ein Vorzeichenbit

Repräsentation durch zweistelliges 2-D LNS ($n = 2, p = 47$): $a_i \in \mathrm{Int}_2(6)$, $b_i \in \mathrm{Int}_2(3)$
$334 \approx 2^9 \times 47^{-1} + 2^{25} \times 47^{-3} \approx 334{,}082429$

Ziel der Darstellung mit mehreren Basen: Verwendung kleinerer Tabellen

Arithmetik mit variierender relativer Genauigkeit

Zweck variierender relativer Genauigkeit:

- Verbesserung der *durchschnittlichen* relativen Genauigkeit.
- Vergrößerung des Zahlenbereichs zur Vermeidung von Überlauf oder Unterlauf.

Die durchschnittliche relative Genauigkeit lässt sich durch *Tapered-Floating-Point-Systeme* verbessern; als Nebeneffekt ergibt sich zusätzlich eine gewisse Vergrößerung des Zahlenbereichs.

Eine entscheidende Vergrößerung des Zahlenbereichs wird erreicht durch verschiedene Methoden des *Leveling*.

Tapered-Floating-Point-Systeme

Ansatz von Morris (1971), aufgegriffen von Iri und Matsui (1981).

Statt eines Gleitkommaformats mit Signifikanten- und Exponentenfeldern fester Länge besteht ein *Tapered-Floating-Point-Format* aus Signifikanten- und Exponentenfeldern variierender Länge.

Hinzu kommt ein *Pointerfeld* fester Länge, das die Anzahl der Ziffern im Exponentenfeld angibt.

Die Anzahl der Ziffern im Signifikantenfeld (einschließlich des Vorzeichens der Zahl) ergibt sich als Differenz der festen Gesamtlänge des Formats abzüglich der Längen des Exponenten- und Pointerfelds.

Tapered-Floating-Point-Systeme: Implementierung

Die Anzahl der Ziffern im Exponentenfeld sollte für jeden konkreten Exponenten minimal gewählt werden, um eine möglichst hohe Genauigkeit zu ermöglichen.

Liegt der Signifikant bzw. Exponent in Binärcodierung vor, so gilt:

- Die führende Ziffer des Betrags des Signifikanten bzw. Exponenten braucht nicht gespeichert zu werden, da sie stets den Wert 1 trägt (Hidden-Bit).
- Ein Signifikantenfeld bzw. Exponentenfeld mit nur einem Bit kann die Signifikanten bzw. Exponenten ± 1 darstellen (besteht nur aus dem Vorzeichen).
- Ein Signifikantenfeld der Länge Null zeigt die Zahl Null an.
- Ein Exponentenfeld der Länge Null codiert den Exponenten 0.

Eigenschaften von Tapered-Floating-Point-Formaten

Vorteile:

- Im gewöhnlichen Zahlenbereich von Gleitkommasystemen können Zahlen, die weder sehr groß noch sehr klein sind, mit erhöhter Genauigkeit gespeichert werden.
- Der Zahlenbereich kann durch geringere Genauigkeit für die hinzukommenden Zahlen wesentlich erweitert werden.

Nachteile:

- Die Implementierung ist aufwändiger als für gewöhnliche Gleitkommasysteme.
- Bei gleichem Speicheraufwand ist für manche Zahlen im gewöhnlichen Zahlenbereich entsprechender Gleitkommasysteme die Genauigkeit wegen des Pointerfelds geringer.

Praktische Implementierungen zeigen, dass bei gleichem Speicheraufwand durch Tapered-Floating-Point-Systeme in der Regel eine höhere durchschnittliche Genauigkeit erreicht wird.

Genauigkeit von Tapered-Floating-Point-Systemen

Betragsgroße und betragskleine Zahlen sind wegen des betragsgroßen Exponenten ungenauer als solche moderater Magnitude.

Beispiel

Länge 64 Bit, davon 6 Bit für das Pointerfeld: Zahlenbereich $\approx \pm 10^{\pm 4 \times 10^{16}}$

$\lvert x \rvert \approx$	relativer Fehler beschränkt durch	
1	2^{-57}	(höchste relative Genauigkeit)
$2^{\pm 16}$	2^{-52}	
$10^{\pm 8 \times 10^{7}}$	2^{-28}	
$10^{\pm 4 \times 10^{16}}$	100%	(soweit x im zulässigen Bereich)

Das Format *double* im Standard IEEE-754 besitzt einen Zahlenbereich von $\approx \pm 10^{\pm 308}$ bei gleichmäßigem maximalen relativen Fehler $\approx 2^{-52}$.

Wahrscheinlichkeit von Überlauf und Unterlauf

Bei Tapered-Floating-Point-Systemen sind Überlauf und Unterlauf wegen des erweiterten Exponentenbereichs recht unwahrscheinlich, können aber – insbesondere bei wiederholter Exponentiation – dennoch auftreten.

Mit *Leveling* erreicht man Größenordnungen, die in der Praxis nicht mehr vorkommen.
Damit ist zwar nicht ausgeschlossen, dass Überlauf oder Unterlauf auftritt, die Wahrscheinlichkeit ist aber so gut wie Null.

Zu beachten ist, dass die Operationen $+, -, \times, /$ für höhere Levels hochgradig ungenau ausfallen.

Leveling nach Iri und Matsui

Ist der Exponent einer darzustellenden Zahl betragsgrößer als der maximale Exponent eines Tapered-Floating-Point-Formates (Level 0), so wird der Betrag dieses Exponenten selbst in einem (etwas kleineren) Tapered-Floating-Point-Format (Level 1) repräsentiert.

Ein spezieller Wert des Pointerfelds zeigt an, wann der Exponent im Level 1 Format vorliegt.

Beispiel

Level 0: Länge 64 Bit, davon 6 Bit Pointerfeld; Zahlenbereich $\approx \pm 10^{\pm 4 \times 10^{16}}$

Level 1: Länge 56 Bit, davon 6 Bit Pointerfeld; Zahlenbereich $\approx \pm 10^{\pm 10^{10^{15}}}$

Dieses Prinzip kann über weitere Stufen (Level 2, Level 3, ...) fortgesetzt werden, bis die immer kleiner werdenden Exponentenfelder dem eine Grenze setzen.

Mit fortschreitendem Level wird die Darstellung aber immer ungenauer.

Level-Index-Arithmetik

Level-Index (LI)

Clenshaw und Olver (1984)

Zahlenformat: Vorzeichen $v \in \{0,1\}$ der Zahl, vorzeichenlose Festkommazahl f zur Codierung des Betrags.

Der ganzzahlige Anteil von f heißt *Level*, der Nachkommateil von f heißt *Index*.

$$x = (-1)^v \times \Phi(f)$$

$$\Phi(f) = \begin{cases} f & \text{falls } 0 \leq f \leq 1 \\ e^{\Phi(f-1)} & \text{sonst} \end{cases}$$

Symmetric Level-Index (SLI)

Clenshaw/Olver/Turner (1987)

Zahlenformat: Vorzeichen $v \in \{0,1\}$ der Zahl, Vorzeichen $u \in \{0,1\}$ des Exponenten, vorzeichenlose Festkommazahl f zur Codierung des Betrags.

$$x = (-1)^v \times \Psi(f)^{(-1)^u}$$

$$\Psi(f) = \begin{cases} e^f & \text{falls } 0 \leq f \leq 1 \\ e^{\Psi(f-1)} & \text{sonst} \end{cases}$$

Symmetric-Level-Index-Arithmetik (Beispiel)

Länge 64 Bit, davon 3 Bit Levelfeld: Zahlenbereich $\approx \pm 10^{\pm 10^{10^{10^{10}}}}$

$|x| \approx 1$: relativer Fehler $\approx 2^{-59}$ (höchste Genauigkeit)

$|x| \approx 1.73 \times 10^{13}$: relativer Fehler $\approx 2^{-52}$

$|x| \approx 10^{5 \times 10^6}$: relativer Fehler $\approx 2^{-30}$

$|x| \approx 10^{10^{15}}$: relativer Fehler ≈ 1

Für noch größere $|x|$ kann der relative Fehler unvorstellbar groß werden.

Dynamische Vereinbarung von Zahlformaten

Cohen/Hamacher/Hull (1981):
Clean Arithmetic with Decimal base And Controlled precision (CADAC)

Hull et al. (1985): NUMERICAL TURING

- Zahlenbasis 10
- Nachkommateile mit p Stellen
- Ganzzahlige Exponenten im Bereich $[-10 \times p, +10 \times p]$

An jeder Stelle des Programms kann eine *precision* p in Form eines dynamisch ausgewerteten Ausdrucks erklärt werden.

Eine *precision*-Definition legt für den Rest des Gültigkeitsbereiches der die Definition enthaltenden Kontrollstruktur bzw. bis zum nächsten Auftreten einer *precision*-Definition innerhalb derselben Kontrollstruktur die Genauigkeit p aller deklarierten Variablen und aller Gleitkommaoperationen fest.

Jede Variable kann alternativ auch direkt mit einem Ausdruck versehen werden, der ihre Genauigkeit p angibt.

Algorithmen in Numerical Turing

- Hohe Genauigkeit für den gesamten Algorithmus ist aufwändig.
- Hohe Genauigkeit wird meist nicht durchgängig benötigt (nur an kritischen Stellen).
- Der Grad an nötiger Genauigkeit hängt oft von den Daten ab, ist also zur Übersetzungszeit schwer abschätzbar.

Beispiel: Berechnung der Quadratwurzel nach dem Newton-Verfahren:

```
var p:= 3
const maxp := currentprecision+2
loop
   p := min(2*p-2,maxp)
            % p = 4, 6, 10, ..., maxp
   precision p
   approx := .5*(approx+f/approx)
   exit when p = maxp
end loop
```

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Intervallarithmetik

Einschluss in Intervalle

Liegt als Ergebnis einer Maschinenoperation (arithmetische Operation, Konvertierung) eine Zahl nicht exakt vor, so lässt ihr gespeicherter Wert alleine keine Rückschlüsse auf den Approximations- bzw. Rundungsfehler zu.

Abhilfe schafft eine Intervalldarstellung:
Die eigentlich gesuchte, nicht genau bekannte und evtl. nicht darstellbare Zahl a wird zwischen den exakt darstellbaren gesicherten Grenzen a_1 und a_2 eines reellen Intervalls A eingeschlossen,

$$A = [a_1, a_2] = \{a \mid a_1 \leq a \leq a_2\}$$

Liegt eine solche Abschätzung durch das Paar (a_1, a_2) vor, so kennt man zwar nicht den exakten Wert von a, kann aber $a_1, a_2, (a_1 + a_2)/2$ oder jeden anderen beliebigen Wert zwischen a_1 und a_2 als Näherung ansehen.

Der absolute Fehler besitzt dabei höchstens den Wert $a_2 - a_1$.

Der relative Fehler kann für $0 \notin [a_1, a_2]$ durch $\dfrac{a_2 - a_1}{\min\{|a_1|, |a_2|\}}$ von oben abgeschätzt werden.

Intervallrundung

Zu einem Zahlenbereich $S \subset \mathbb{R}^{\pm\infty}$ mit $\pm\infty \in S$ werden Intervallräume betrachtet:
$\mathbb{IR} = \{[x_1, x_2] \mid x_1, x_2 \in \mathbb{R}^{\pm\infty}, x_1 \leq x_2\}$ (nicht leere, reelle Intervalle)
$\mathbb{IS} = \{[x_1, x_2] \mid x_1, x_2 \in S, x_1 \leq x_2\} \subset \mathbb{IR}$ (Maschinenintervalle)

Eine Abbildung $\Diamond\colon \mathbb{IR} \to \mathbb{IS}$ heißt *Intervallrundung*, wenn gilt:
(IR1) $\forall X \in \mathbb{IS}\colon \Diamond X = X$ (Invarianz in $\mathbb{IS}$)
(IR2) $\forall X, Y \in \mathbb{IR}\colon X \subseteq Y \Rightarrow \Diamond X \subseteq \Diamond Y$ (Monotonie in $\mathbb{IR}$)
(IR3) $\forall X \in \mathbb{IR}\colon X \subseteq \Diamond X$ (Rundung nach außen)

Die Intervallrundung $\Diamond$ ist durch (IR1), (IR2) und (IR3) wohldefiniert und maximal genau:
$\forall X = [x_1, x_2] \in \mathbb{IR}\colon \Diamond X = [\max\{x \in S \mid x \leq x_1\}, \min\{x \in S \mid x_2 \leq x\}] = [\triangledown x_1, \triangle x_2]$

Die Intervallrundung lässt sich fortsetzen zu einer Operation $\Diamond\colon \mathbb{P}(\mathbb{R}^{\pm\infty}) \to \mathbb{IS}$:
$\forall Y \subseteq \mathbb{R}^{\pm\infty}\colon \Diamond Y = [\max\{x \in S \mid \forall y \in Y\colon x \leq y\}, \min\{x \in S \mid \forall y \in Y\colon y \leq x\}]$

Maschinenintervalloperationen

Beim Rechnen mit Intervallen sollen aus gesicherten Intervallschranken der Operanden wieder gesicherte Intervallschranken der Ergebnisse hervorgehen.

Für jede aus einer zweistelligen Operation $\circ$ abgeleitete Maschinenintervalloperation $\diamondsuit$ hat daher zu gelten:

$$[a_1, a_2] \circ [b_1, b_2] = \{a \circ b \mid a_1 \leq a \leq a_2, b_1 \leq b \leq b_2\} \subseteq [a_1, a_2] \diamondsuit [b_1, b_2]$$

Eine solche Maschinenintervalloperation heißt *maximal genau*, wenn gilt:

$$\forall X, Y \in \mathrm{IS}: X \diamondsuit Y = \Diamond(X \circ Y)$$

Ebenso hat für die Anwendung von Standardfunktionen f auf Maschinenintervalle zu gelten:

$$f[a_1, a_2] = \{f(a) \mid a_1 \leq a \leq a_2\} \subseteq [b_1, b_2] = f^{\Diamond}[a_1, a_2]$$

Eine Intervallstandardfunktion ist maximal genau, wenn $f^{\Diamond}[a_1, a_2] = \Diamond f[a_1, a_2]$ gilt.

Implementierung maximal genauer Intervalloperationen

Maximal genaue Implementierungen der Maschinenintervalloperationen $\diamondsuit$
für $\circ \in \{+, -, \times, /\}$ stellen kein prinzipielles Problem dar;
sie werden realisiert durch

$$[a_1, a_2] \diamondsuit [b_1, b_2] = [a_1 \mathbin{\triangledown} b_1, a_2 \mathbin{\triangle} b_2]$$

$$[a_1, a_2] \diamondsuit [b_1, b_2] = [a_1 \mathbin{\triangledown} b_2, a_2 \mathbin{\triangle} b_1]$$

$$[a_1, a_2] \diamondsuit [b_1, b_2] = [\min_{i,j=1,2}\{a_i \mathbin{\triangledown} b_j\}, \max_{i,j=1,2}\{a_i \mathbin{\triangle} b_j\}]$$

$$[a_1, a_2] \diamondsuit [b_1, b_2] = [\min_{i,j=1,2}\{a_i \mathbin{\triangledown} b_j\}, \max_{i,j=1,2}\{a_i \mathbin{\triangle} b_j\}], \text{ falls } 0 \notin [b_1, b_2]$$

Dagegen ist eine maximal genaue Implementierung von Intervallstandardfunktionen
häufig problematisch.

Es wird deshalb für Standardfunktionen in der Regel zugelassen, dass zwischen einer
berechneten Intervallgrenze und ihrem exakten Wert noch eine weitere Maschinenzahl liegt.

Intervallarithmetik in Vektorräumen

Werte aus $\mathbb{R}^{(n)}$, $\mathbb{R}^{(m,n)}$, etc. können repräsentiert werden, indem für jede ihrer Komponenten ein Intervall angegeben wird, in dem sämtliche in Frage kommenden Werte der Komponente gesichert eingeschlossen sind.

Für zweistellige Matrixoperationen $\circ$ hat dann zu gelten:

$$([a_1, a_2]^{(i,j)}) \circ ([b_1, b_2]^{(i,j)}) = \{(a^{(i,j)}) \circ (b^{(i,j)}) \mid a^{(i,j)} \in [a_1, a_2]^{(i,j)}, b^{(i,j)} \in [b_1, b_2]^{(i,j)}\}$$

$$\subseteq ([c_1, c_2]^{(i,j)}) = ([a_1, a_2]^{(i,j)}) \diamondsuit ([b_1, b_2]^{(i,j)})$$

Entsprechendes gilt für $\mathbb{C}, \mathbb{C}^{(n)}, \mathbb{C}^{(m,n)}$, etc.

Intervalloperationen in Vektorräumen werden auf skalare Intervalloperationen zurückgeführt.

Effizienter Einsatz der Intervallarithmetik

Im praktischen Einsatz tendieren die Intervallgrenzen bei naiver Anwendung meistens zur Aufblähung, die Aussagekraft lässt dann schnell nach.

Notwendige Voraussetzungen für eine erfolgreiche Anwendung der Intervallarithmetik sind

- indirekte Berechnung durch iteratives Lösen von linearen Intervallgleichungssystemen,
- maximal genaue Auswertung von Skalarprodukten:

$$(a_1, \ldots, a_n) \diamondsuit (b_1, \ldots, b_n) = \diamondsuit \left(\sum_{k=1}^{n} a_k \times b_k \right)$$

Maximal genaue Auswertung von Skalarprodukten

Die effizienteste Methode ist „Auflaufenlassen“ unter Verwendung eines genügend langen Festkommaregisters, des sogenannten Skalarprodukt-Akkumulators.

Der Skalarprodukt-Akkumulator muss Summen doppelt langer Produkte beliebiger Zahlen im Ausgangsformat exakt darstellen können und sollte über einige zusätzliche Schutzziffern zum Auffangen zwischenzeitlicher Überläufe bei der Summation verfügen.

Zahlen nach ANSI/IEEE-754 erfordern mindestens 556 Bit (*single*) bzw. 4198 Bit (*double*).

Die Berechnung von Zwischenergebnissen erfolgt rundungsfrei:
Gerundet wird nur ein einziges Mal, nämlich am Schluss der Akkumulation.

Die Ausführung kann meist schneller als in gewöhnlicher Gleitkomma-Arithmetik erfolgen, da für Zwischenergebnisse Mantissenanpassung, Normalisierung, Runden, Packen und Entpacken entfallen.

Lösung linearer Gleichungssysteme in klassischer Arithmetik

Ergebnisse, die in klassischer Arithmetik berechnet wurden, können beliebig falsch sein, ohne dass dies auch nur erkennbar wäre.

Beispiel: Zu lösen sei das lineare Gleichungssystem

$$\begin{pmatrix} 64919121{,}0 & -159018721{,}0 \\ 41869520{,}5 & -102558961{,}0 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1{,}0 \\ 0{,}0 \end{pmatrix}$$

Klassische Arithmetik, z. B. mittels MS-Excel 7.0, liefert

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 102558961{,}0 \\ 41869520{,}5 \end{pmatrix}$$

Das korrekte Result, berechnet z. B. mit Pascal-XSC, ist

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 205117922{,}0 \\ 83739041{,}0 \end{pmatrix}$$

Iteratives Lösen linearer Intervallgleichungssysteme

Zu lösen sei $A \cdot x = b$ mit $A \in \mathbb{R}^{(n,n)}$ und $x, b \in \mathbb{R}^{(n)}$.

Mit geeigneten Näherungen $x^{(0)} \approx x$ und $R \approx A^{-1}$ ergibt sich eine Defekt-Iteration mittels der vereinfachten Newton-Vorschrift nach der Formel

$$x^{(k+1)} = x^{(k)} - R \cdot (A \cdot x^{(k)} - b)$$

Zur Steigerung der Genauigkeit der Intervallrechnung wird zunächst äquivalent umgeformt:

$$x^{(k+1)} = R \cdot b + (I - R \cdot A) \cdot x^{(k)}$$

Überführung der Iterationsvorschrift in einen Maschinenintervallalgorithmus liefert schließlich

$$X^{(k+1)} = R \diamond b \oplus \diamondsuit(I - R \cdot A) \diamond X^{(k)}$$

Mit etwas höherem Aufwand lässt sich die Genauigkeit noch weiter steigern:

$$X^{(k+1)} = \diamondsuit(R \cdot b + \diamondsuit(I - R \cdot A) \cdot X^{(k)})$$

Iteratives Lösen linearer Intervallgleichungssysteme: 3E-Methoden

Mit Hilfe funktionalanalytischer Methoden lässt sich dieser Algorithmus so modifizieren,
dass die *Existenz* einer Lösung x durch gesicherten *Einschluss* in einem Intervall Y gezeigt
und gleichzeitig die *Eindeutigkeit* einer solchen Lösung in Y bewiesen wird;
dies verschärft den

Fixpunktsatz von Brouwer:

Es sei $\phi\colon \mathbb{R}^n \to \mathbb{R}^n$ eine stetige Abbildung
und $X \subseteq \mathbb{R}^n$ nicht leer, konvex, abgeschlossen und beschränkt.
Es sei $Y = \phi(X)$.
Gilt $Y \subseteq X$, so besitzt ϕ mindestens einen Fixpunkt in Y.

Wir wenden zunächst diesen Satz auf $\phi(x) = R \cdot b + (I - R \cdot A) \cdot x$ an.

Ergibt sich bei Durchführung der Iterationsvorschrift die Beziehung $X^{(k+1)} \subseteq X^{(k)}$,
so kann für reguläres R auf die Existenz einer Lösung x im Intervall $X^{(k+1)}$ geschlossen werden:

$$
\begin{aligned}
& X^{(k+1)} \subseteq X^{(k)} \\
\Rightarrow \quad & R \cdot b + (I - R \cdot A) \cdot X^{(k)} \subseteq X^{(k)} \quad \text{wegen } R \cdot b + (I - R \cdot A) \cdot X^{(k)} \subseteq X^{(k+1)} \\
\Rightarrow \quad & \exists x \in X^{(k+1)} \colon R \cdot (A \cdot x - b) = 0 \quad \text{nach dem Fixpunktsatz von Brouwer} \\
\Rightarrow \quad & \exists x \in X^{(k+1)} \colon A \cdot x = b \quad \text{wegen der Regularität von } R
\end{aligned}
$$

Iteratives Lösen linearer Intervallgleichungssysteme: Eindeutigkeit

Zur Garantierung der Existenz einer Lösung mit dem Fixpunktsatz von Brouwer
muss die Regularität von R nachgewiesen werden,
für deren Eindeutigkeit zusätzlich noch die Regularität von A.

Durch Verschärfen der Voraussetzungen des Satzes kann dieser Nachweis implizit erfolgen:
Theorem (Kulisch/Rump)
Es sei $X \subseteq \mathbb{R}^n$ nicht leer, konvex, abgeschlossen und beschränkt.
Gilt $\phi(X) \subseteq \mathring{X}$ mit $\phi(x) = R \cdot b + (I - R \cdot A) \cdot x$,
dann sind A und R regulär,
es gibt genau ein $x \in \mathbb{R}$ mit $Ax = b$
und es gilt $x \in \phi(X)$.

Dabei ist $\mathring{X}$ das *Innere* der Menge X.
Die Bedingung $\phi(X) \subseteq \mathring{X}$ heißt *Retraktionseigenschaft*.

Der zugehörige Algorithmus startet mit einer (z. B. durch das Gauß-Verfahren berechneten)
Näherung $R \approx A^{-1}$ sowie einem Anfangsintervall $X^{(0)}$.

Ergibt sich bei Durchführung der Iterationsvorschrift die Beziehung $X^{(k+1)} \subseteq \mathring{X}^{(k)}$,
was die Maschine anhand der Intervallgrenzen leicht prüfen kann,
dann sind A und R regulär,
es gibt genau ein $x \in \mathbb{R}$ mit $A \cdot x = b$
und es gilt $x \in X^{(k+1)}$.

Einschluss des Fehlers

Genauere Resultate lassen sich erzielen, wenn statt der Lösung x
der Fehler $(x - \tilde{x})$ einer Näherungslösung $\tilde{x} \approx x$ eingeschlossen wird.

Wegen $A \cdot (x - \tilde{x}) = b - A \cdot \tilde{x}$ erhalten wir das folgende Verfahren:

Ausgehend von $R \approx A^{-1}$,
einer Näherungslösung $\tilde{x} \approx x$
sowie einem initialen Fehlerintervall $E^{(0)}$
wird folgende Iterationsvorschrift angewandt:

$$E^{(k+1)} = R \diamond \Diamond(b - A \cdot \tilde{x}) \boxplus \Diamond(I - R \cdot A) \diamond E^{(k)}$$

Gilt $E^{(k+1)} \subseteq \mathring{E}^{(k)}$,
dann sind A und R regulär,
es gibt genau ein $x \in \mathbb{R}$ mit $A \cdot x = b$
und es gilt $x \in \tilde{x} \boxplus E^{(k+1)}$.

Auch bei bereits erfüllter Retraktionseigenschaft kann es sinnvoll sein, die Iteration fortzusetzen,
um das Einschließungsintervall weiter zu reduzieren.

ε-Aufblähung

Gilt — wegen der Rundung nach außen — statt der Retraktionseigenschaft nur die Inklusion, oder konvergieren die berechneten Fehlerintervalle von außen gegen den tatsächlichen Fehler, so kann $E^{(k)}$ vor dem Iterationsschritt etwas *aufgebläht* werden:

Sei $X = [x_1, x_2] \in \mathrm{IS}, \varepsilon \in S, \varepsilon > 0$ sowie η die kleinste positive Maschinenzahl.

Das um ε aufgeblähte Intervall X_ε wird definiert durch

$$X_\varepsilon = \begin{cases} X \diamondsuit \varepsilon \diamondsuit [x_1 \bigtriangledown x_2, x_2 \bigtriangleup x_1] & \text{für} \quad x_1 \neq x_2 \\ X \diamondsuit [-\eta, \eta] & \text{für} \quad x_1 = x_2 \end{cases}$$

Ausdrücke als lineare Intervallgleichungssysteme

Die Auswertung arithmetischer Ausdrücke in klassischer Arithmetik ist unzuverlässig.

Beispielsweise liefert MS-Excel 7.0 für den Ausdruck $9 \times x^4 - y^4 + 2 \times y^2$ mit den Argumenten $x = 40545$ und $y = 70226$ das Ergebnis 1160, statt des korrekten Results 1.

Naive Auswertung arithmetischer Ausdrücke mittels Intervallarithmetik ergibt zwar gesicherte Einschlüsse, jedoch häufig mit schlechten Fehlerschranken.

Genauer ist die iterative Lösung eines geeigneten linearen Gleichungssystems.

Beispiel: Der Ausdruck $y = (a + b) \times c - d/e$ wird überführt in das Gleichungssystem

$$\begin{aligned} x_1 &= a \\ x_2 &= x_1 + b \\ x_3 &= c \times x_2 \\ x_4 &= d \\ e \times x_5 &= x_4 \\ x_6 &= x_3 - x_5 \end{aligned}$$

Jede Einschlusslösung X_6 für x_6 ist dann eine sichere Abschätzung für y.

Mathematisches Beweisen mittels Intervallarithmetik

Sei $f(x)$ eine reelle Funktion, X ein reelles Intervall und gelte $\{f(x) \mid x \in X\} \subseteq F(X)$.

Ein solches $F(X)$ kann z. B. für Polynome oder für Funktionen, die durch Taylor-Reihen etc. approximiert werden, mittels der eben beschriebenen Technik recht genau berechnet werden.

Gilt nun beispielsweise $0 \notin F([x_1, x_2])$,
so ist bewiesen,
dass $f(x)$ keine Nullstelle zwischen x_1 und x_2 besitzt.

Mit klassischer Arithmetik lässt sich ein derartiger Beweis prinzipiell niemals zweifelsfrei führen, da der Argumentbereich dort immer nur punktweise betrachtet wird.

Mit ähnlichen Techniken lassen sich alle Extrema reeller Funktionen
auf vorgegebenen Intervallen berechnen.

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Addierschaltungen

Addierer für $\mathrm{UInt}_2(l)$

Berechnung von $C = A + B$ meist in $\mathrm{UInt}_2(l+1)$: $(C_{l-1}, \ldots, C_0) = (A + B) \bmod 2^l$

Höchstwertiges Resultatbit erlaubt Überlauferkennung: $C_l = 1 \Leftrightarrow A + B \geq 2^l \Leftrightarrow$ Überlauf.

Bei regulärem Aufbau (nicht hardware-minimal) fällt C_l ohnehin an.

Optionen für die Überlaufbehandlung:

- C_l wird nicht berechnet (irregulärer Hardwareaufbau): zirkuläre Arithmetik.
- C_l wird berechnet, aber ignoriert (regulärer Hardwareaufbau): zirkuläre Arithmetik.
- Wert von C_l wird in ein Flag abgespeichert: Kann für Überlaufarithmetik, Sättigungsarithmetik oder Überlauferkennung in zirkulärer Arithmetik benutzt werden.
- Zustandsänderung in C_l löst Unterbrechung aus: Kann für Überlaufarithmetik, Sättigungsarithmetik oder Überlauferkennung in zirkulärer Arithmetik benutzt werden.

Berechnung der Summe in all diesen Fällen zunächst gleich!

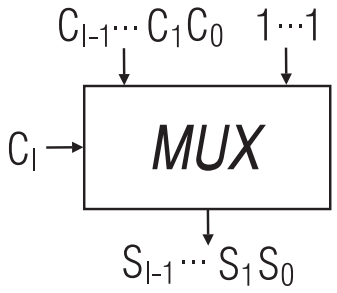
- In Sättigungsarithmetik bewirkt $C_l = 1$ zusätzlich das Setzen von C auf den Wert $1 \ldots 1$.

Beispieladditionen in $\mathrm{UInt}_2(4)$

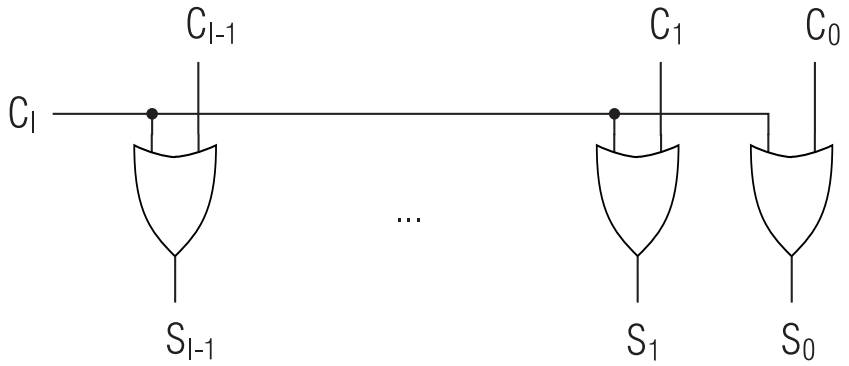
	A	$(1011)_2$	$=$	$(11)_{10}$	$(1011)_2$	$=$	$(11)_{10}$
	B	$(0011)_2$	$=$	$(3)_{10}$	$(0111)_2$	$=$	$(7)_{10}$
exakte Arithmetik	$A + B$	$(01110)_2$	$=$	$(14)_{10}$	$(10010)_2$	$=$	$(18)_{10}$
Überlaufarithmetik	$A \oplus_{16} B$	$(1110)_2$	$=$	$(14)_{10}$	undefiniert		
zirkuläre Arithmetik		$(1110)_2$	$=$	$(14)_{10}$	$(0010)_2$	$=$	$(2)_{10}$
Sättigungsarithmetik		$(1110)_2$	$=$	$(14)_{10}$	$(1111)_2$	$=$	$(15)_{10}$

Sättigung der Addition in $\mathrm{UInt}_2(l)$

Multiplexer (Schaltung)



Oder-Verknüpfung (Schaltung)



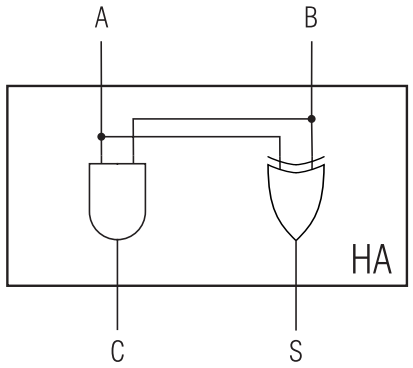
Bedingter Sprung (Maschinensprache)

Folge arithmetisch/logischer Operationen (Maschinensprache)

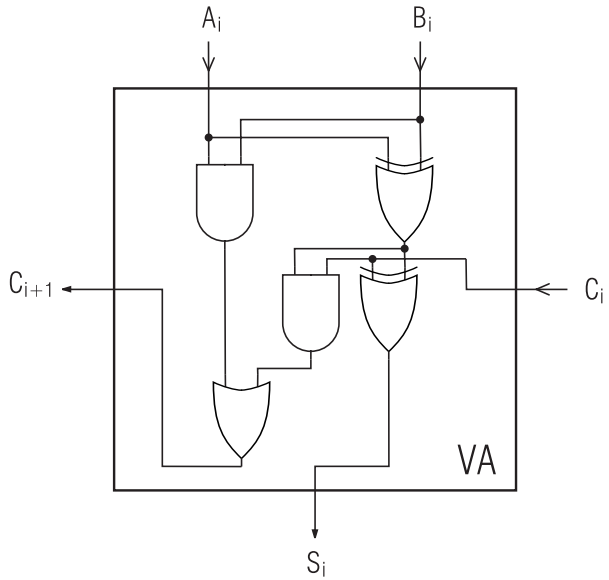
Bedingte Anweisung (höhere Programmiersprache)

Grundbausteine: Halbaddierer (HA) und Volladdierer (VA)

$A + B = 2 \times C + S$			
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

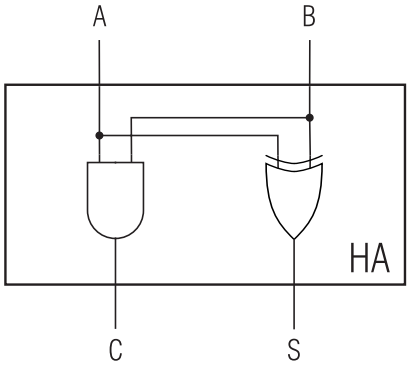


$A_i + B_i + C_i = 2 \times C_{i+1} + S_i$				
A_i	B_i	C_i	C_{i+1}	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



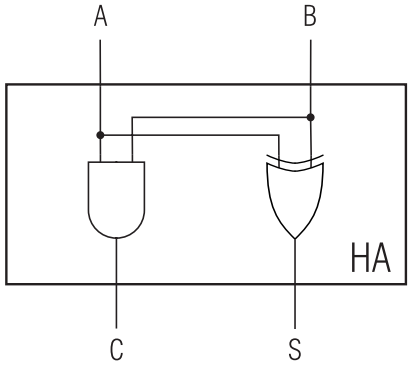
Halbaddierer: Eigenschaften

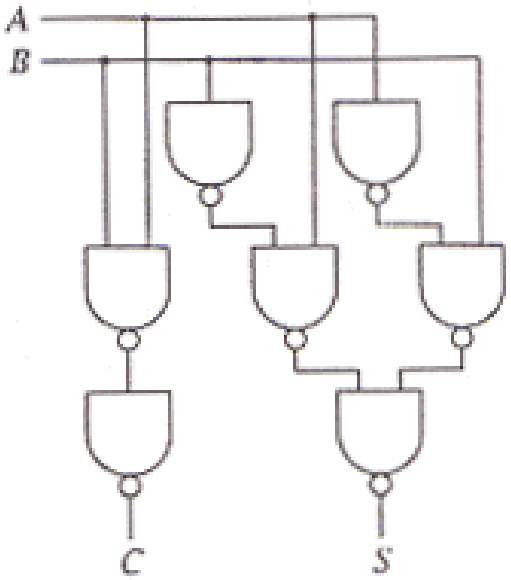
$A + B = 2 \times C + S$			
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

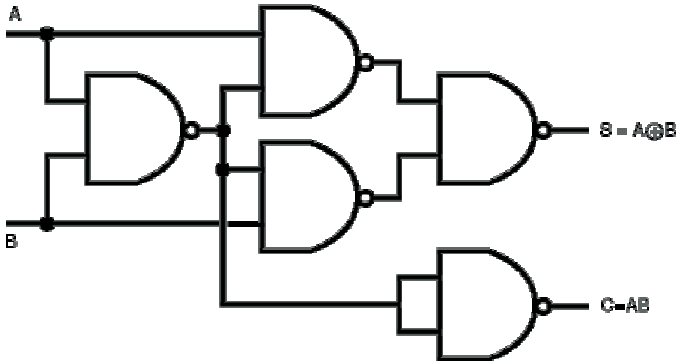


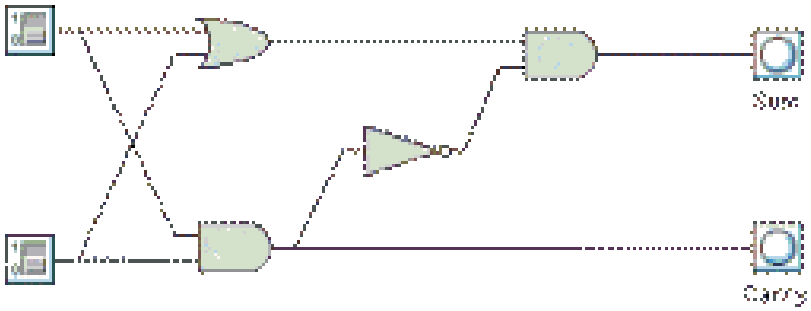
- addiert zwei Eingänge derselben Bitposition (single bit binary adder)
- liefert 2-Bit-Summe, $\mathrm{UInt}_2(1) \times \mathrm{UInt}_2(1) \rightarrow \mathrm{UInt}_2(2)$
- Ergebnis interpretierbar als Carry-Save-Darstellung der Summe
- Ausgang S alleine entspricht $A \oplus_2 B$ in zirkulärer Arithmetik
- Wertetabelle durch algebraische Gleichung eindeutig bestimmt
- verschiedene (aber äquivalente) logische Gleichungen aus der Wertetabelle ableitbar

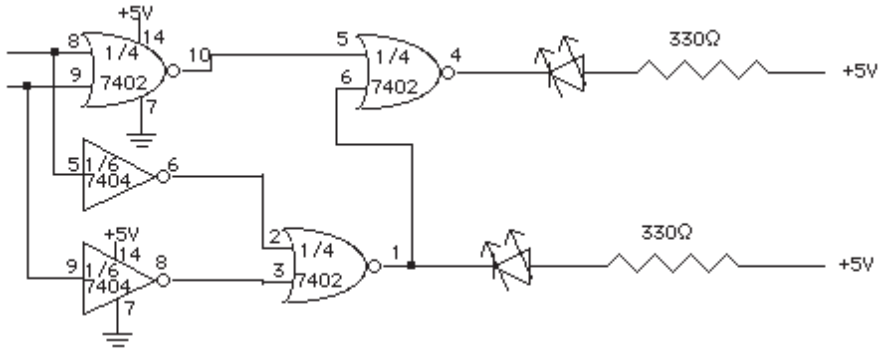
Halbaddierer: Implementierungsvarianten





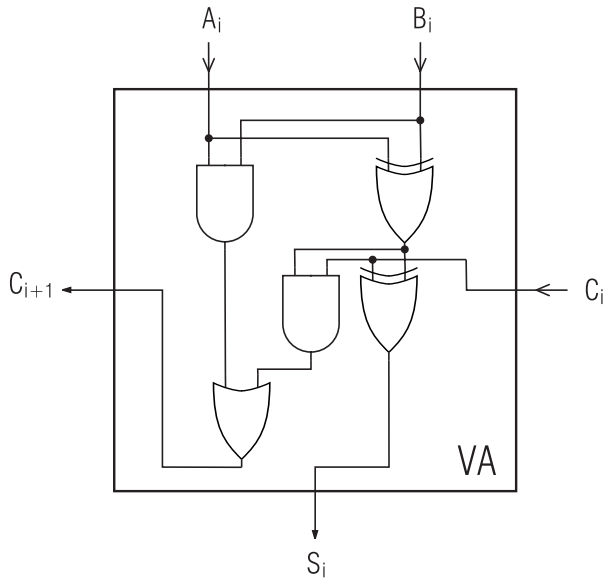






Volladdierer: Eigenschaften

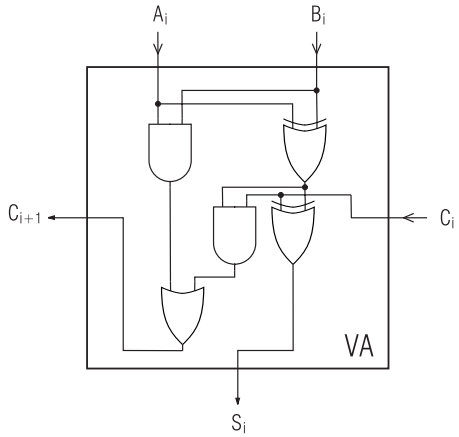
$A_i + B_i + C_i = 2 \times C_{i+1} + S_i$				
A_i	B_i	C_i	C_{i+1}	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



- addiert drei Eingänge derselben Bitposition (3-operand single bit binary adder)
- liefert 2-Bit-Summe, $\mathrm{UInt}_2(1) \times \mathrm{UInt}_2(1) \times \mathrm{UInt}_2(1) \rightarrow \mathrm{UInt}_2(2)$
- Ergebnis interpretierbar als Carry-Save-Darstellung der Summe
- Wertetabelle durch algebraische Gleichung eindeutig bestimmt
- verschiedene (aber äquivalente) logische Gleichungen aus der Wertetabelle ableitbar

Zeitliche Modellierung eines Volladdierers

$$
\begin{aligned}
C_{i+1}(t) &= A_i(t-2)\,B_i(t-2) + (A_i(t-4) \oplus B_i(t-4))\,C_i(t-2) \\
S_i(t) &= A_i(t-4) \oplus B_i(t-4) \oplus C_i(t-2)
\end{aligned}
$$



Dabei wird t in Einheiten von t_g (eine Gattergrundschaltzeit) gemessen und $t_{and} = t_{or} = t_g$ sowie $t_{xor} = 2 \times t_g$ angenommen.

Wir setzen $t_{VA} = t_{and} + t_{or}$ zur Beschreibung der Verzögerung auf dem Übertragspfad.

Zuordnung der Operanden zu den Eingängen A_i, B_i, C_i nach boolescher Logik beliebig.

Minimierung des kritischen Pfads durch Anlegen des zuletzt stabilen Signals an C_i.

Hilfsgrößen der Übertragserzeugung

Für die Codierung $0 \mathrel{\widehat{=}} \textit{false}$, $1 \mathrel{\widehat{=}} \textit{true}$ erweisen sich folgende Definitionen als nützlich:

G_i	=	$A_i\, B_i$	*generieren* (definitiv Ausgangsübertrag)
P_i	=	$A_i \oplus B_i$	*propagieren* (Eingangsübertrag wird durchgeleitet)
T_i	=	$A_i + B_i$	*transferieren* ($= G_i + P_i$)
L_i	=	$\overline{A_i}\ \overline{B_i}$	*löschen* (definitiv kein Ausgangsübertrag)

Genau eines der Prädikate G_i, P_i oder L_i besitzt den Wert *true*.
Genau eines der Prädikate T_i oder L_i besitzt den Wert *true*.

Wichtig: G_i, P_i, T_i, L_i direkt und simultan aus A_i und B_i, also ohne Kenntnis von C_i, berechenbar.

Implementierungsoptionen für Volladdierer

$$
\begin{aligned}
S_i &= A_i\,B_i\,C_i + A_i\,\overline{B_i}\,\overline{C_i} + \overline{A_i}\,B_i\,\overline{C_i} + \overline{A_i}\,\overline{B_i}\,C_i \\
&= A_i \oplus B_i \oplus C_i \qquad \text{(3-Bit-XOR)} \\
&= A_i \oplus (B_i \oplus C_i) \\
&= (A_i \oplus B_i) \oplus C_i \quad = \quad P_i \oplus C_i
\end{aligned}
$$

Formel $P_i \oplus C_i$ akzentuiert die *Assimilation* der *Partialsumme* mit dem Eingangsübertrag.

$$
\begin{aligned}
C_{i+1} &= A_i\,B_i + A_i\,C_i + B_i\,C_i \qquad \text{(3-Bit-Majoritätsfunktion)} \\
&= A_i\,B_i + (A_i + B_i)\,C_i \quad = \quad G_i + T_i\,C_i \\
&= A_i\,B_i + (A_i \oplus B_i)\,C_i \quad = \quad G_i + P_i\,C_i
\end{aligned}
$$

C_0 wird bei der Addition formal identisch Null gesetzt.

Formeln $S_i = P_i \oplus C_i$ und $C_{i+1} = G_i + P_i\,C_i$ zusammen ermöglichen gemeinsame Verwendung des Signals P_i.

Übertragserzeugung aus den Hilfsgrößen

$$C_{i+1} = G_i + P_i\, C_i$$ Ripple-Carry-Addierer

$$C_{i+1} = G_i + T_i\, C_i$$ Carry-Skip-Addierer

$$\hat{C}_{km} = C_{km} + \bigwedge_{i=km-m}^{km-1} T_i\, \hat{C}_{km-m}$$

$$C_{i+1} = \overline{L_i}\,(G_i + P_i\, C_i)$$ Manchester-Addierer

$$C_{i+1} = G_i + P_i\, C_i$$ Carry-Completion-Addierer

$$\overline{C_{i+1}} = L_i + P_i\, \overline{C_i}$$

Standardentwurf von CMOS-Schaltungen

Zu berechnen sei eine boolesche Funktion $y = f(x_1, \ldots, x_n)$.

Gesuchte CMOS-Schaltung besitzt NMOS-Netzwerk (N-Netz) und PMOS-Netzwerk (P-Netz).

Das N-Netz kann den Ausgang y mit Masse verbinden.
Die Transistoren des N-Netzes leiten, wenn am Eingang die Versorgungsspannung anliegt.
Das N-Netz implementiert daher eine Funktion $\overline{y} = g(x_1, \ldots, x_n)$.
Damit hat $g(x_1, \ldots, x_n) = \overline{f(x_1, \ldots, x_n)}$ zu gelten.

Das P-Netz kann den Ausgang y mit der Versorgungsspannung verbinden.
Die Transistoren leiten, wenn am Eingang Masse anliegt.
Das P-Netz implementiert daher eine Funktion $y = h(\overline{x}_1, \ldots, \overline{x}_n)$.
Damit hat $h(x_1, \ldots, x_n) = f(\overline{x}_1, \ldots, \overline{x}_n)$ zu gelten.

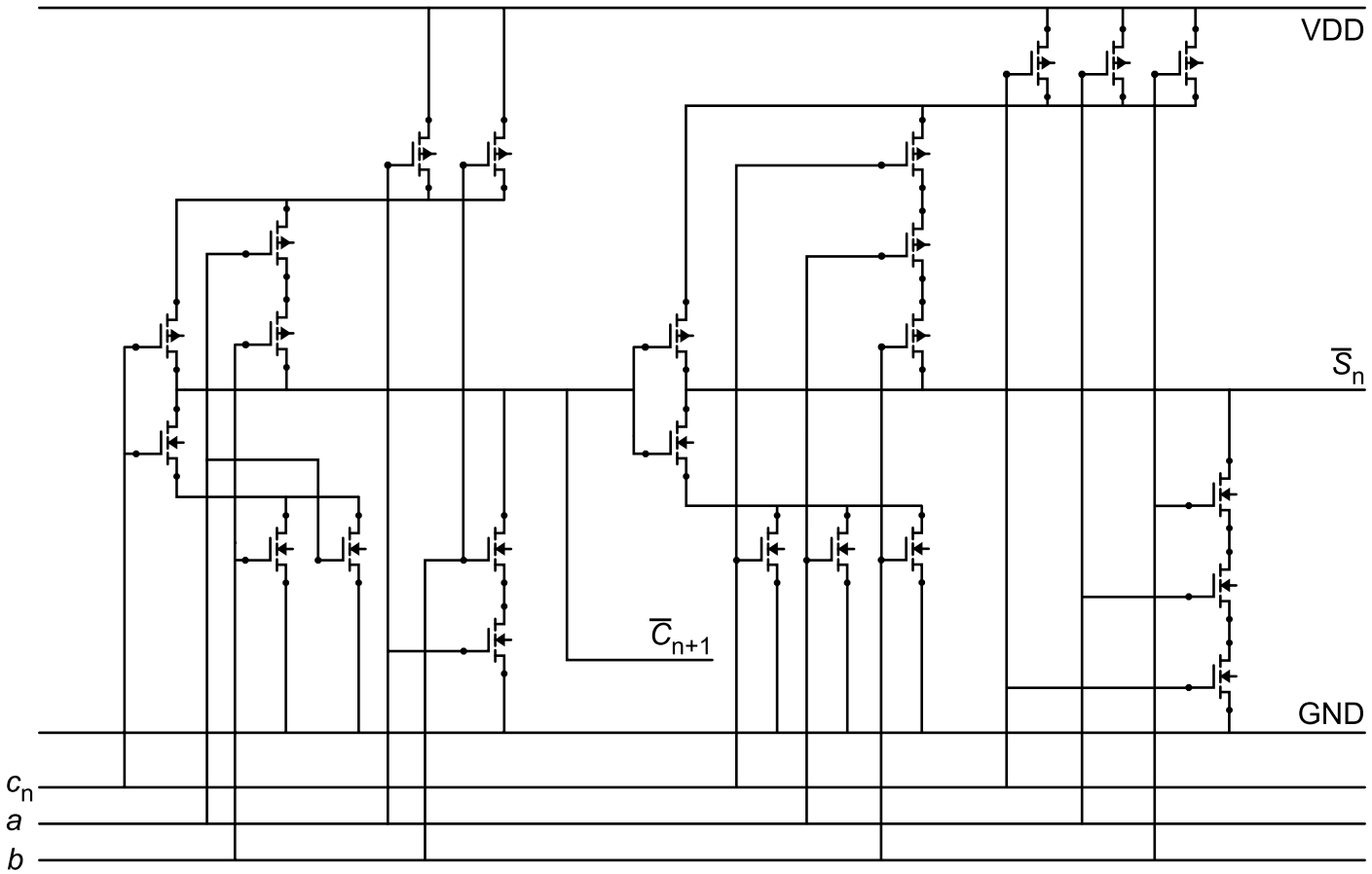
Ein N-Netz für $g = g_1 \wedge g_2$ wird als Reihenschaltung der Subnetze für g_1 und g_2 realisiert,
für $g = g_1 \vee g_2$ als Parallelschaltung.
Ebenso für ein P-Netz mit boolescher Form h.

Ein nur aus einer Variablen x_i bestehendes N-Netz wird durch einen NMOS-Transistor mit Eingang x_i realisiert, ein nur aus einer negierten Variablen $\overline{x}_i$ bestehendes P-Netz durch einen PMOS-Transistor mit Eingang x_i.

Es folgt, dass die booleschen Formen g und h zueinander dual sind.

Beispiel: $f(A, B, C) = \overline{(A \vee B) \wedge C} \Rightarrow g(A, B, C) = (A \vee B) \wedge C,\ h(A, B, C) = (A \wedge B) \vee C$

Volladdierer in CMOS-Standardaufbau



(aus N. Reifschneider: CAE-gestützte IC-Entwurfsmethoden, 1998, p. 125)

Kaskadierter Aufbau.

Nur 24 Transistoren.

Ergebnisse S_i und C_{i+1} fallen invertiert an.

Wenn nicht negierte Signale gewünscht werden,
sind zwei Inverter (vier Transistoren) mehr nötig.

Kritischer Pfad für Carry-zu-Carry geht durch zwei Gatter.

Kritischer Pfad für Carry-zu-Sum geht durch drei Gatter.

$$C_{i+1} = A_i\,B_i + (A_i + B_i)\,C_i \qquad S_i = A_i\,B_i\,C_i + (A_i + B_i + C_i)\,\overline{C_{i+1}}$$

Selbstdualität von XOR und Majoritätsfunktion

Die Volladdiererfunktionen S_i (3-Bit-XOR) und C_{i+1} (3-Bit-Majoritätsfunktion) sind selbstdual:

$$S_i(\overline{A_i}, \overline{B_i}, \overline{C_i}) = \overline{S_i(A_i, B_i, C_i)}$$

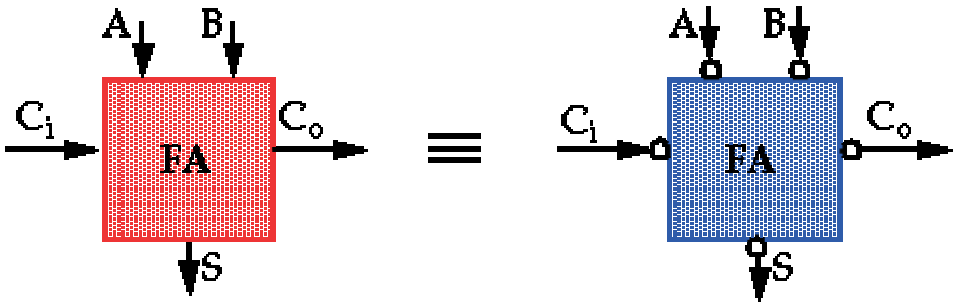
$$C_{i+1}(\overline{A_i}, \overline{B_i}, \overline{C_i}) = \overline{C_{i+1}(A_i, B_i, C_i)}$$

Eine Überprüfung ist leicht an Hand der Wertetabelle des Volladdierers möglich.

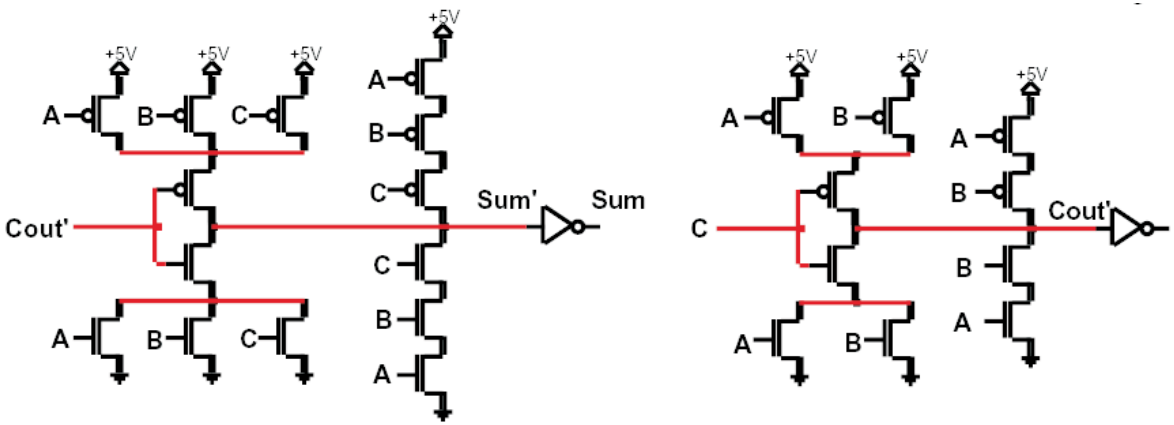
Selbstdualität ist eine seltene Eigenschaft, die zu Optimierungen genutzt werden kann.

Aus $f(\overline{x}) = \overline{f(x)}$ folgt zum Beispiel $g(x) = h(x)$ für den CMOS-Standardentwurf.

Die Selbstdualität impliziert die Inversionseigenschaft des Volladdierers:



Volladdierer als Mirror Adder



Vollständig symmetrischer CMOS-Addierer (mirror adder).

Wie zuvor 28 Transistoren.

N-Netz nicht dual zu P-Netz.

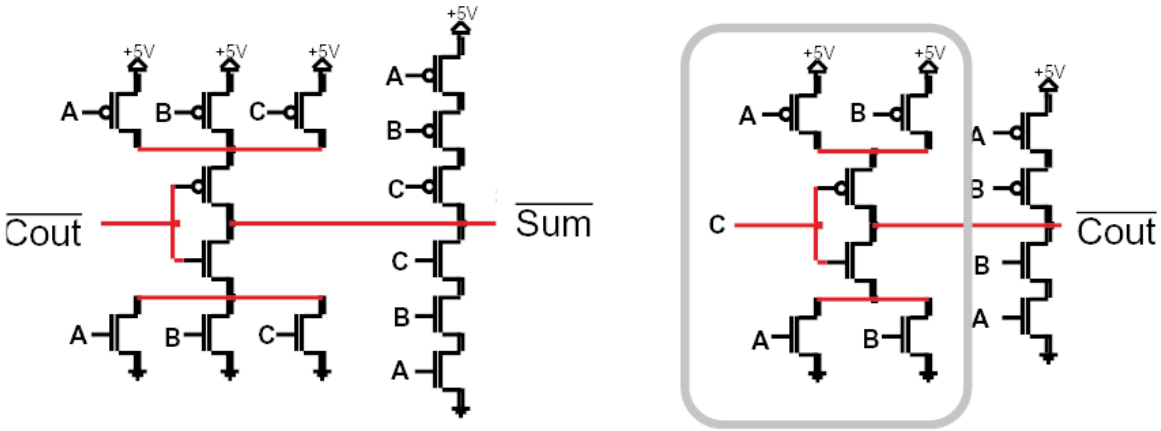
Selbstdualität des Volladdierers ausgenutzt.

Zweck:
Weniger Transistoren in Reihe, Widerstand reduziert; uniformes Layout.

Kritischer Pfad für Carry-zu-Carry geht immer noch durch zwei Gatter.

Kritischer Pfad für Carry-zu-Sum geht immer noch durch drei Gatter.

Transistor-Dimensionierung im Mirror Adder



Beschleunigung durch geeignete Dimensionierung der Transistoren.

Nur Transistoren im markierten Bereich liegen auf kritischem Pfad.

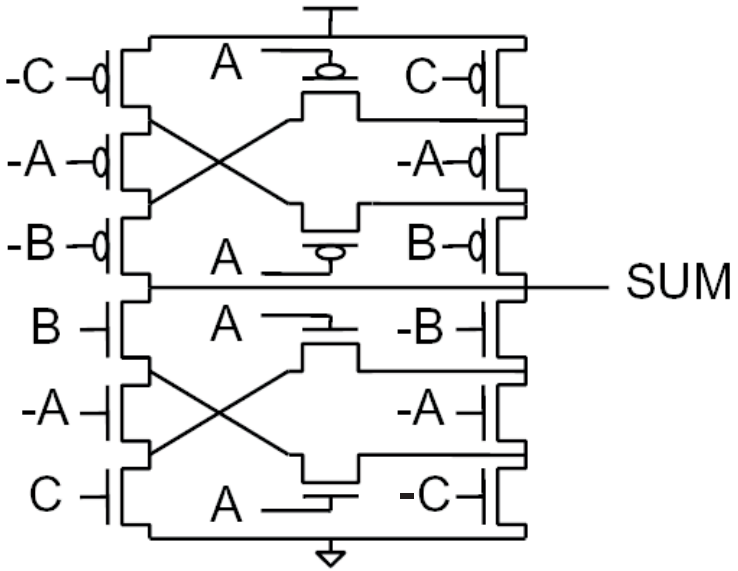
Alle anderen Transistoren sollten klein sein, um getriebene Last zu reduzieren.

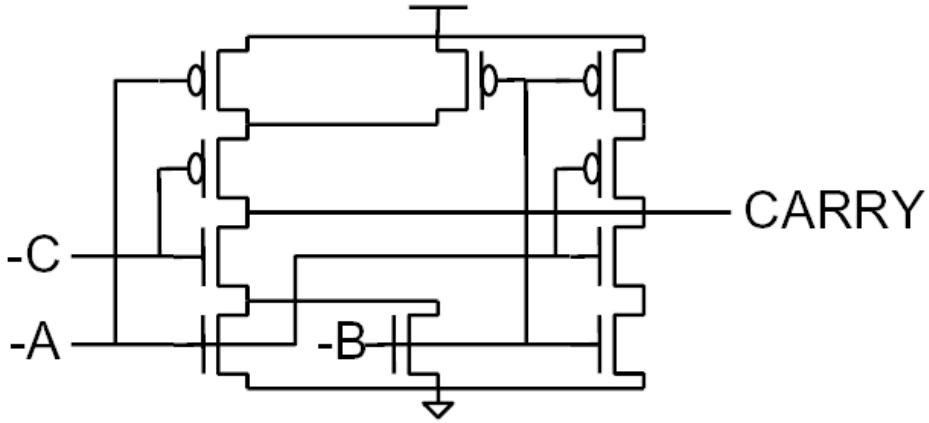
Dimensionierungsbeispiel in N. Weste, K. Eshraghian: Principles of CMOS VLSI design.

Gilt als langsam, auch bei sorgfältiger Optimierung.

Nicht kompositionale Implementierung des Volladdierers

Für seriellen Addierer, Carry-Save-Addierer, etc. ist es sinnvoll,
annähernd gleiche Latenzen für Summe und Carry zu haben.

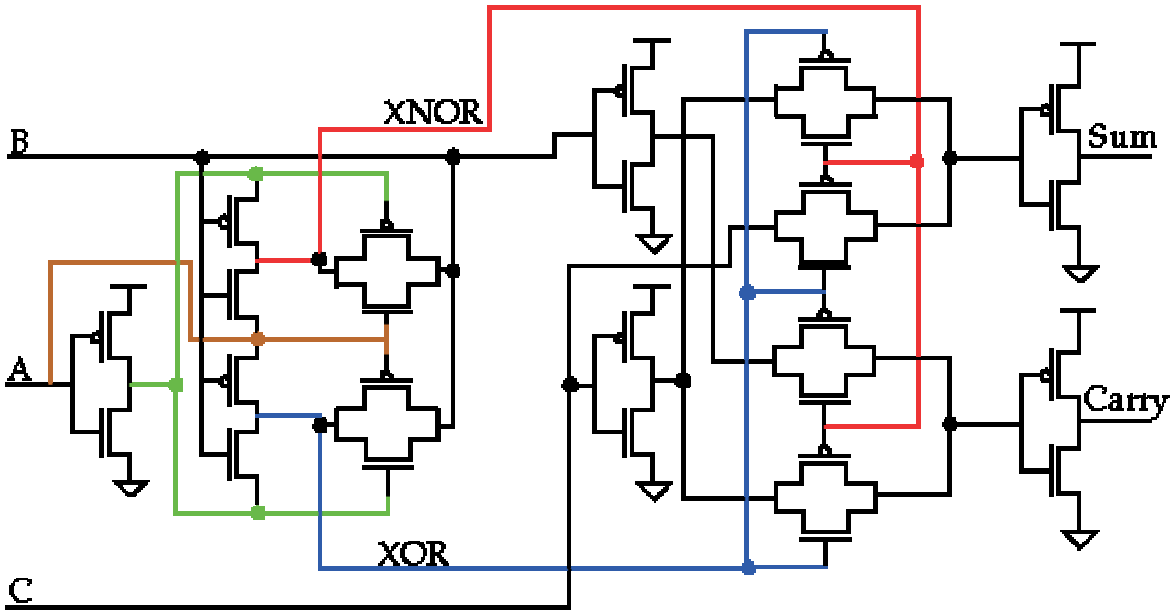




26 Transistoren (plus 6 Transistoren der Inverter für A, B, C).
Kritischer Pfad für Carry-zu-Carry und Carry-zu-Sum geht durch zwei Gatter.

Volladdierer mit Transmissionsgattern

Anzahl der Transistoren kann durch Transmissionsgatter verringert werden.



26 Transistoren.

Mit Inverter für XNOR nur 24 Transistoren, aber langsamer.

Variante mit 18 Pass-Transistoren bekannt!

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Ripple-Carry- und Carry-Skip-Addierer

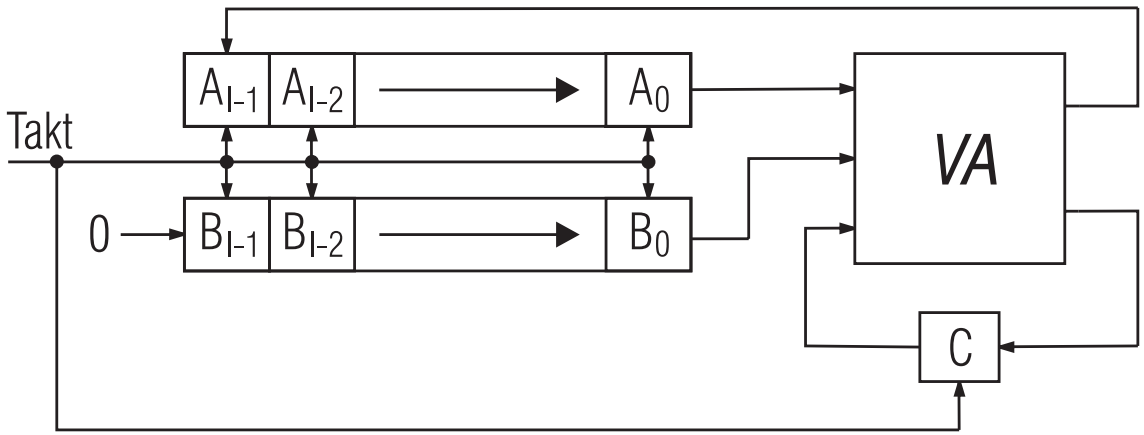
Serieller Addierer

Serielle Addition (synchron): l-Bit-Addition wird realisiert durch l sukzessive 1-Bit-Additionen.

Pro Takt wird ein Paar von Operandenbits verarbeitet und ein Summenbit berechnet.

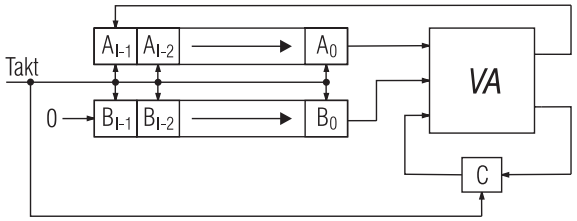
Übertrag aus vorherigem Schritt verknüpft Einzeladditionen (3-Operanden/1-Bit-Addition).

LSDF-Stil (least significant digit first) beginnt mit dem niederwertigsten Bit (Position 0).



Zur Optimierung von Latenz oder Durchsatz ist der Volladdierer so zu entwerfen,
dass die Verzögerung beider Ausgänge bezüglich aller drei Eingänge minimal wird!

Eigenschaften des seriellen Addierers



Charakteristika: Langsamster, aber meist auch billigster Addierer.

Latenz: l Taktzyklen

Durchsatz: $1/l$

Aufwand:

grundsätzlich: Volladdierer, 1-Bit-Latch

fallweise: Zähllogik, $\lceil \log_2 l \rceil$-Bit-Zählregister, bis zu drei l-Bit-Schieberegister

Gründe für den Einsatz serieller Addierer:

- Minimale Kosten oder Chipfläche sind manchmal wichtiger als Geschwindigkeit.
- Extrem kleiner Hardware-Aufwand, insbesondere für lange Zahlen.
- Kleine Modulabmessungen, geringe Modulkomplexität.
- Minimaler Routing-Aufwand: 1-Bit-Verbindungen statt l-Bit-Verbindungen.
- Bei Nutzung massiver Parallelität kostengünstiger als andere Addierer.

Bit-serielle Arbeitsweise, Bit-Level-Pipelining

Statt der Entnahme der Daten aus Schieberegistern und der Abspeicherung der Ergebnisse in Schieberegister ist auch Bit-Level-Pipelining möglich:

Ausgang des bit-seriellen Operators (hier: serieller Addierer) über Latch mit Eingang eines weiteren bit-seriellen Operators (Addition, Subtraktion, Multiplikation, Vergleich) verbunden.

In der digitalen Signalverarbeitung liefern die Quellen oft bereits serielle Datenströme.

Bit-Level-Pipelining ermöglicht sehr kleine Zykluszeiten und damit einen hohen Durchsatz.

Bei datenabhängigen Operationen minimiert Bit-Level-Pipelining die effektive Latenz.

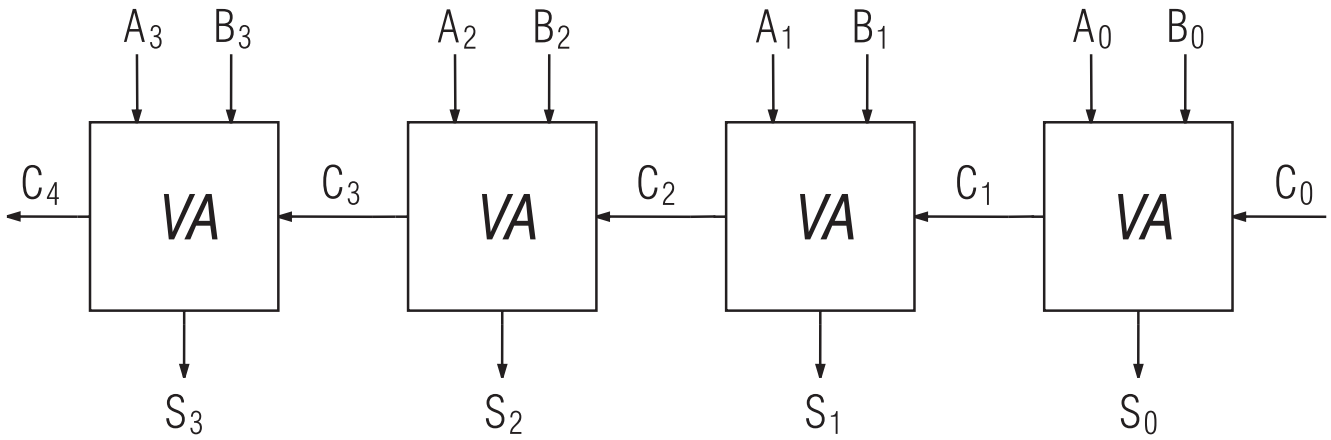
Bit-Level-Pipelining ist außer im LSDF-Stil auch im MSDF-Stil (most significant digit first) möglich.

Dabei wird stets mit dem höchstwertigsten Bit (Position $l - 1$) begonnen.

Der MSDF-Stil erfordert für Addition, Subtraktion und Multiplikation zwar redundante Zahlendarstellungen, erlaubt dafür aber auch die bit-serielle Berechnung der Division und der wichtigsten transzendenten Funktionen.

Ripple-Carry-Addierer (RCA)

Schaltnetz aus l Volladdierern, wird auch *Paralleladdierer* genannt.



Ripple-Carry: Überträge $C_1, C_2, \ldots, C_l$ werden sukzessive auseinander gebildet, selbst wenn alle Daten gleichzeitig anliegen (engl. *to ripple*, plätschern).

Volladdierer für Bitposition 0 könnte durch Halbaddierer ersetzt werden, da $C_0 = 0$.

Vorliegende Variante erlaubt aber, alternativ $A + B + 1$ zu berechnen.

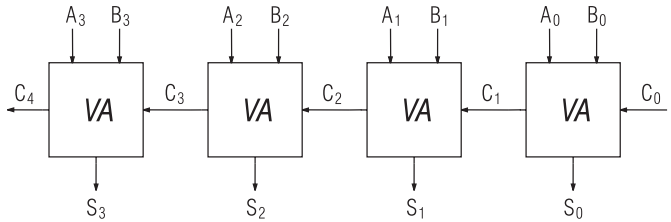
Außerdem ermöglicht regulärer Aufbau eine Realisierung in Bit-Slice-Technik.

Zustandsübergänge der Summenbits im Ripple-Carry-Addierer

$t = 0$			1	1	1	1		1	0	1	0
		+	0	0	0	1	+	0	1	0	1
$t = t_{VA}$	Summe		0	0	0	0		0	0	0	0
	Übertrag	0	0	0	1	0	0	0	0	0	0
$t = 2 \times t_{VA}$	Summe		1	1	1	0		1	1	1	1
	Übertrag	0	0	1	1	0	0	0	0	0	0
$t = 3 \times t_{VA}$	Summe		1	1	0	0					
	Übertrag	0	1	1	1	0					
$t = 4 \times t_{VA}$	Summe		1	0	0	0					
	Übertrag	1	1	1	1	0					
$t = 5 \times t_{VA}$	Summe		0	0	0	0					
	Übertrag	1	1	1	1	0					

Die Stabilisierungszeit ist also abhängig von den eingegebenen Daten.

Eigenschaften des Ripple-Carry-Addierers



Aufwand: l Volladdierer

Maximale Latenz: $\approx l \times t_{VA}$

(Wenn die Berechnung von C_1 aus A_0, B_0 und C_0 oder die Berechnung von S_{l-1} aus A_{l-1}, B_{l-1} und C_{l-1} länger als t_{VA} dauert, kann noch ein kleiner additiver Anteil hinzukommen.

Bei Durchführung der Addition ohne C_0 kann die Latenz auch geringfügig kleiner ausfallen.)

Konsequenz: Optimierung des Carry-Gatters bringt mehr als Optimierung des Summengatters.

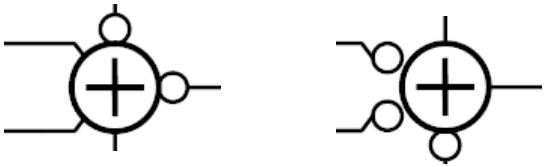
Durchschnitt über längste Ripple-Distanzen: $\approx \log_2 l$

(Entscheidend ist längste 1-Carry-Chain $X_{i+h} P_{i+h-1} P_{i+h-2} \ldots P_{i+1} G_i$.)

Geschwindigkeitsvorteil gegenüber seriellem Addierer:

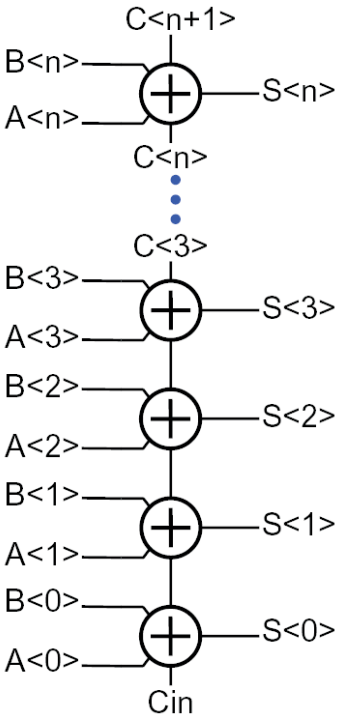
- Synchron:
  Wegfall der Latching-Zeit (aber gleiche Zeitkomplexität $O(l)$).
- Asynchron:
  Im Mittel schneller, da längste 1-Carry-Chain meist kürzer als l; dies kann aber nur genutzt werden, wenn Stabilisierung erkannt wird, z. B. durch *Completion-Logik*.

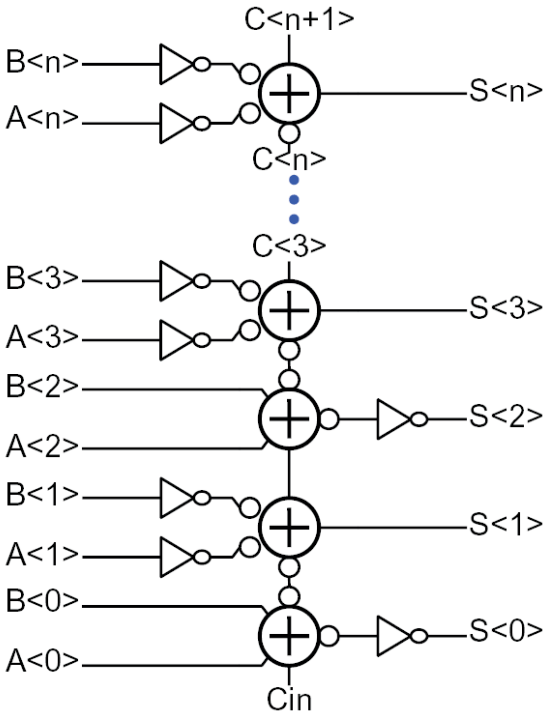
Ripple-Carry-Addierer mit alternierendem Carry-Signal



Inversionseigenschaft des Volladdierers auch bei negierten Ausgängen anwendbar.

Inverter auf dem kritischen Pfad werden dadurch eingespart.





Anzahl durchlaufener Gatter auf kritischem Pfad (praktisch) halbiert. Verzögerung der Ein- und Ausgänge fällt nicht ins Gewicht.

Kapazitive Last des Ausgangs C_{i+1} steigt an.

Serienwiderstand des Carry-Gatters nicht von Gate-Kapazitäten der nächsten Stufe getrennt.

Konsequenz: Gesamtschaltzeit zunächst unklar, evtl. sogar langsamer, da höhere Treiberlast!

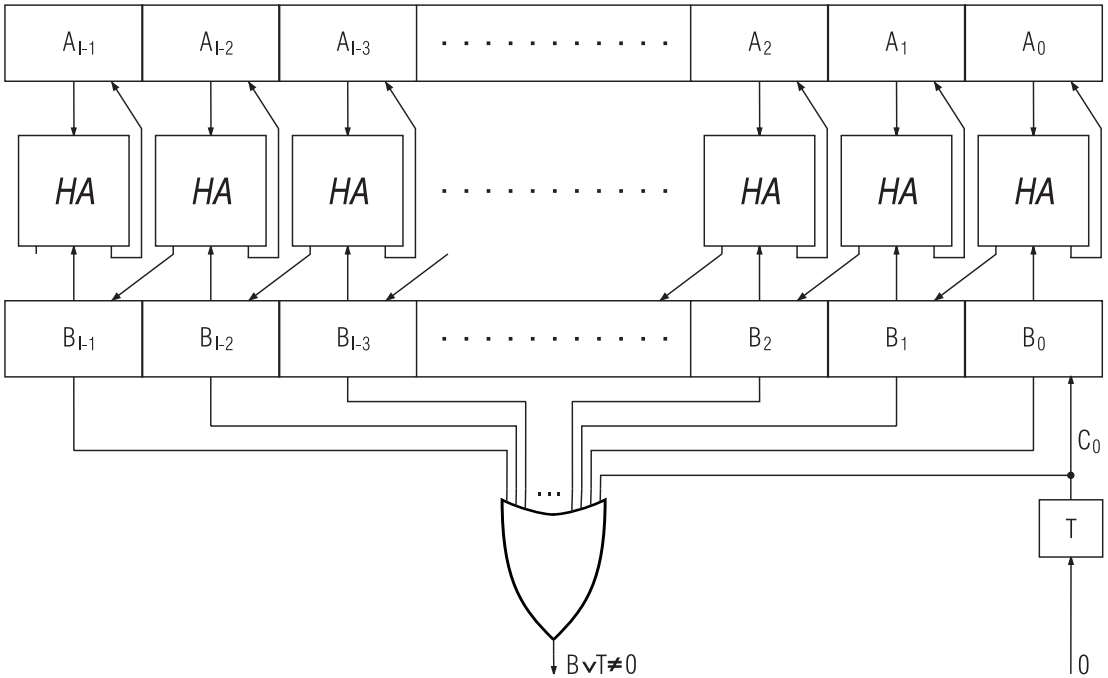
Beschleunigung setzt voraus, dass die beiden Transistoren des folgenden Carry-Gatters die für C_{i+1} dominierende kapazitive Last darstellen.

Von-Neumann-Addierer

Synchroner Paralleladdierer mit Eigenschaften des seriellen und des Ripple-Carry-Addierers.

In jedem Takt überschreiben eine Teilsumme und ein noch hinzuzuaddierender Übertrag die beiden Operanden; ist der Übertragsoperand Null geworden, endet die Berechnung.

Die Anzahl der Takte entspricht der längsten 1-Carry-Chain des Ripple-Carry-Addierers.



Aufwand:

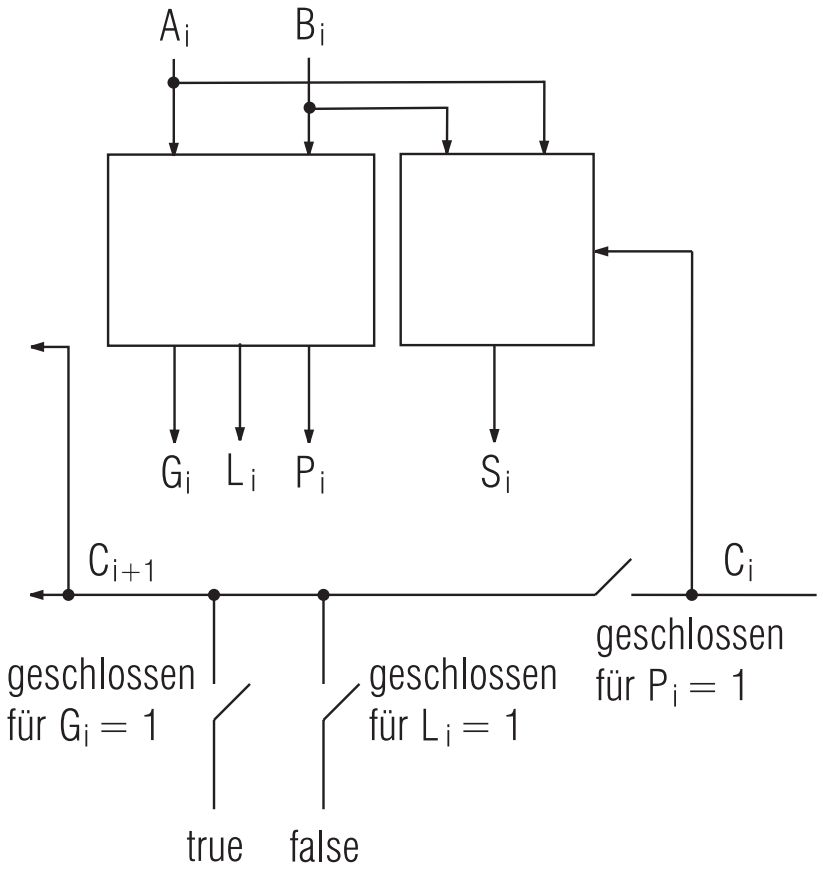
- l Halbaddierer
- T-Flipflop
- Oder-Gatter mit Fan-in $l + 1$
- $2 \times l + 1$ Multiplexer
- 2 Register der Länge l Bit

Manchester-Addierer

Ripple-Carry-Addierer mit beschleunigter Weiterleitung der Überträge.

G_i, P_i und L_i steuern je einen Schalter, der das ausgehende Übertragssignal auf *true* oder *false* zieht bzw. den eingehenden Übertrag auf den ausgehenden Übertrag durchschaltet.

Alle diese Schaltvorgänge finden simultan mit einer kleinen konstanten Verzögerung statt.



Manchester-Technik

Erzielte Verbesserung gegenüber dem Ripple-Carry-Addierer:

Auf dem kritischen Pfad (Übertragserzeugung) können schnelle Schalter
(z. B. Pass-Transistoren) statt mehrstufiger Schaltnetze verwendet werden.

Übertragsberechnung soll pro Bitposition nur mit einer Transistorschaltzeit verzögert werden.

Problem: Tatsächliche Verzögerung steigt quadratisch in der Anzahl kaskadierter Stufen!

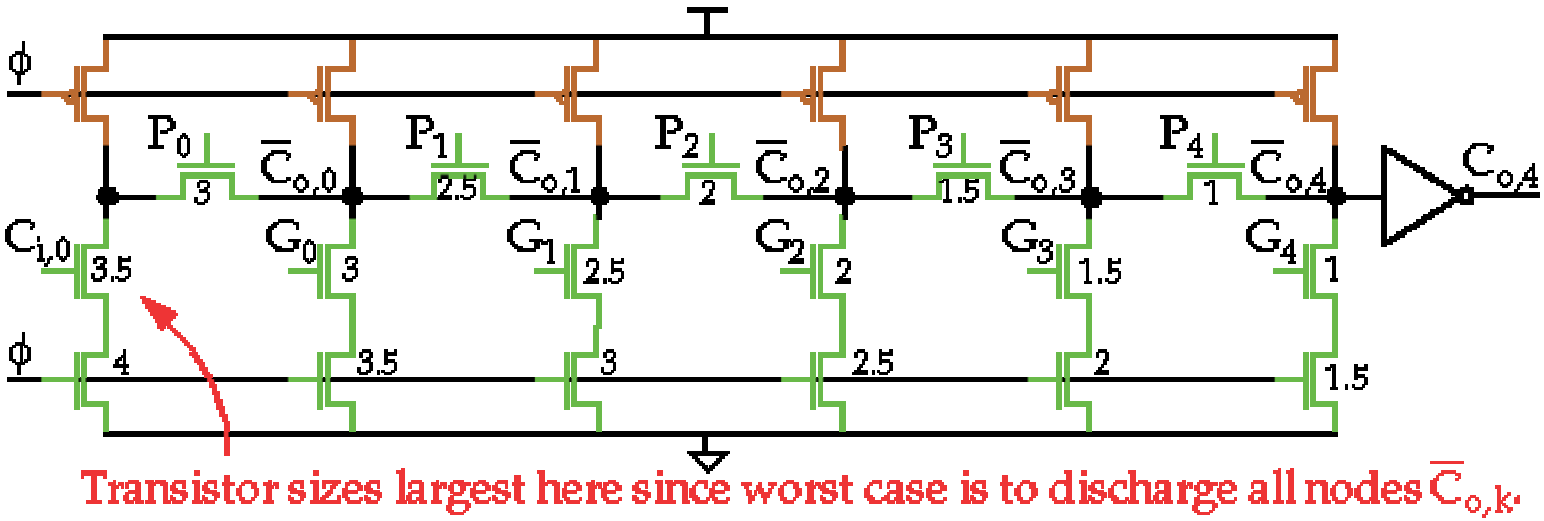
Abhilfe durch geeignete Dimensionierung der Transistoren, Puffer sowie evtl. Bypass-Logik.

Problem: Anzahl direkt kaskadierbarer Schalter in der Praxis beschränkt
(weniger als acht, häufig auch sehr viel kleiner).

Eingeschränkte Nutzung des Prinzips durch Partitionierung der Schaltung in Blöcke,
die durch Puffer elektrisch getrennt werden (optimal: 3–4 kaskadierte Stufen).

Manchester-Technik zur schnellen Propagierung auf dem kritischen Pfad
prinzipiell auch in anderen Schaltungen einsetzbar.

Manchester-Addierer in dynamischer Logik

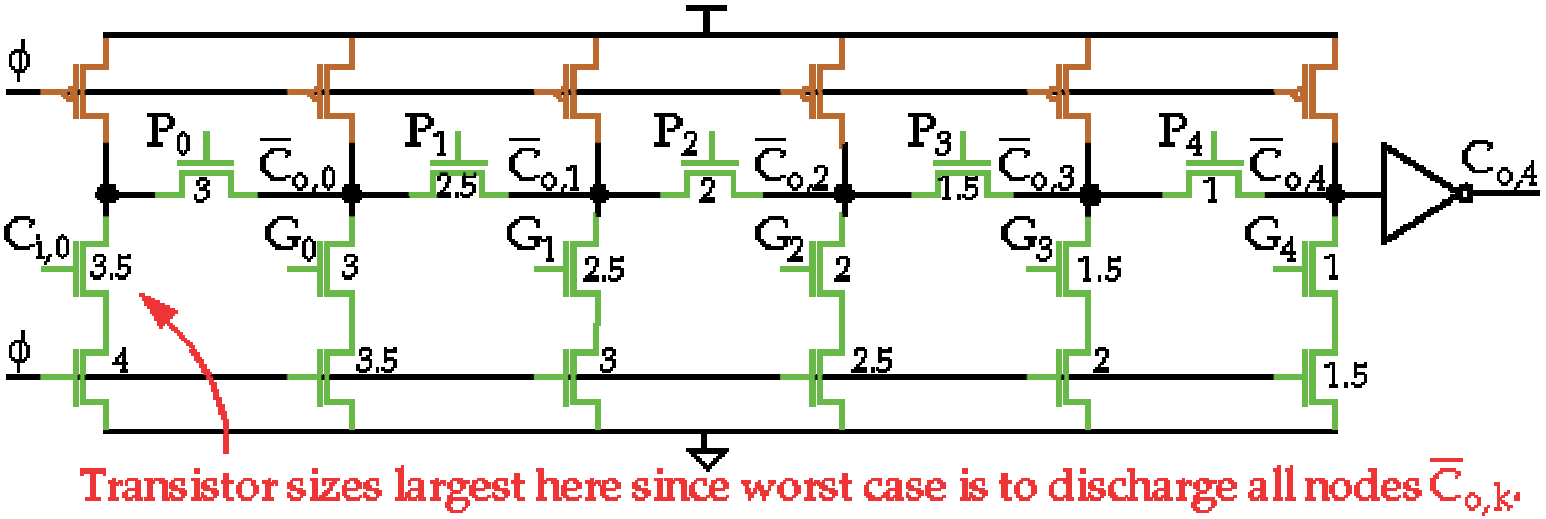


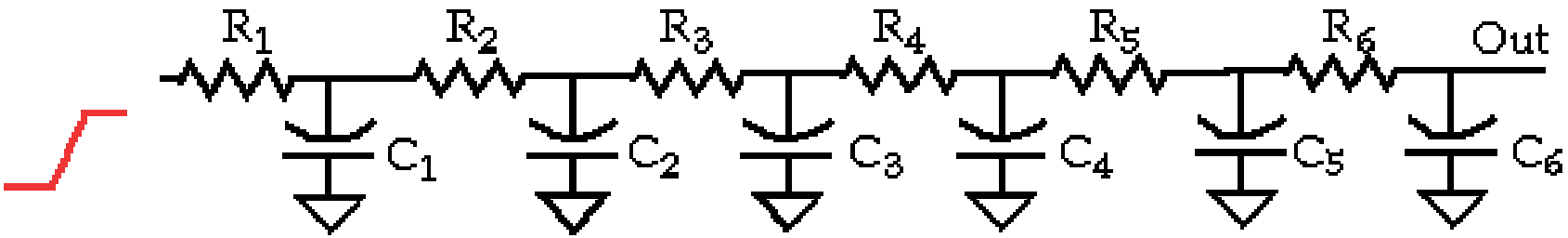
P-Prädikat wird durch Pass-Transistor implementiert.
G-Prädikat wird mittels Pull-down-Transistor realisiert.

L-Prädikat würde in statischer Logik mittels Pull-up-Transistor realisiert. Da P, G und L sich ausschließen, genügt in dynamischer Logik hierfür der normale Precharge-Vorgang.

Das P-Prädikat kann hier nicht durch das T-Prädikat ersetzt werden, weil bei aktivem G-Prädikat die Vorgängerknoten sonst ebenfalls entladen würden!

Verzögerung im Manchester-Addierer





Pro Knoten nur 4 Diffusionskapazitäten, aber Elmore-Delay $\sim \sum_{i=1}^{N}(C_i \sum_{j=1}^{i} R_j)$.

Je kleiner der Index i, desto größer der Einfluss von R_i, daher größer dimensionieren.

Carry-Skip-Addierer

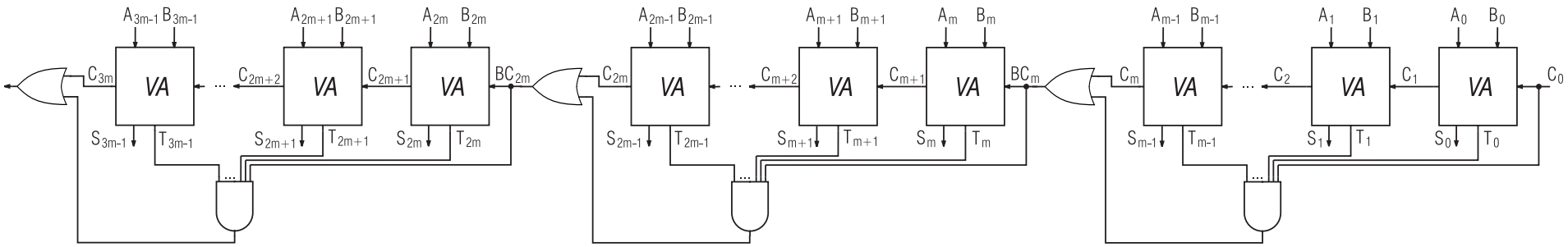
Beim Ripple-Carry-Addierer wird die Latenz der Summenberechnung
durch die sich verzögert stabilisierenden Eingangsüberträge bestimmt.

Die Carry-Skip-Technik (auch Carry-Bypass-Technik genannt)
reduziert die maximale bzw. durchschnittliche Latenz der Übertragsberechnung.

Die Grundstruktur ist ein in Blöcke aufgeteilter RCA.

Jeder Block erhält eine Carry-Skip-Logik,
die aus dem eingehenden Blockübertrag schnell den ausgehenden Blockübertrag berechnet.

Der ausgehende Blockübertrag ist wertgleich mit dem ausgehenden Übertrag des höchstwertigen Volladdierers des Blocks, aber schneller stabil.



Rekurrenzen im Carry-Skip-Addierer

Allgemein: Der Block h besteht aus den Bitpositionen $j_h, j_h + 1, \ldots, j_{h+1} - 1$.

Block-Carry: $BC_{j_h} = C_{j_h}$

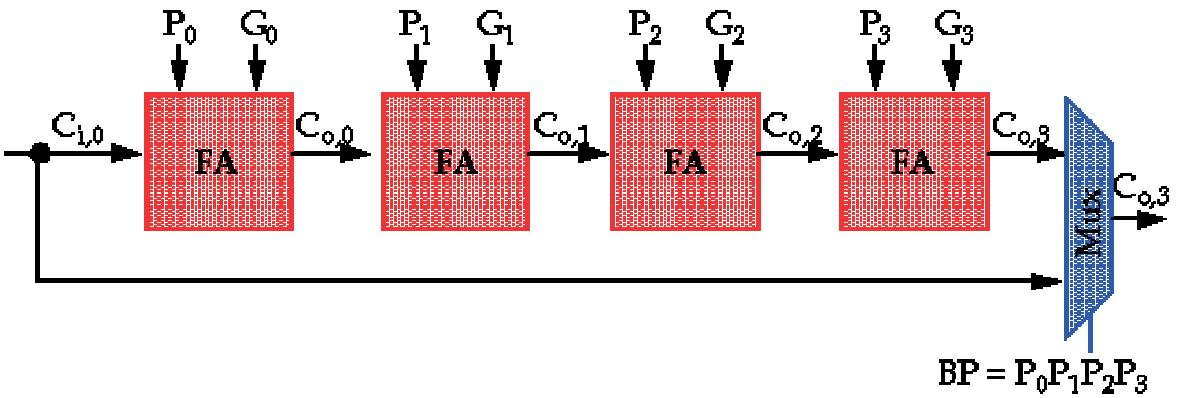
Block-Carry-Transfer: $BT_{(j_h:j_{h+1}-1)} = \bigwedge_{i=j_h}^{j_{h+1}-1} T_i$

Block-Carry-Berechnung: $BC_{j_{h+1}} = C_{j_{h+1}} + BT_{(j_h:j_{h+1}-1)}\, BC_{j_h}$

Der endgültige Wert von $BC_{j_{h+1}}$ stimmt mit dem endgültigen Wert von $C_{j_{h+1}}$ überein, steht aber im Fall $BT_{(j_h:j_{h+1}-1)} = BC_{j_h} = \mathit{true}$ schneller als beim Ripple-Carry-Addierer zur Verfügung.

Gestaltung der Carry-Skip-Logik (1)

Die Zusammenführung des Carry-Pfads und des Block-Carry-Pfads kann durch ein OR-Gatter (möglicherweise auch integriert in die Übertragsberechnung des höchstwertigen Volladdierers), ein Wired-OR oder einen Multiplexer erfolgen.



Aus Pass-Transistoren oder Transmissionsgattern aufgebaut, ist der Multiplexer in der Regel schneller als das OR-Gatter und elektrisch zuverlässiger als das Wired-OR.

Statt des Blocktransfersignals BT ist für einen Multiplexer unbedingt das entsprechende Blockpropagierungssignal $BP = \prod_i P_i$ zu verwenden, damit der Bypass nur öffnet, wenn ansonsten Propagierung durch den gesamten Block erforderlich wäre.

Beispiel:

Mit $C_0 = 0$ und $G_0 = G_1 = 1$ folgt $C_2 = 1$,

was mit $P_0 = P_1 = BP_{(0:1)} = 0$ durch den Multiplexer korrekt zu $BC_2 = 1$ ausgewertet wird.

Dagegen liefert $T_0 = T_1 = BT_{(0:1)} = 1$ fälschlicherweise $BC_2 = 0$.

Gestaltung der Carry-Skip-Logik (2)

Falls das Transfer-Signal T_i einer Stufe nicht bereits bei der Berechnung $C_{i+1} = G_i + T_i C_i$ bzw. $C_{i+1} = T_i(G_i + C_i)$ anfällt, muss es separat berechnet werden.

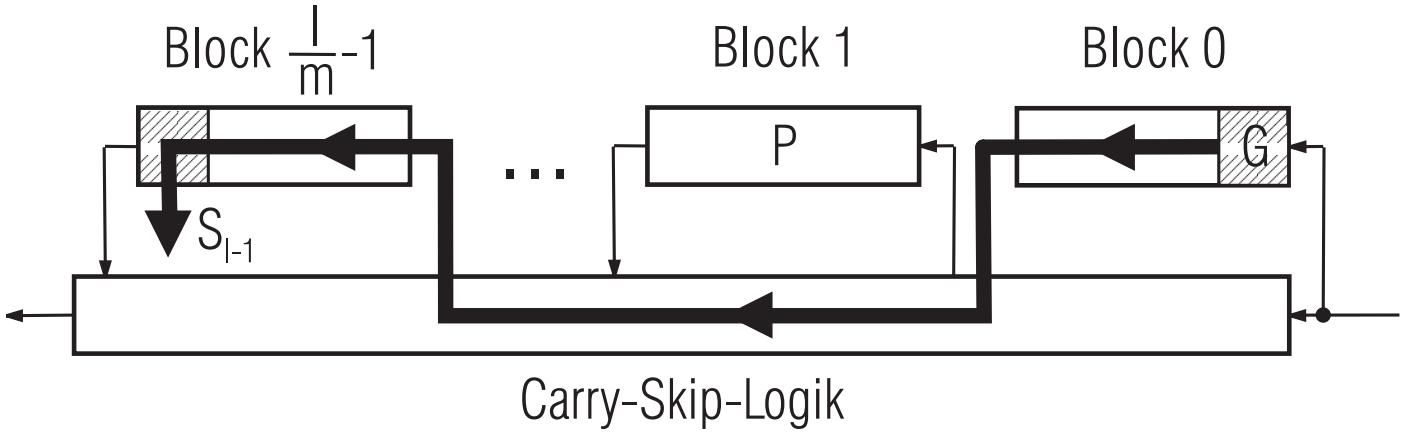
Von Interesse ist dabei evtl. auch die Formel $P_i = T_i(\overline{A} + \overline{B})$.

In CMOS muss in der Regel ein separates Gatter für T_i angelegt werden.

Die Prädikate *BT* bzw. *BP* werden simultan und in kleiner, zumindest bei Blöcken gleicher Länge konstanter, Zeit berechnet.

Die Erweiterung eines Ripple-Carry-Addierers zum Carry-Skip-Addierer erfordert insgesamt nur einen geringen Hardware-Aufwand.

Carry-Skip-Addierer: Kritischer Pfad bei Blöcken gleicher Länge



Maximale Latenz: $(l/m - 2) \times t_S + 2 \times m \times t_{VA} + const$,
wobei m die Blocklänge und t_S die Latenz der Carry-Skip-Logik eines Blocks ist.

Optimierungsproblem: $\min_m \; l/m \times t_S + 2 \times m \times t_{VA}$

Optimale Blocklänge m hängt für festes l vom Verhältnis $t_S : t_{VA}$ ab.
Für $t_S = t_{VA}$ (häufigste Grundannahme) gilt $m_{opt} \approx \sqrt{l/2}$.

Beispiel (Parhami): $l = 32$, $m_{opt} = 4$, als Latenz ergibt sich 25 t_g mit Skip, 64 t_g ohne Skip.

Zu beiden Seiten des Optimalwerts ist die Zielfunktion jeweils monoton
(wichtig, falls $\sqrt{l/2}$ nicht ganzzahlig oder kein Teiler von l).

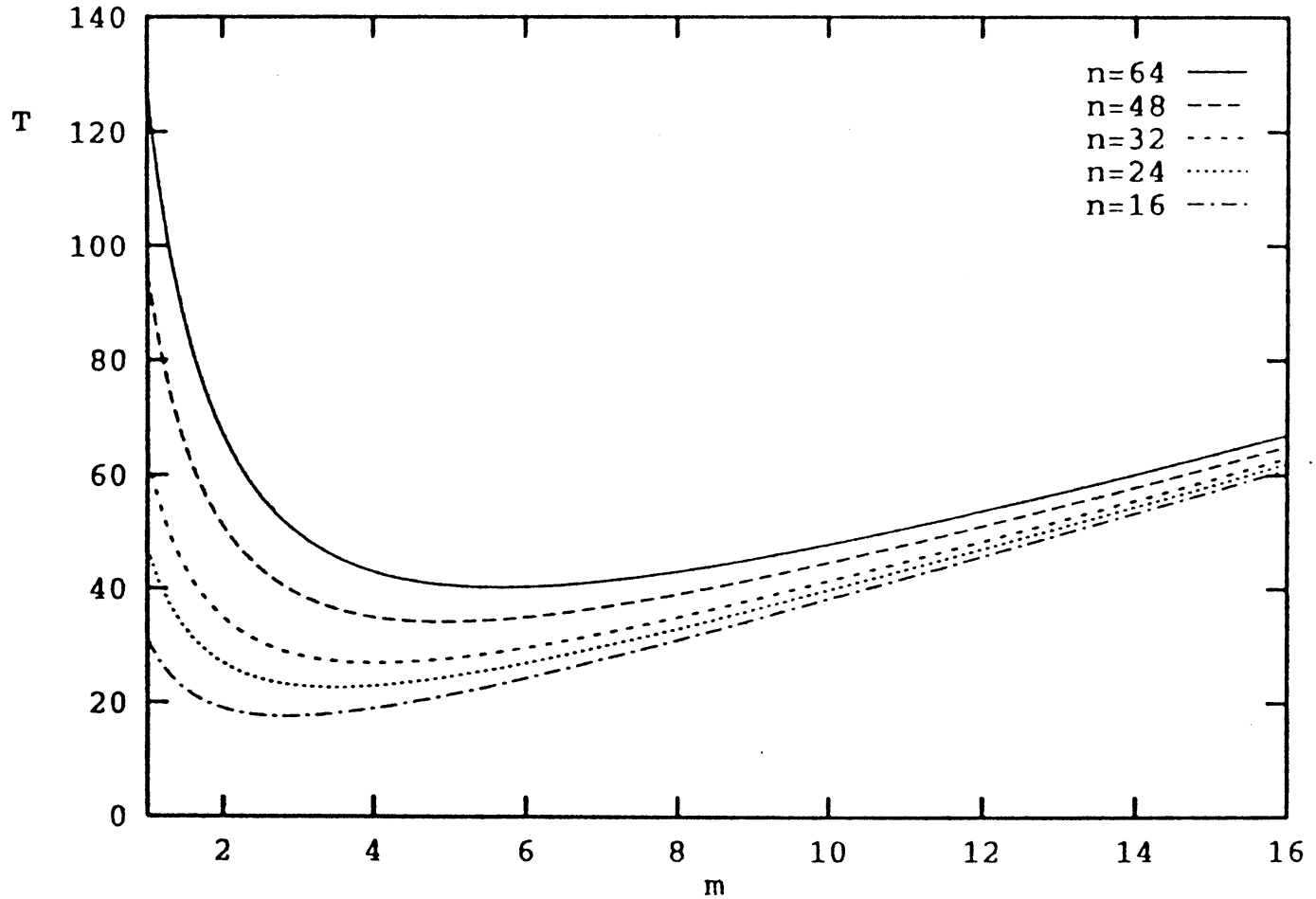
Beispielhafte Bewertung von Carry-Skip-Addierern

Länge (l)	Blocklänge (m)	Aufwand (a)	Latenz (t)	$a \times t \times 10^{-4}$
16	2	288	19	0,55
	4	280	19	0,53
	8	276	31	0,86
32	2	576	35	2,02
	4	560	27	1,51
	8	552	35	1,93
	16	548	64	3,45
64	2	1152	67	7,72
	4	1120	43	4,82
	8	1104	43	4,75
	16	1096	67	7,34
	32	1092	127	13,87

(aus A. R. Omondi: Computer arithmetic systems, 1994, p. 33)

Blocklängen m_1, m_2 mit $m_1 \times m_2 = (t_S : t_{VA}) \times l/2$ ergeben gleiche Latenzen!

Carry-Skip-Addierer: Einfluss von Länge und Blocklänge auf die Latenz



Blocklängen m_1, m_2 mit $m_1 \times m_2 = (t_S : t_{VA}) \times l/2$ ergeben gleiche Latenzen!

Weitere Verkürzung der Latenz des Carry-Skip-Addierers

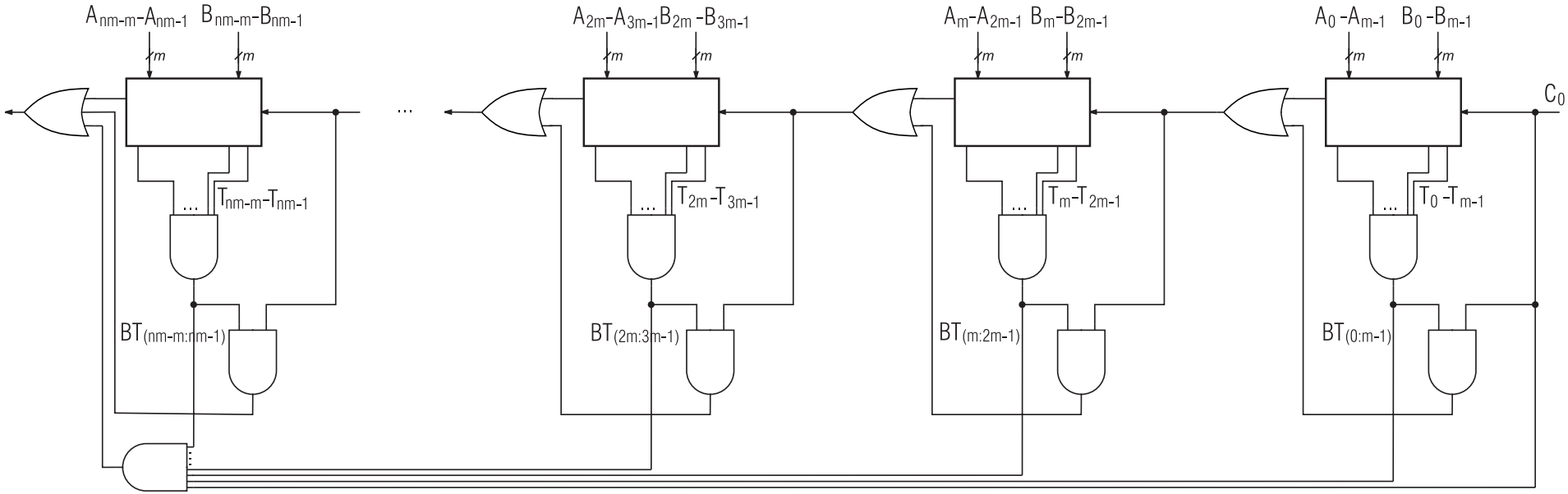
- Blöcke unterschiedlicher Länge.
  Generelle Strategie: Blöcke werden zur Mitte hin länger.
  Beispiel: Für $l = 32$ wähle $[3, 4, 5, 6, 5, 4, 3, 2]$ als Blocklängen.
  t_S variiert evtl., wenn Fan-in der Und-Gatter nicht mehr für gesamte Skip-Logik ausreicht.
- Mehrstufige Carry-Skip-Addierer.
  Anwendung desselben Prinzips auf Blöcke höherer Ordnung.

Schwieriges Optimierungsproblem; verschiedene Kosten trotz gleicher maximaler Latenz.

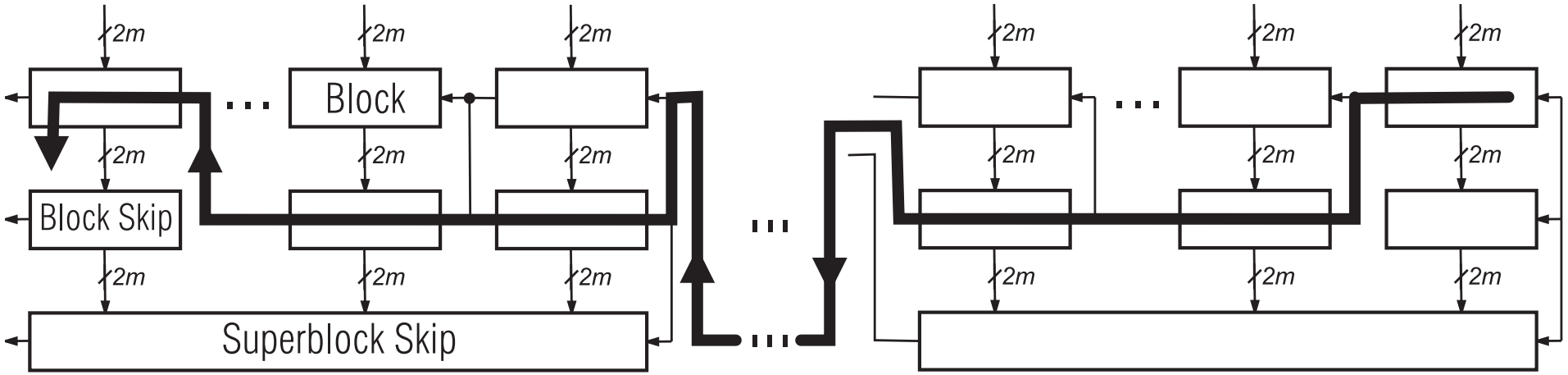
Weiterführende Literatur

P. K. Chan und M. D. F. Schlag: A note on designing two-level carry-skip adders.
Journal of VLSI Signal Processing 3, 275–281 (1991)

Modul eines zweistufigen Carry-Skip-Addierers



Kritischer Pfad im zweistufigen Carry-Skip-Addierer



Beispielhafte Bewertung zweistufiger Carry-Skip-Addierer

Länge (l)	Blocklänge		Aufwand (a)	Latenz (t)	$a \times t \times 10^{-4}$
	m	n			
16	2	2	300	12	0,36
		4	298	16	0,48
	4	2	286	16	0,46
32	2	2	600	20	1,20
		4	596	20	1,19
		8	594	32	1,90
	4	2	572	20	1,14
		4	570	24	1,37
	8	2	558	32	1,79
64	2	2	1200	36	4,32
		4	1192	28	3,34
		8	1188	36	4,28
		16	1186	64	7,59
	4	2	1144	28	3,20
		4	1140	28	3,19
		8	1138	40	4,55
	8	2	1116	36	4,01
		4	1114	40	4,46
	16	2	1102	64	7,05

(aus A. R. Omondi: Computer arithmetic systems, 1994, p. 42)

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Carry-Lookahead-Addierer, baumartige Addierer

Carry-Lookahead-Prinzip

Prinzipielle Idee: Die Überträge $C_1, C_2, \ldots, C_l$ werden direkt und parallel aus den Eingaben A_i und B_i generiert und liegen dadurch früher vor als beim Ripple-Carry-Addierer.

Direkte Übertragserzeugung jedoch meist nicht durch ein einziges zweistufiges Schaltnetz realisierbar: Aufwand für Übertragsberechnung steigt exponentiell in Operandenlänge l.

Abhilfe: Übertragserzeugung durch dreistufiges Schaltnetz.
Technik: Komprimierung der Eingabedaten mittels G_i und P_i (oder T_i).

Entrekursivierung der Übertragsberechnung im Carry-Lookahead-Addierer:
Ausgangspunkt ist die Basisrekurrenz des Volladdierers: $C_{i+1} = G_i + P_i C_i$
(analog wird für die Darstellung $C_{i+1} = G_i + T_i C_i$ verfahren)

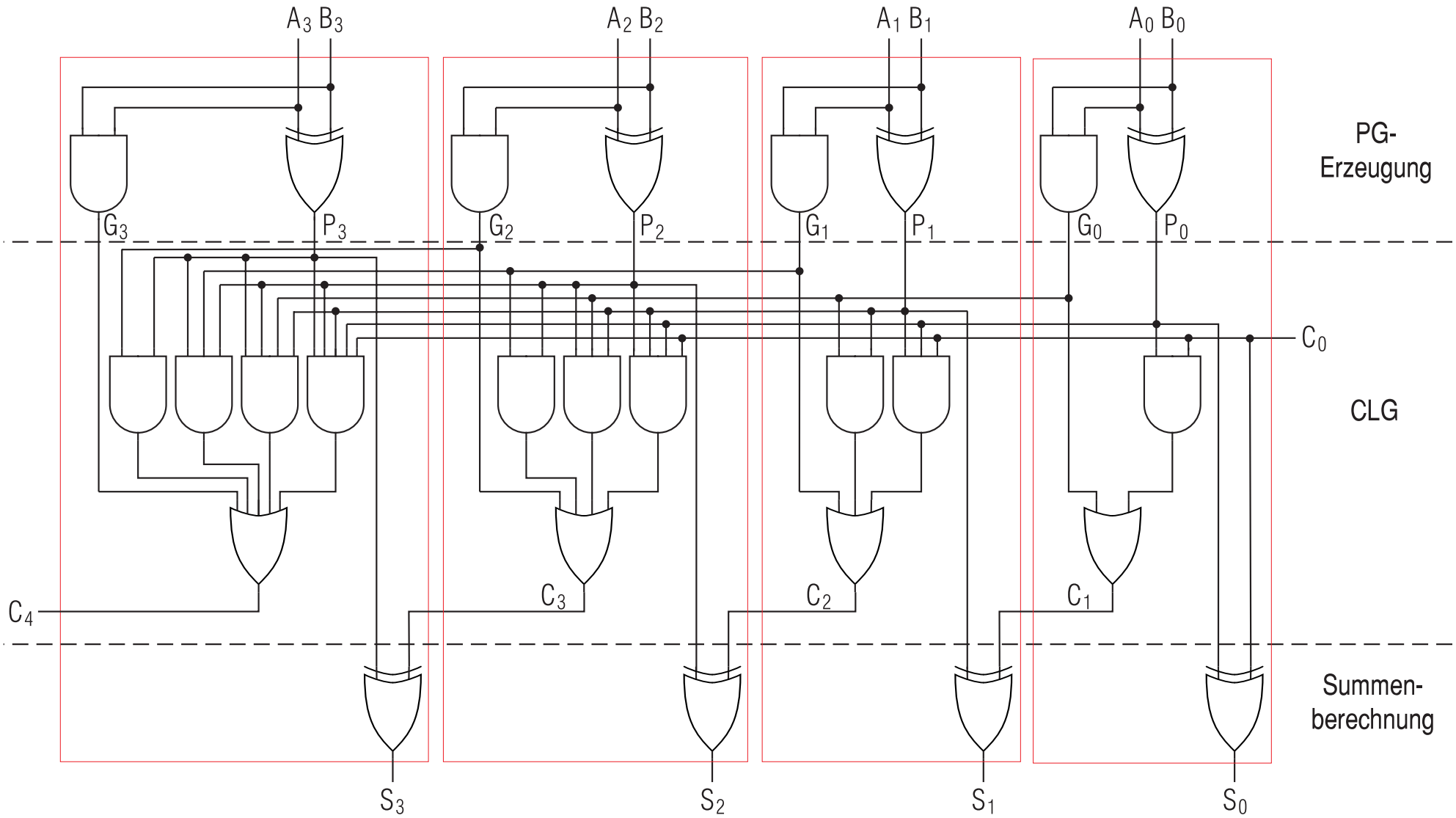
Substitution $C_i = G_{i-1} + P_{i-1} C_{i-1}$ liefert $C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} C_{i-1}$

Vollständige Eliminierung aller C_i (außer C_0) liefert
$C_{i+1} = G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots + P_i \cdots P_1 G_0 + P_i \cdots P_0 C_0$

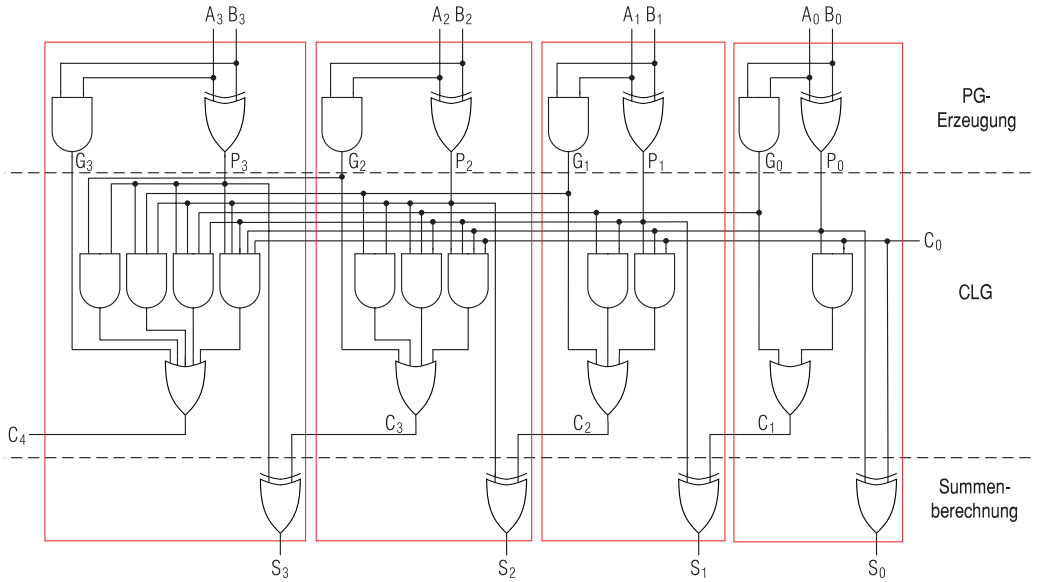
Die Überträge in einem Carry-Lookahead-Addierer für 4 Bit Wortlänge
werden somit nach folgenden Formeln berechnet:

$$
\begin{aligned}
C_1 &= G_0 + P_0 C_0 \\
C_2 &= G_1 + P_1 G_0 + P_1 P_0 C_0 \\
C_3 &= G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0 \\
C_4 &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_0
\end{aligned}
$$

Carry-Lookahead-Addierer (CLA) für 4 Bit Wortlänge



Nachteile großer Carry-Lookahead-Addierer in AND-OR-Logik

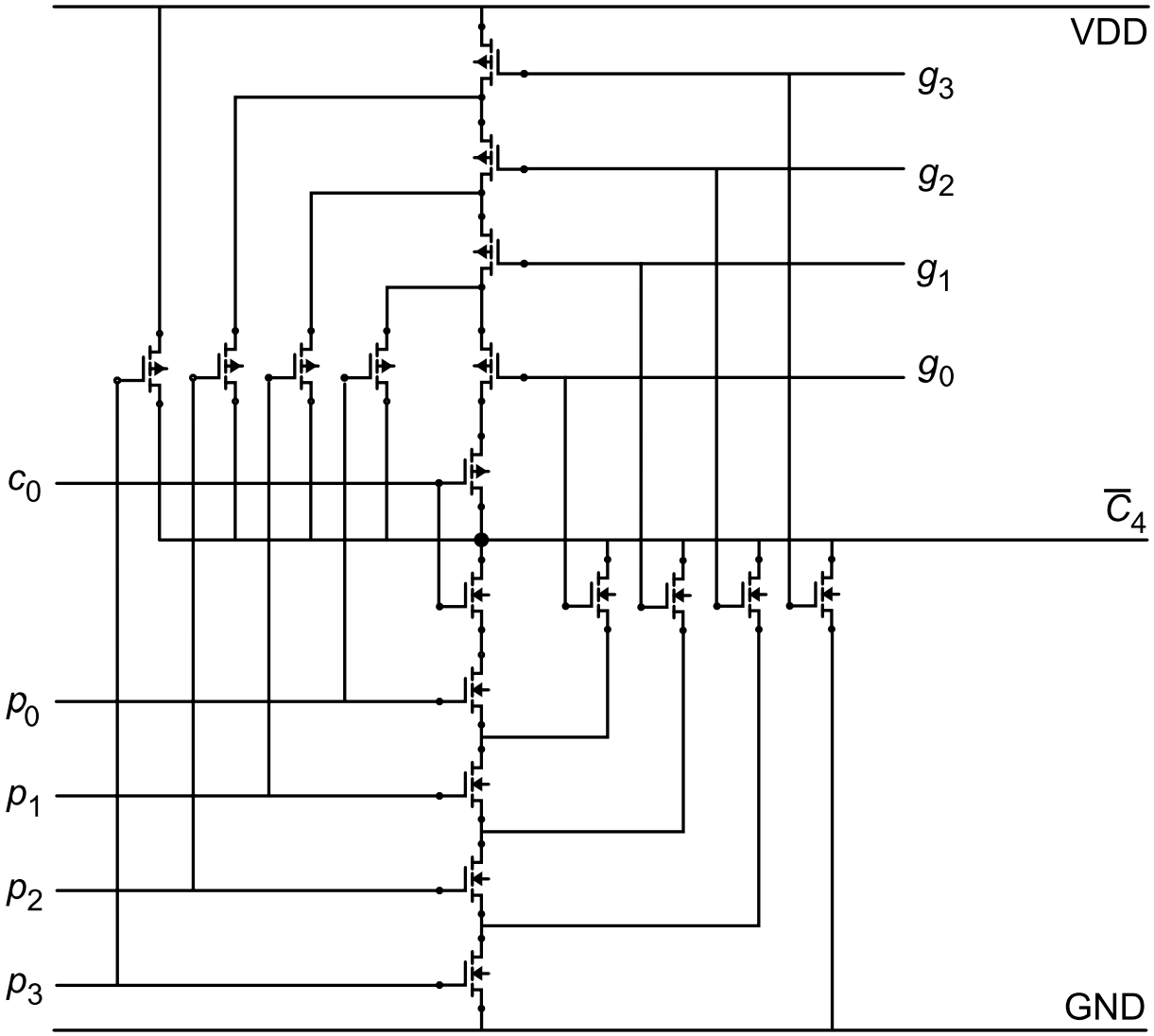


Der bisher beschriebene Ansatz besitzt eine Reihe von Nachteilen:

- Es sind Gatter mit einem Fan-in bis zu $l + 1$ nötig.
  Für großes l sind solche Gatter technisch nicht effizient realisierbar.
- Die Anzahl der Und-Gatter wächst quadratisch in l.
  Die Kostenfunktion ist wegen ansteigendem Fan-in der Gatter meist sogar kubisch in l.
- Die Verbindungskomplexität ist hoch.
- Hoher Fan-out für die Signale C_0, G_i und P_i (bzw. T_i).

Carry-Lookahead-Addierer dieser Bauart werden normalerweise nur bis $l = 4$ benutzt.

Statische CMOS-Logik für Carry-Lookahead-Generator (CLG)



(aus N. Reifschneider: CAE-gestützte IC-Entwurfsmethoden, 1998, p. 127)

In dieser Schaltung muss für alle Eingänge p_i zwingend das Transfer-Prädikat T_i verwendet werden!

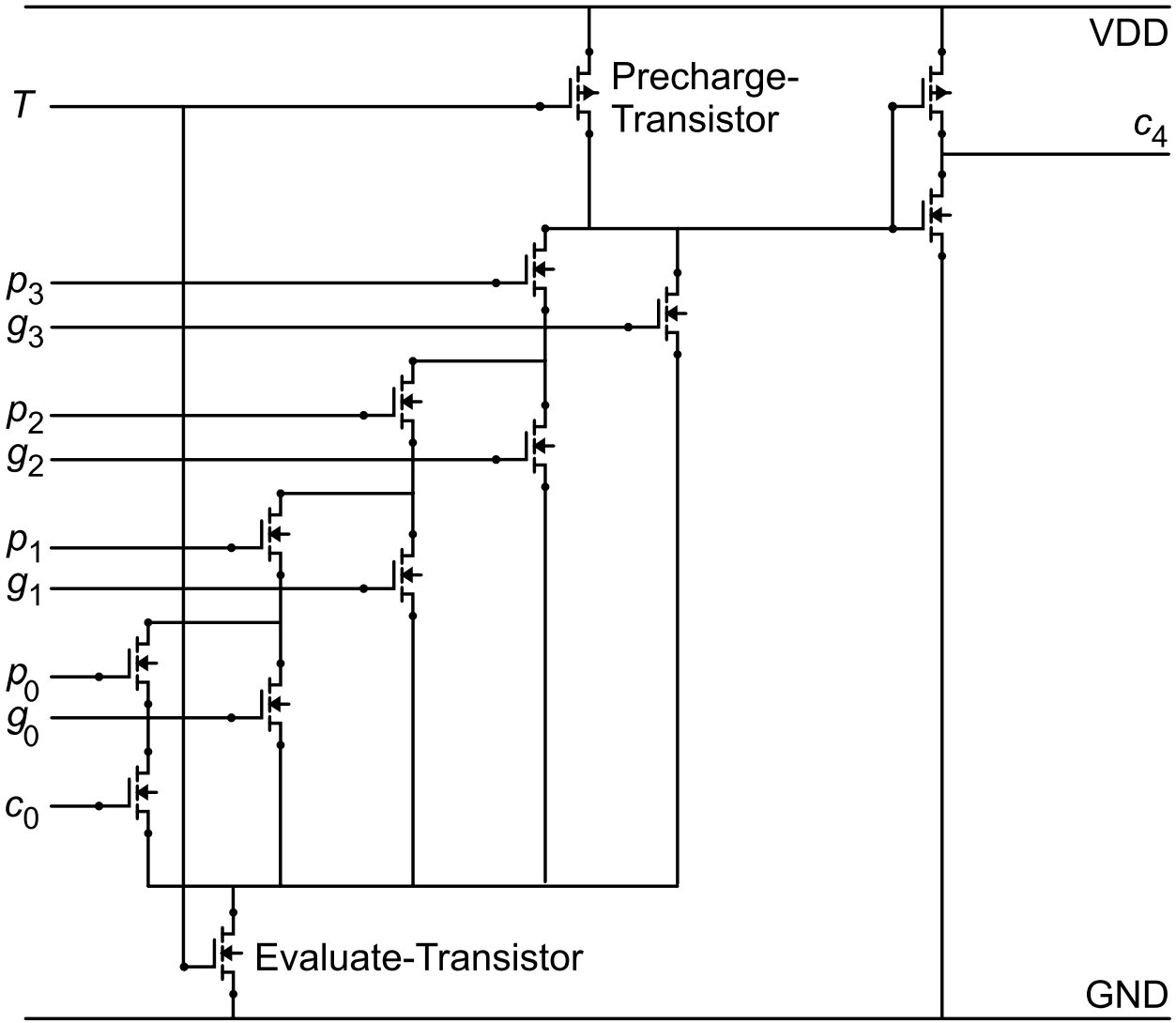
Ansonsten kann es zum Kurzschluss kommen.

Beispiel: $G_3 = 1 \Rightarrow P_3 = 0$.

Fläche des Gatters $O(l^2)$, $O(l)$ Transistoren.

Latenz $O(l^2)$ wegen Serienwiderstand $O(l)$ und Diffusionskapazität $O(l)$.

Carry-Lookahead-Generator in Domino-Logik



(aus N. Reifschneider: CAE-gestützte IC-Entwurfsmethoden, 1998, p. 128)

In dieser Schaltung kann für einen Eingang p_i wahlweise das Transfer-Prädikat T_i oder das Propagate-Prädikat P_i verwendet werden.

Serienwiderstand bis zu $l + 2$ Transistoren.

$$C_4 = G_3 + P_3(G_2 + P_2(G_1 + P_1(G_0 + P_0 C_0)))$$

Ling-Addierer

Der Ling-Addierer gilt derzeit als schnellster bekannter Addierer.

Statt intermediärer Überträge C_i werden Pseudo-Überträge $H_i = C_i + C_{i-1}$ berechnet. Es folgt

$$C_i = H_i T_{i-1}$$

$$H_i = G_{i-1} + H_{i-1} T_{i-2}$$

$$S_i = P_i \oplus H_i T_{i-1} = (T_i \oplus H_{i+1}) + H_i G_i T_{i-1}$$

So ergibt sich z. B. statt

$$C_i = G_{i-1} + T_{i-1} G_{i-2} + T_{i-1} T_{i-2} G_{i-3} + T_{i-1} T_{i-2} T_{i-3} G_{i-4} + T_{i-1} T_{i-2} T_{i-3} T_{i-4} C_{i-4}$$

mit einem Aufwand von 19 Gate-Eingängen (bzw. 14 mit Wired-OR) nun

$$H_i = G_{i-1} + G_{i-2} + T_{i-2} G_{i-3} + T_{i-2} T_{i-3} G_{i-4} + T_{i-2} T_{i-3} T_{i-4} T_{i-5} H_{i-4}$$

mit einem Aufwand von 15 Gate-Eingängen (bzw. 10 mit Wired-OR).

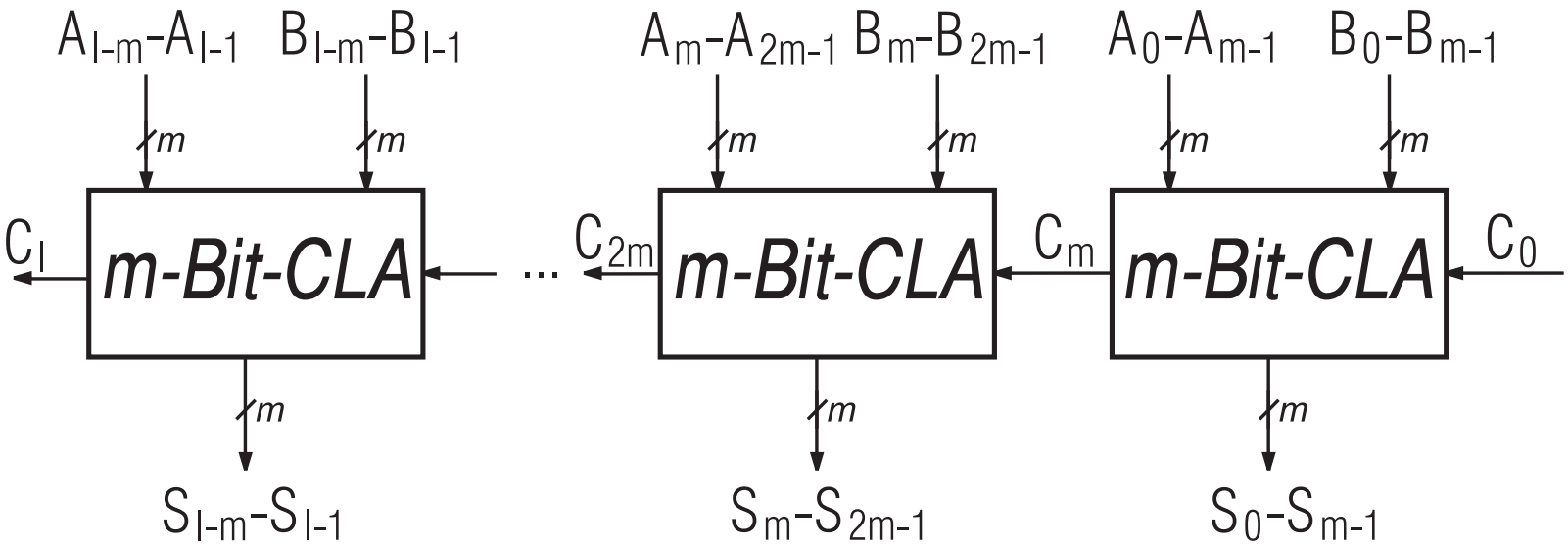
Ripple-Block-Carry-Lookahead-Addierer (RCLA)

Prinzip: Verbinden „kleiner“ CLA mittels Ripple-Carry-Prinzip.

Zwecks höherer Modularität werden meist lauter gleiche CLA für den Aufbau gewählt.

Ausführungsdauer bei Blocklänge m verringert sich gegenüber reinem Ripple-Carry-Addierer um nahezu den Faktor m, Aufwand wächst im Gegensatz zum reinen CLA nur linear in l.

Modulbibliotheken enthalten häufig kleine CLA, Blocklänge $m = 4$ ist praktisch Standard.



Superblock-Carry-Lookahead

Weitere Beschleunigung gegenüber RCLA: Eingehende Überträge für die einzelnen Blöcke werden selbst wieder durch einen CLG berechnet, und damit alle Blocküberträge parallel.

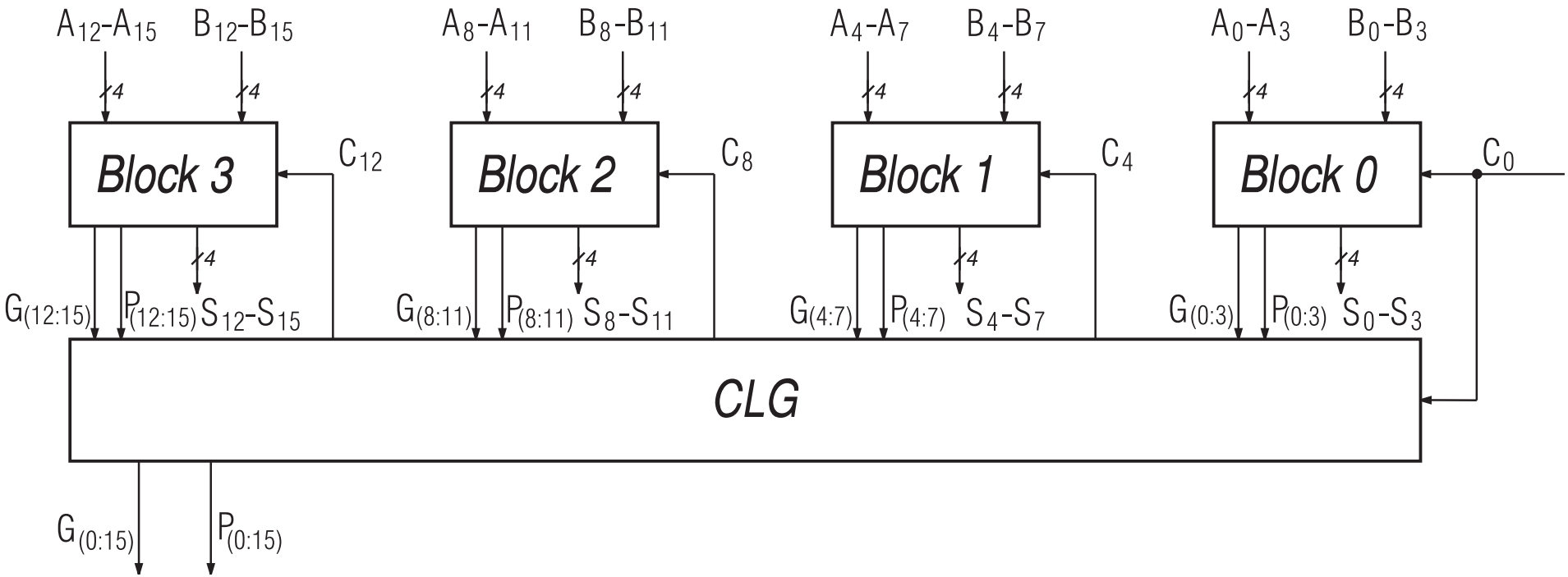
Zusätzlich zu G_i und P_i jedes einzelnen Volladdierers müssen für jeden Block entsprechende Prädikate *block-generate* und *block-propagate* berechnet werden.

Der dazu gegenüber dem RCLA nötige Aufwand ist relativ gering,
da dann in den Blöcken keine ausgehenden Überträge mehr erzeugt werden müssen.

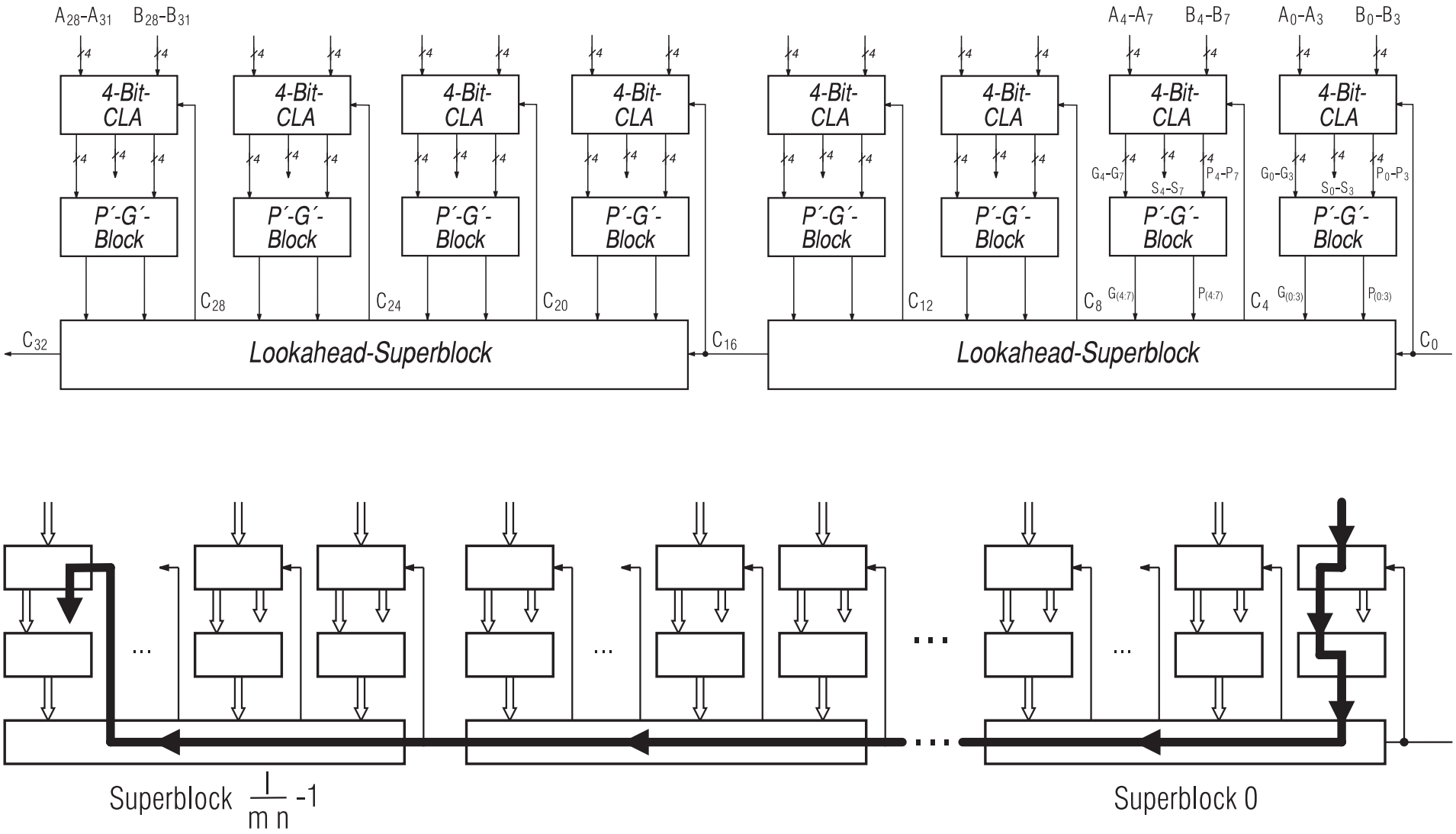
Beispiel für Wortlänge $l = 16$ und Blocklänge $m = 4$:

$$
\begin{aligned}
G_{(0:3)} &= G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 & \qquad P_{(0:3)} &= P_3 P_2 P_1 P_0 \\
G_{(4:7)} &= G_7 + P_7 G_6 + P_7 P_6 G_5 + P_7 P_6 P_5 G_4 & \qquad P_{(4:7)} &= P_7 P_6 P_5 P_4 \\
&\ldots \\
C_4 &= G_{(0:3)} + P_{(0:3)} C_0 \\
C_8 &= G_{(4:7)} + P_{(4:7)} G_{(0:3)} + P_{(4:7)} P_{(0:3)} C_0 \\
C_{12} &= G_{(8:11)} + P_{(8:11)} G_{(4:7)} + P_{(8:11)} P_{(4:7)} G_{(0:3)} + P_{(8:11)} P_{(4:7)} P_{(0:3)} C_0
\end{aligned}
$$

Superblock-Carry-Lookahead der Wortlänge 16 und Blocklänge 4



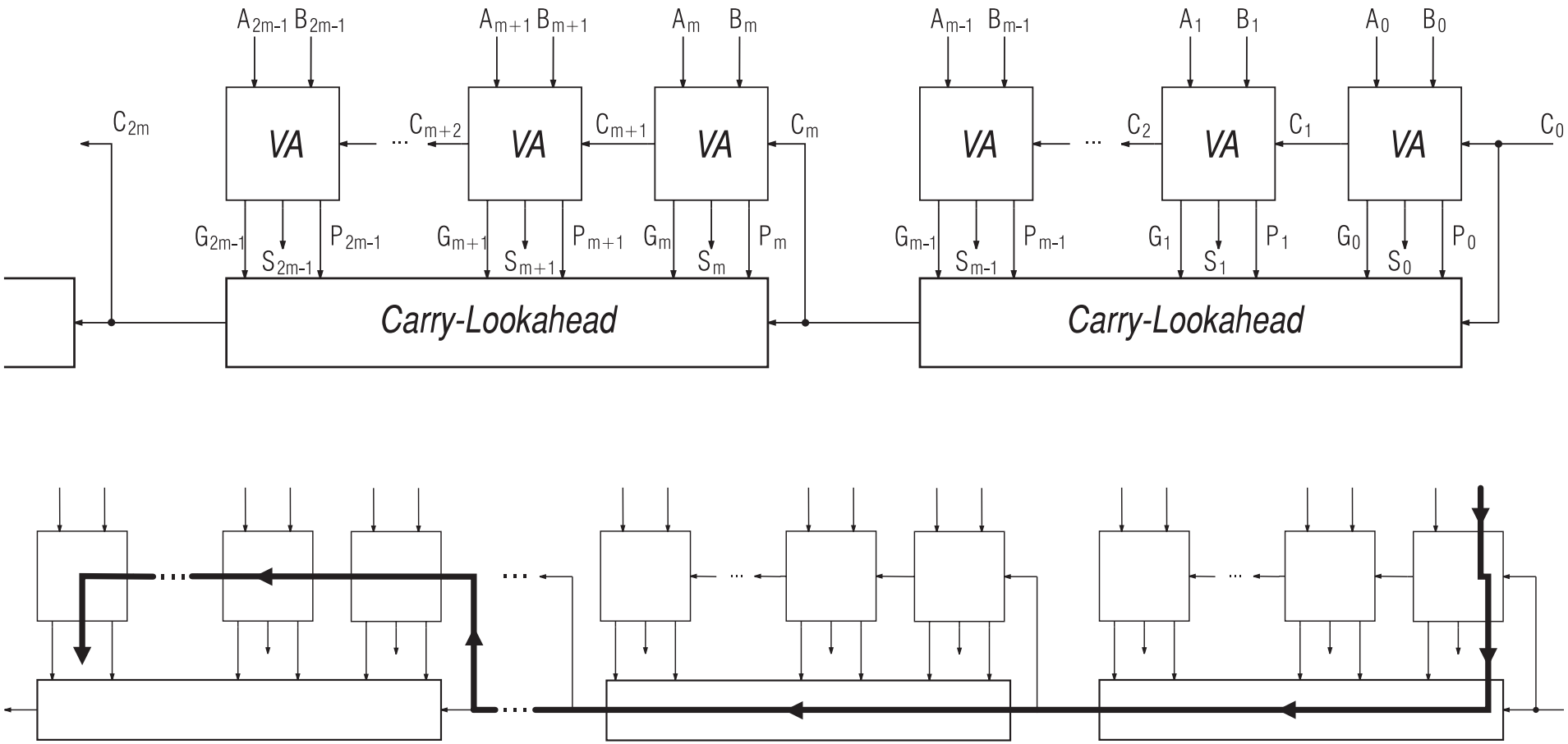
Superblock-RCLA (SRCLA)



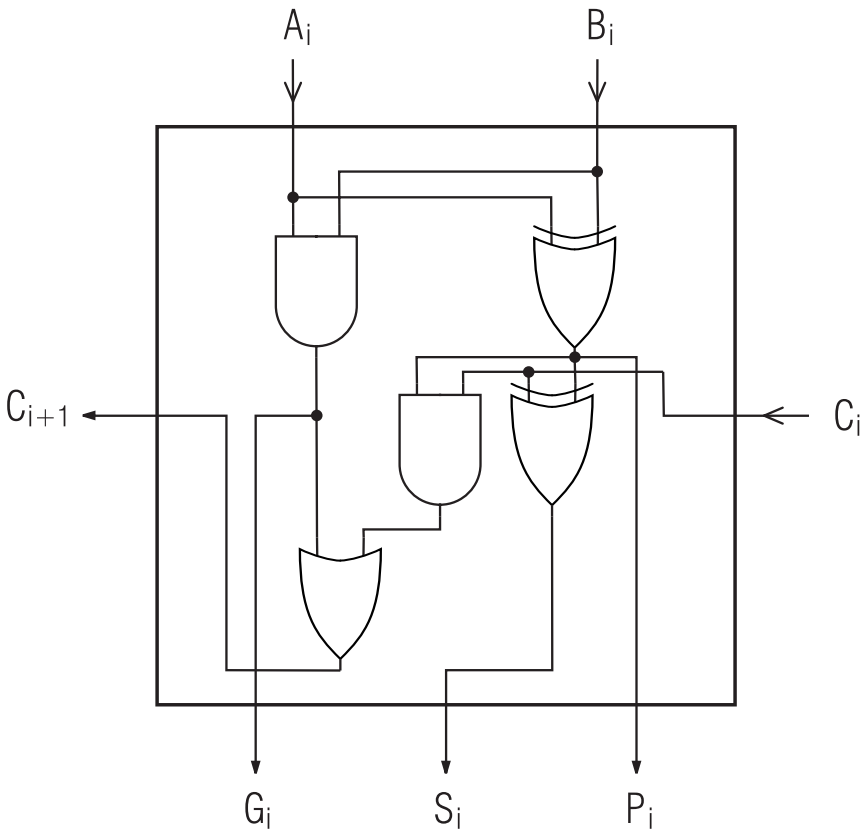
Verbindung mehrerer Superblock-Carry-Lookahead-Einheiten mittels Ripple-Carry-Prinzip.

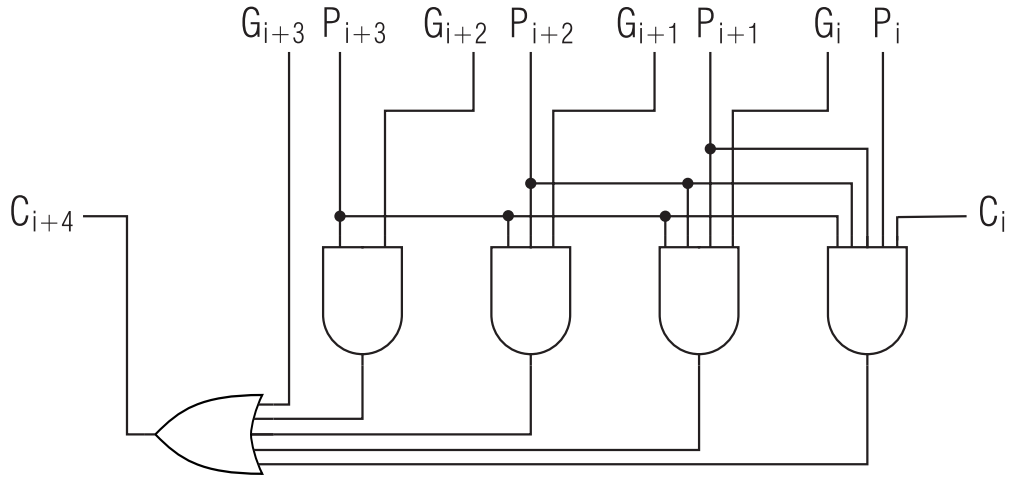
Block-Carry-Lookahead-Addierer (BCLA)

Ripple-Carry-Prinzip innerhalb jedes Blocks von Volladdierern,
Carry-Lookahead-Prinzip zur beschleunigten Erzeugung von Blocküberträgen:

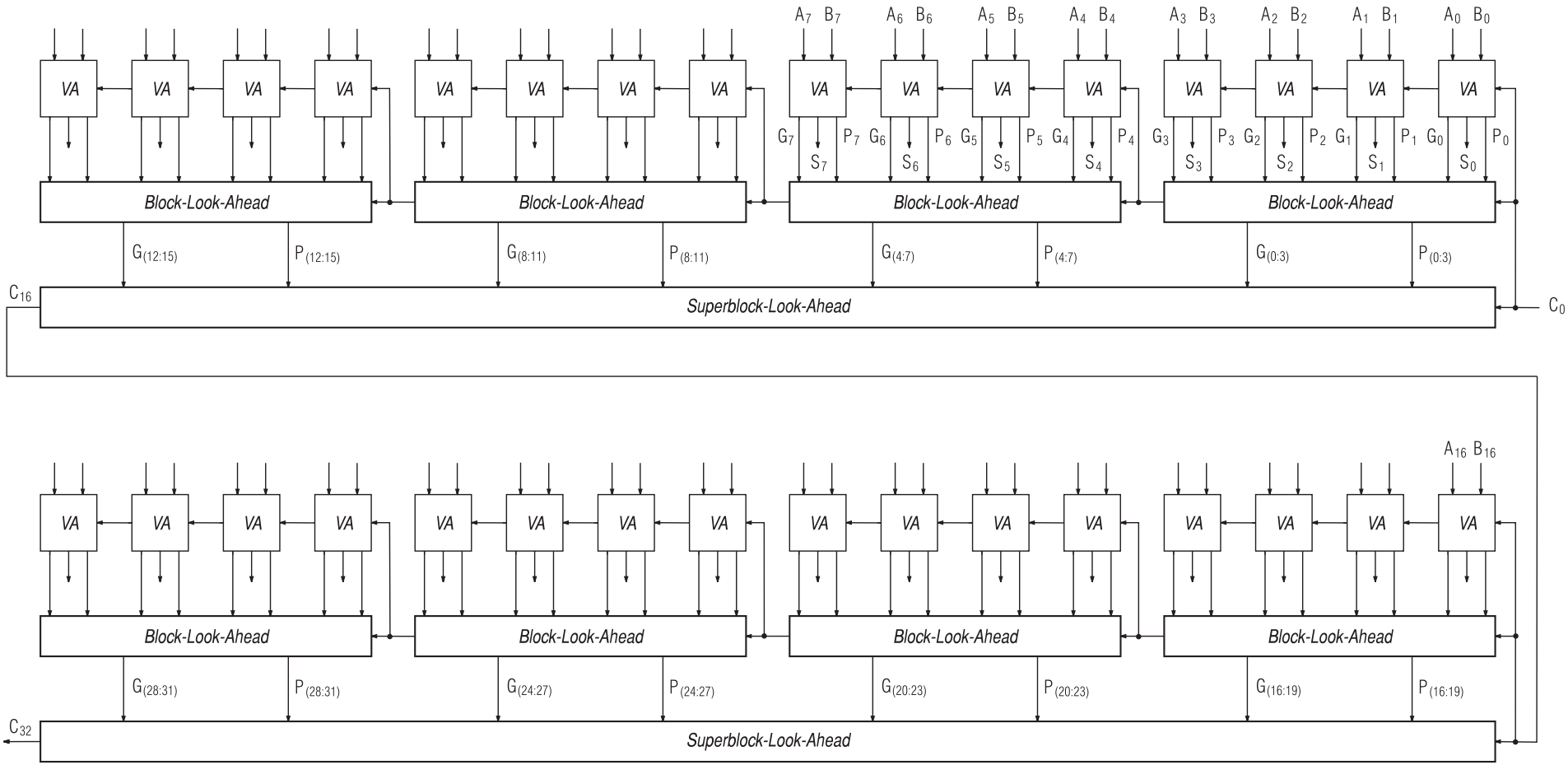


Erweiterter Volladdierer und Carry-Lookahead-Einheit im BCLA

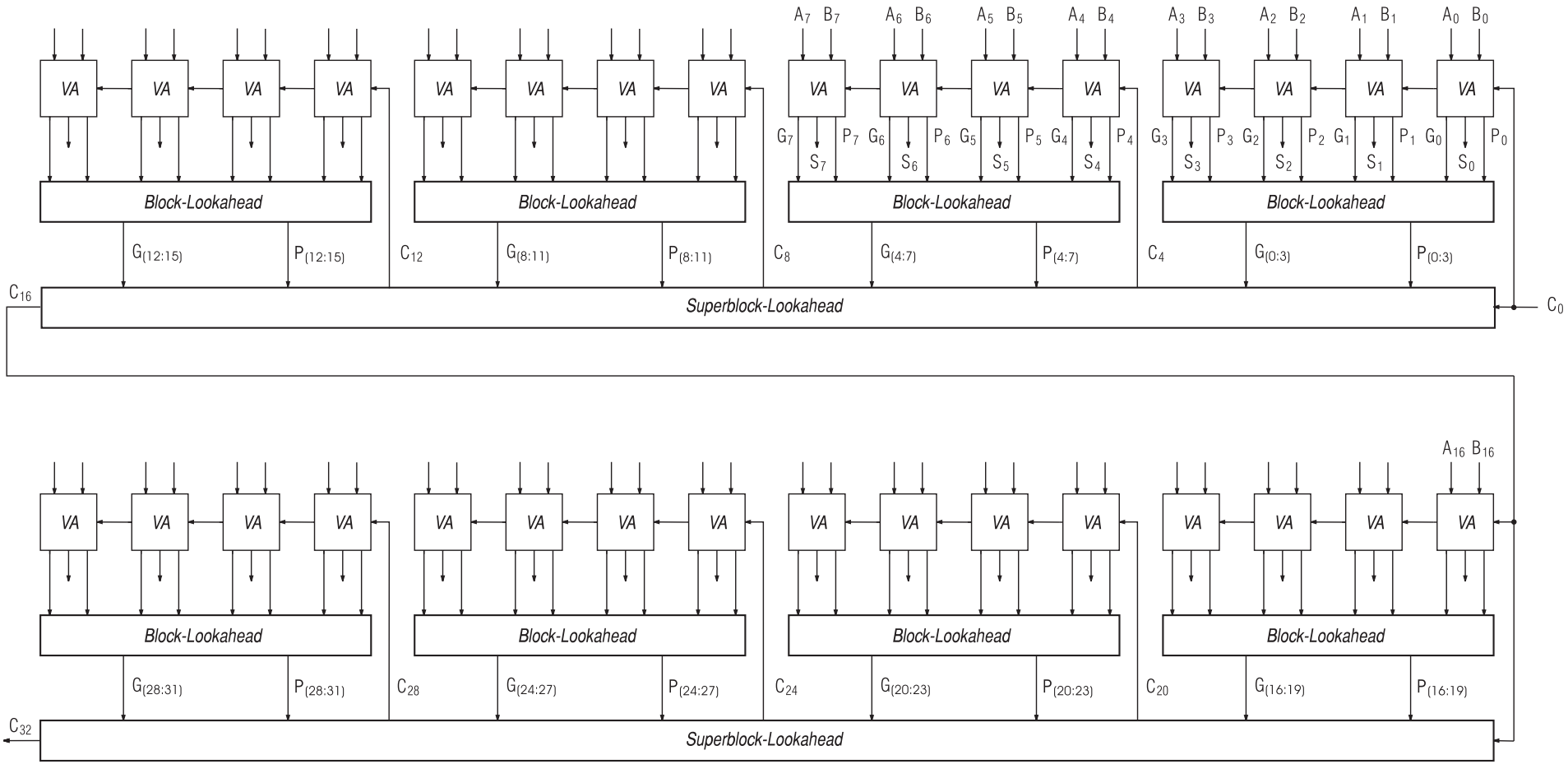




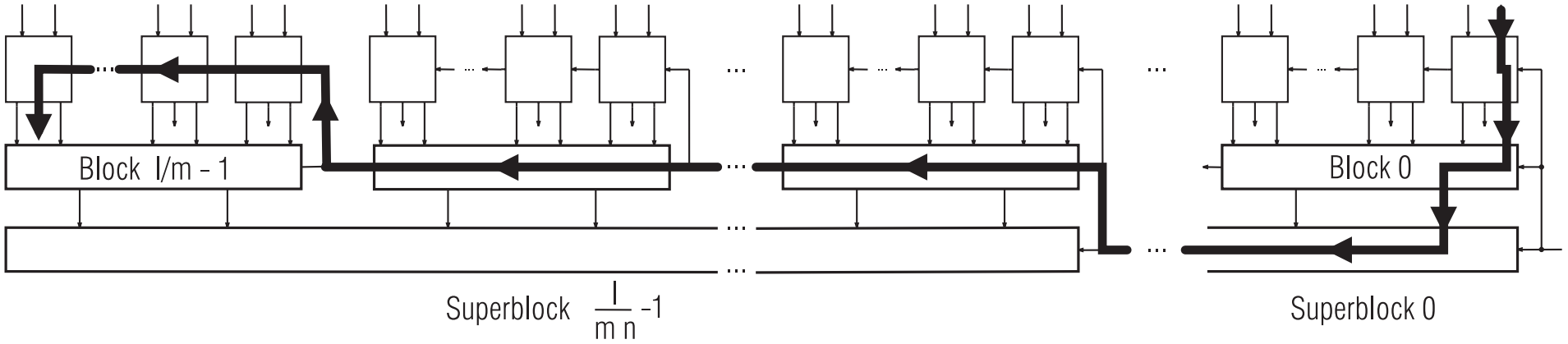
Superblock-BCLA (SBCLA)

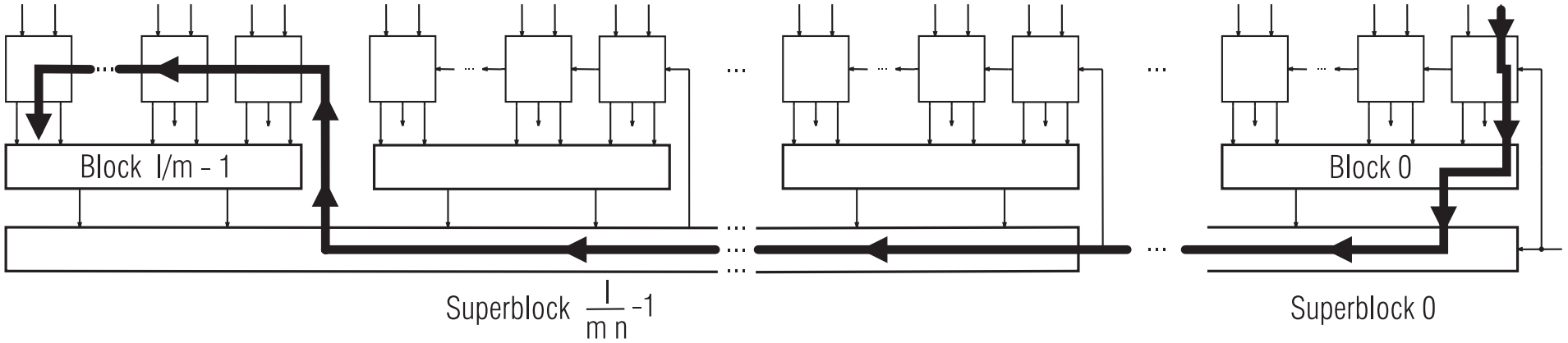


Modifizierter Superblock-BCLA (MSBCLA)

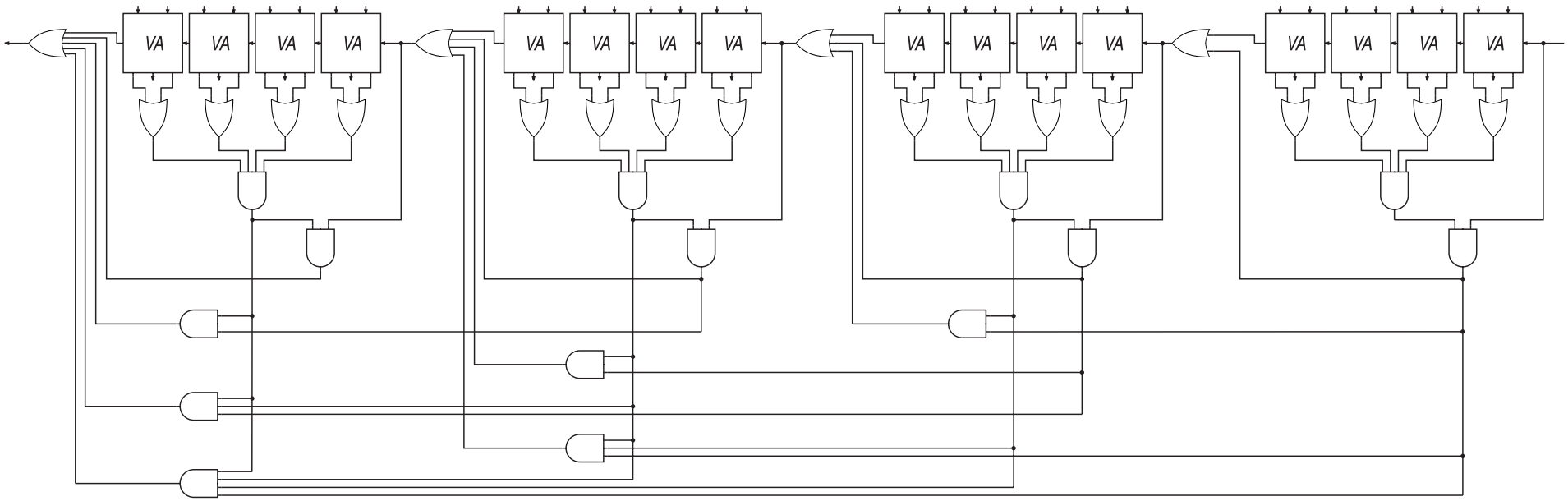


Kritische Pfade in SBCLA und MSBCLA



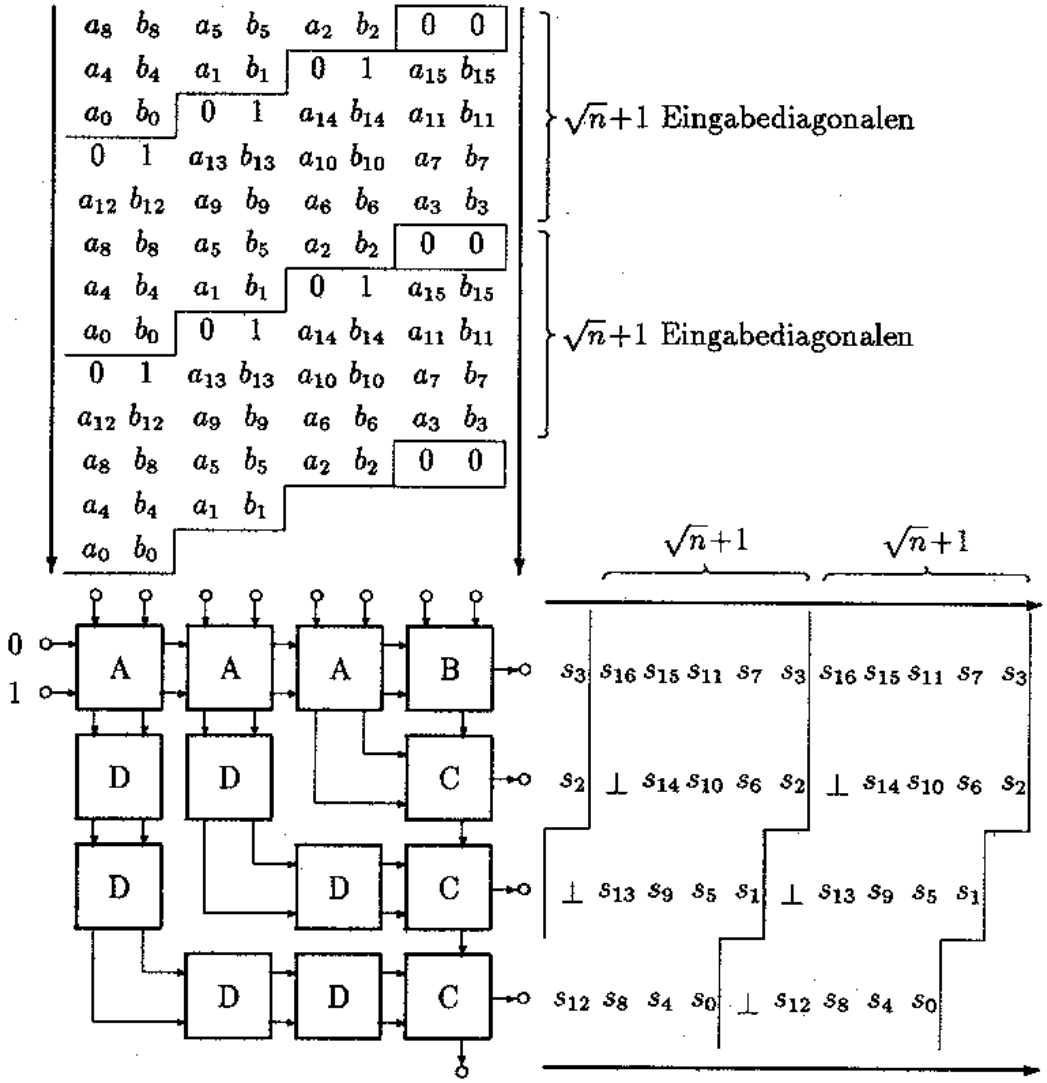


Kombination von Carry-Skip-Addierer und CLA



Ripple-, Skip- und Lookahead-Prinzip können relativ frei kombiniert werden.

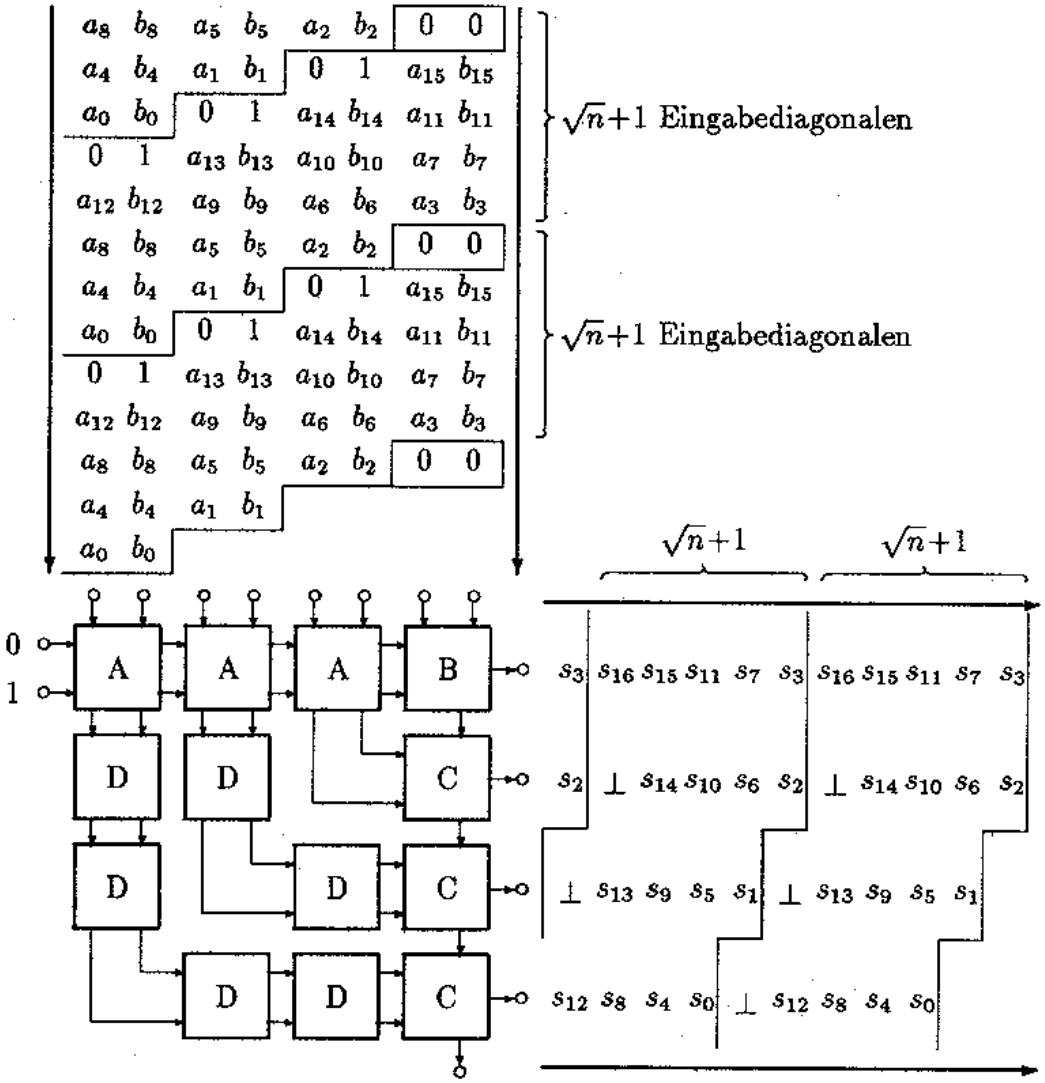
Systolischer Addierer (FASTA)



VLSI-gerechte Implementierung eines schnellen Addierers, mit Eingabedatensätzen.

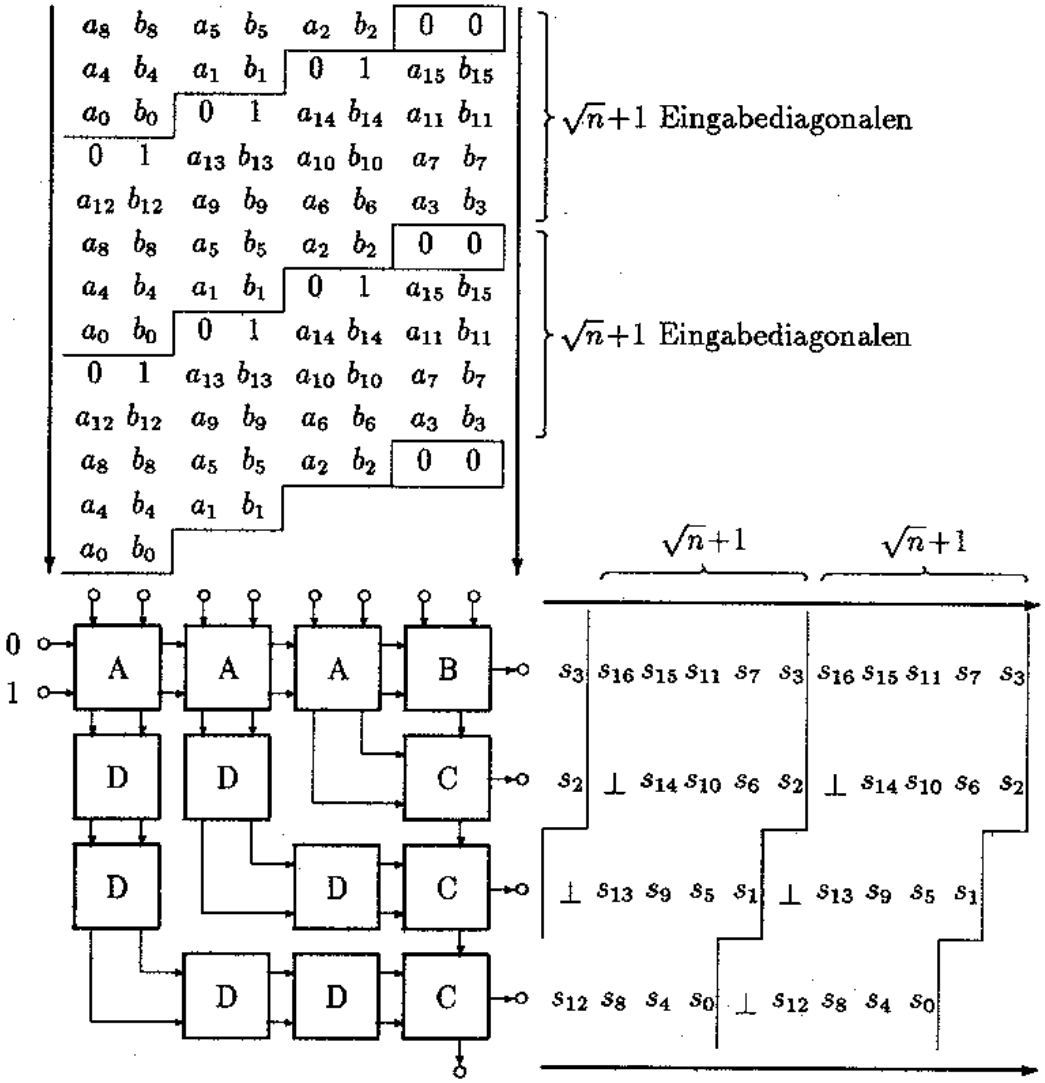
(aus L. Kühnel: Optimale systolische Präfixberechnungen, Dissertation, 1991.)

Wirkungsweise des systolischen Addierers FASTA (A-Zellen)



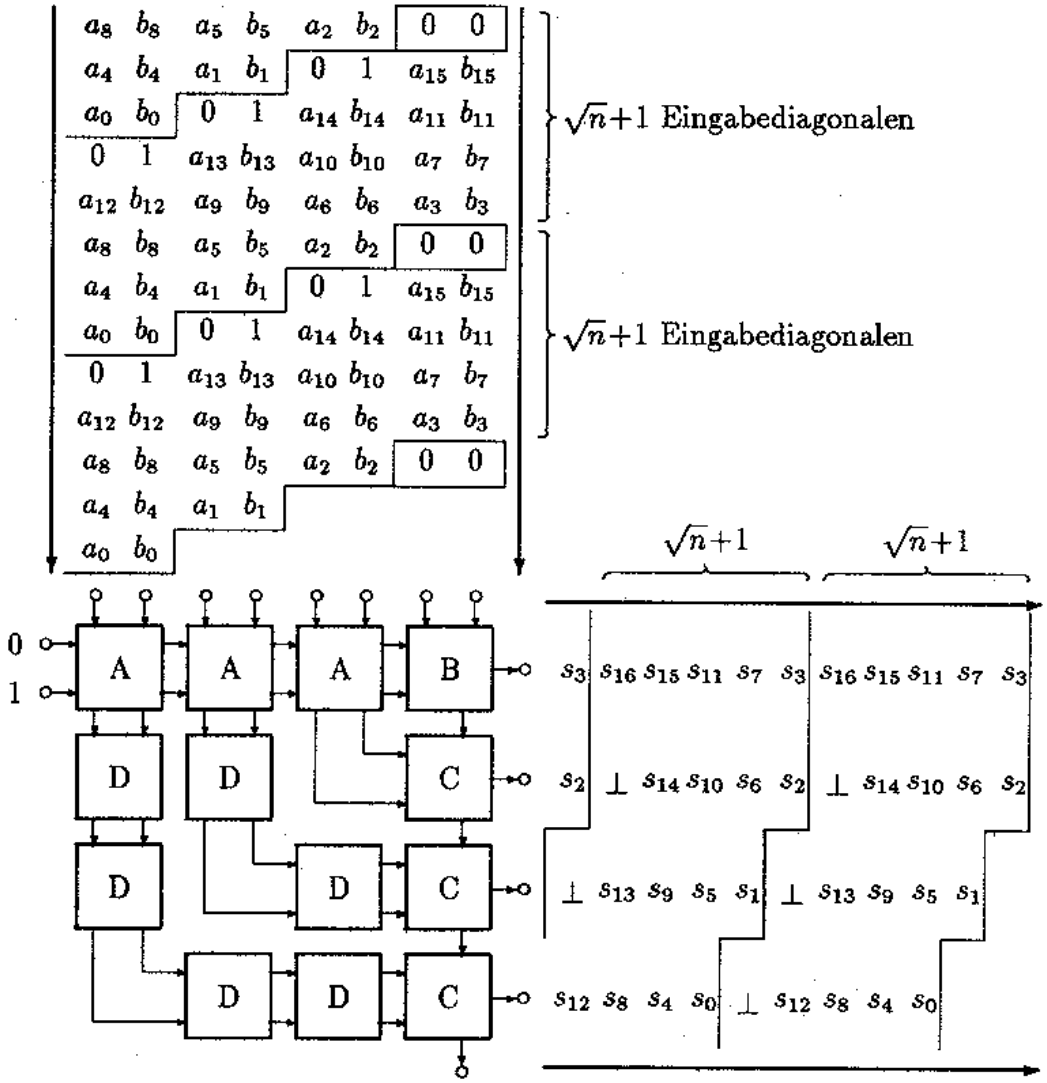
Jede A-Zelle erzeugt aus den Datenbits A_i, B_i für eine niederwertige Stelle des Blocks zunächst die Signale P_i und G_i, aus diesen dann mit Hilfe des P-Signals und des G-Signals des Vorgängerteilblocks die P- und G-Signale des Teilblocks sowie das vorläufige Summenbit S'_i.

Wirkungsweise des systolischen Addierers FASTA (B-Zelle)



Die B-Zelle erzeugt aus den Datenbits A_i, B_i für die höchstwertige Stelle des Blocks zunächst die Signale P_i und G_i, aus diesen dann mit Hilfe des P-Signals und des G-Signals des Vorgängerteilblocks sowie dem in der Zelle gespeicherten Blockausgangsübertrag des Vorgängerblocks den Blockausgangsübertrag und das Summenbit S_i.

Wirkungsweise des systolischen Addierers FASTA (C- und D-Zellen)



Jede C-Zelle berechnet aus einem S'_i sowie dem zugehörigen P-Signal des Teilblocks und dem Blockeingangsübertrag das end- gültige Summenbit S_i.

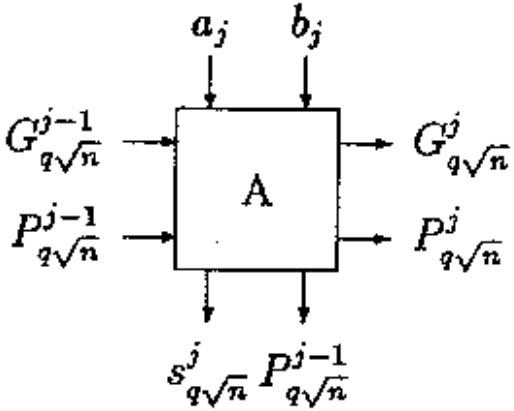
Die D-Zellen dienen lediglich der Synchronisierung.

Bezüge zwischen RCLA und FASTA

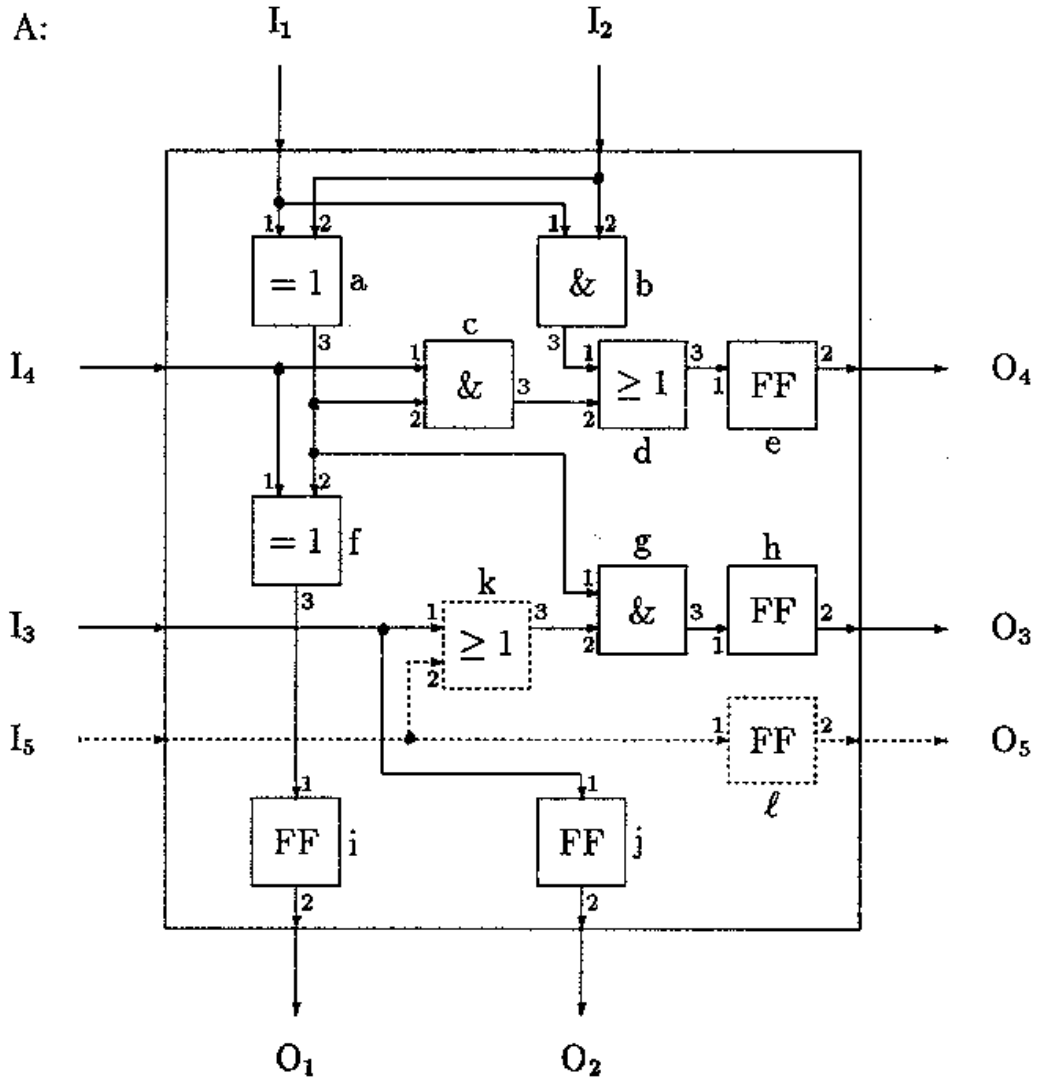
Die Wirkungsweise von FASTA ähnelt der eines Ripple-Block-Carry-Lookahead-Addieres (RCLA), dessen Blöcke $\sqrt{l}$-Bit-CLA sind:

- Je eine A-Zelle und eine zugehörige C-Zelle zusammen berechnen eine niederwertige Stelle des Blocks ähnlich wie in einem CLA.
- Die B-Zelle generiert das höchstwertige Summenbit und den Ausgangsübertrag eines Blocks ähnlich wie in einem CLA direkt aus seinem Eingangsübertrag.
- Die Rückkopplung der Blocküberträge innerhalb der B-Zelle sequenzialisiert das Ripple-Carry-Prinzip eines RCLA.

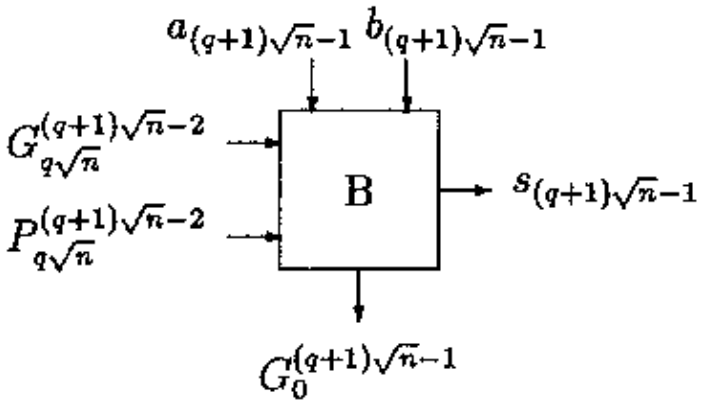
Systolischer Addierer (FASTA): Zelltyp A

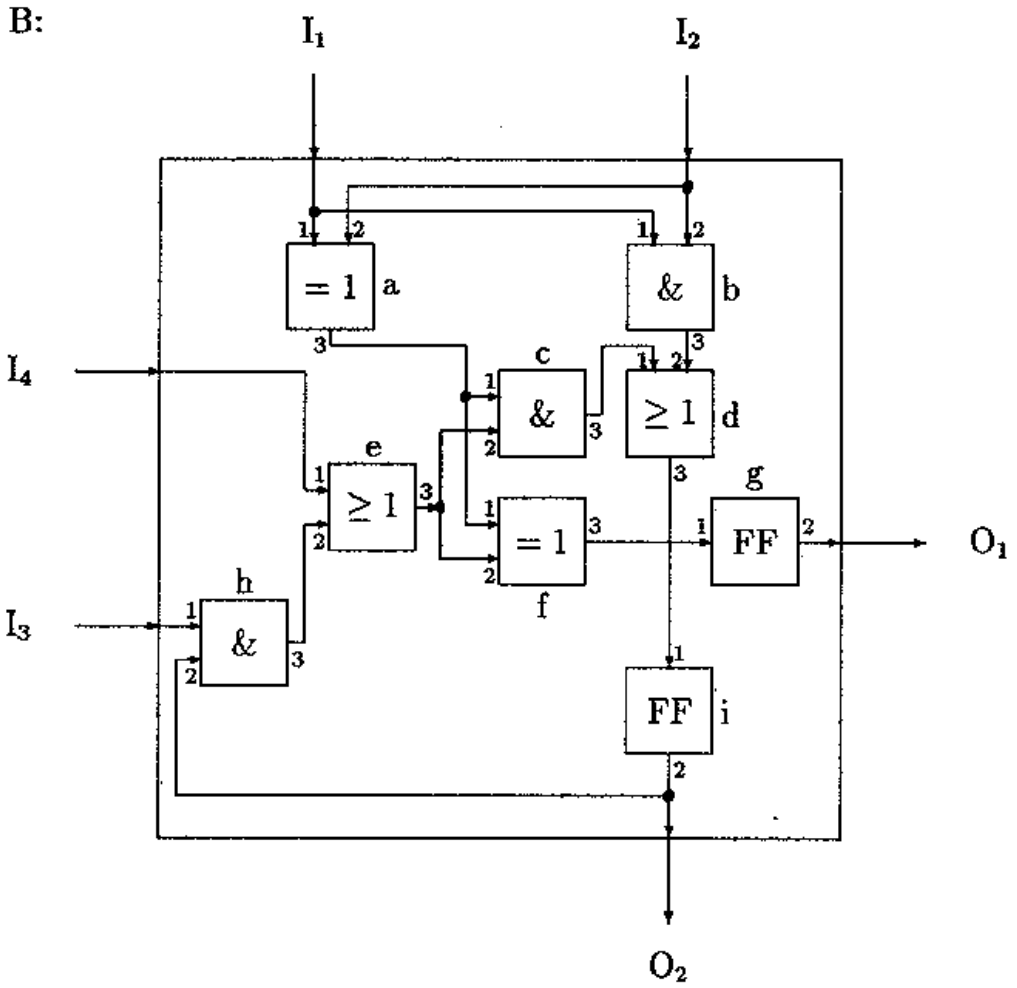


Gestrichelte Teile dienen der Verbesserung der Testbarkeit, im Normalbetrieb ist mit $I_5 = 0$ zu arbeiten.

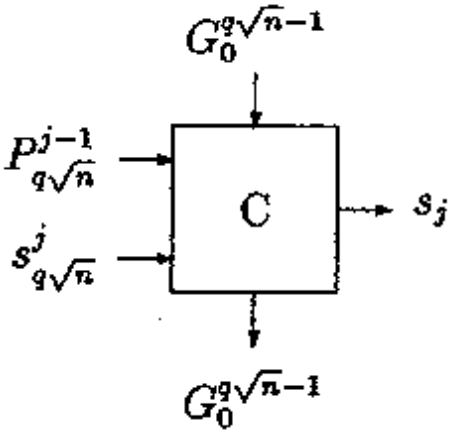


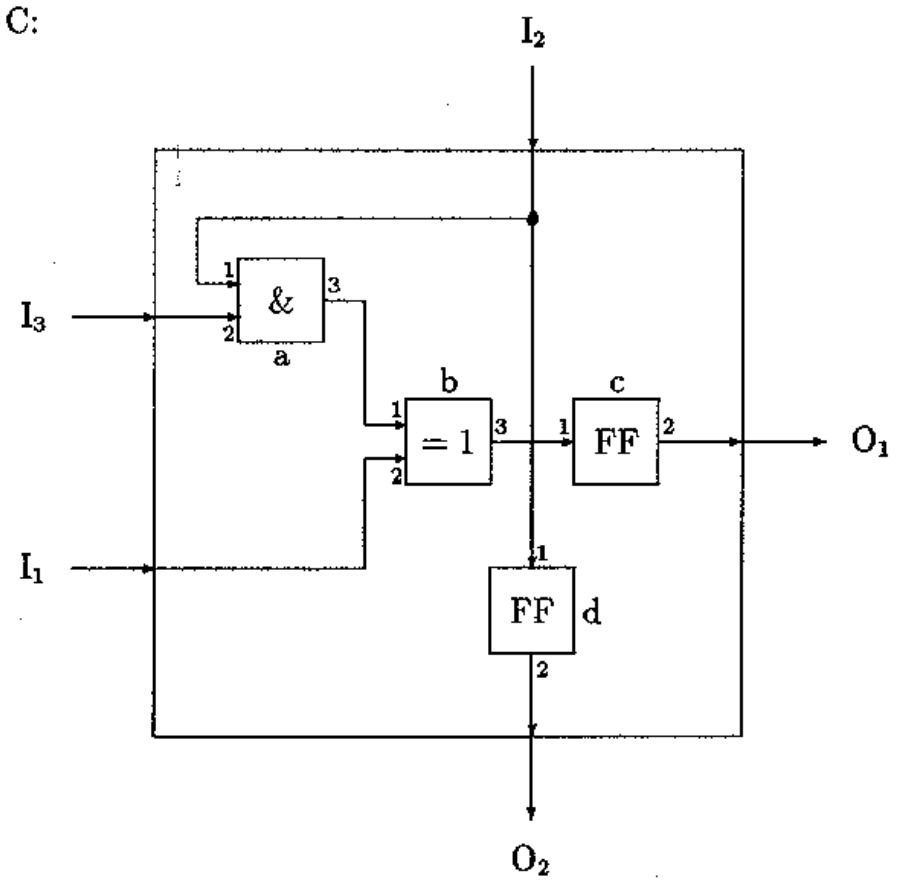
Systolischer Addierer (FASTA): Zelltyp B

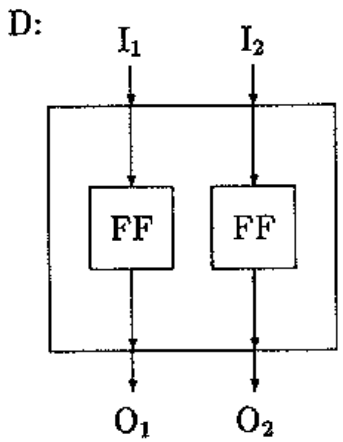




Systolischer Addierer (FASTA): Zelltypen C und D





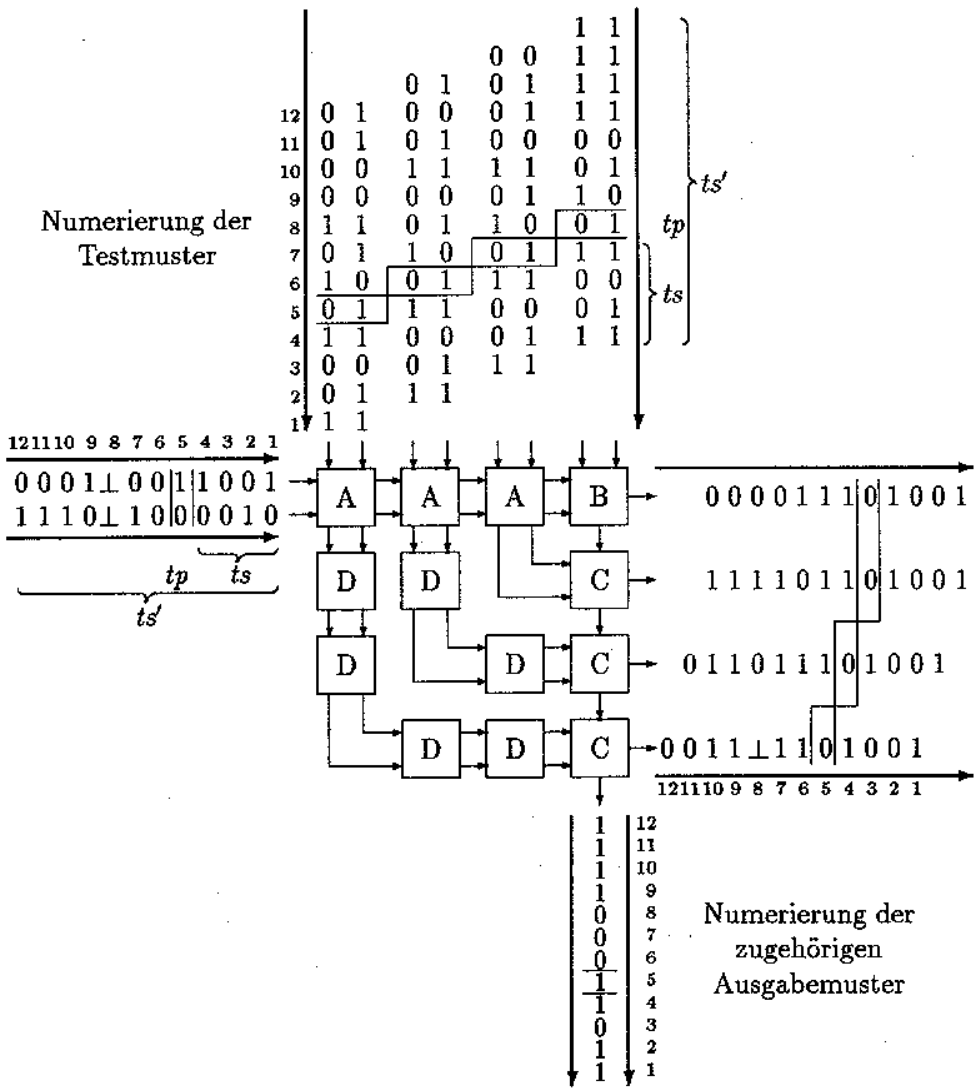


Fehlermodell für synchrone Schaltungen

Im GALEXSA-Fehlermodell (*gate level extended stuck-at fault model*) wird angenommen, dass sich höchstens ein Fehler im Schaltkreis befindet und dieser wie ein einzelner, permanenter Fehler einer der folgenden Kategorien wirkt:

- Haftfehler:
  Ein Ein- oder Ausgang eines Gatters, eines Flipflops oder der gesamten Schaltung liegt fest auf logisch 0 oder 1.
- Übergangsfehler:
  Ein D-Flipflop kann einen der Übergänge $0 \longrightarrow 1$ oder $1 \longrightarrow 0$ nicht vollziehen.
- Verzögerungsfehler:
  Ein D-Flipflop hat keine Verzögerungswirkung mehr, d. h. sein Ausgabewert zur Zeit t ist gleich seinem Eingabewert zur Zeit t.

Testbarkeit von FASTA



Wesentlich: Rückkopplung in der B-Zelle muss durch $l_1 = l_2$ aufgebrochen werden.

Im GALEXSA-Fehlermodell genügen $\sqrt{l} + 8$ Testmuster, um den Test in $3 \times \sqrt{l} + 6$ Takten durchzuführen.

Durch Ergänzung der Schaltung Verbesserung auf $2 \times \sqrt{l} + 9$ Takte mit 11 Testmustern.

Baumartige Carry-Lookahead-Addierer

Sind nur Gatter mit einer kleinen Anzahl von Eingängen verfügbar,
kann der Carry-Lookahead-Addierer baumartig organisiert werden.

Mit einer solchen Organisation gilt (idealisiert):

- Aufwand $\sim l \times \log_2 l$
- Latenz $\sim \log_2 l$

Verknüpfung $\circ$ auf Paaren (G, P) durch $(G, P) \circ (G', P') = (G + PG', PP')$
(hierarchische Berechnung von Block-G/P-Signalen)

Die Verknüpfung $\circ$ ist assoziativ (aber nicht kommutativ),

und es gilt $C_{i+1} = G_{(0:i)} + P_{(0:i)} C_0$ mit

$(G_{(0:i)}, P_{(0:i)}) = (G_i, P_i) \circ (G_{i-1}, P_{i-1}) \circ \cdots \circ (G_1, P_1) \circ (G_0, P_0)$

Wegen der Assoziativität von $\circ$ können Teilprodukte parallel berechnet werden.

Baumartige Carry-Lookahead-Addierer (Beispiel für $i = 7$)

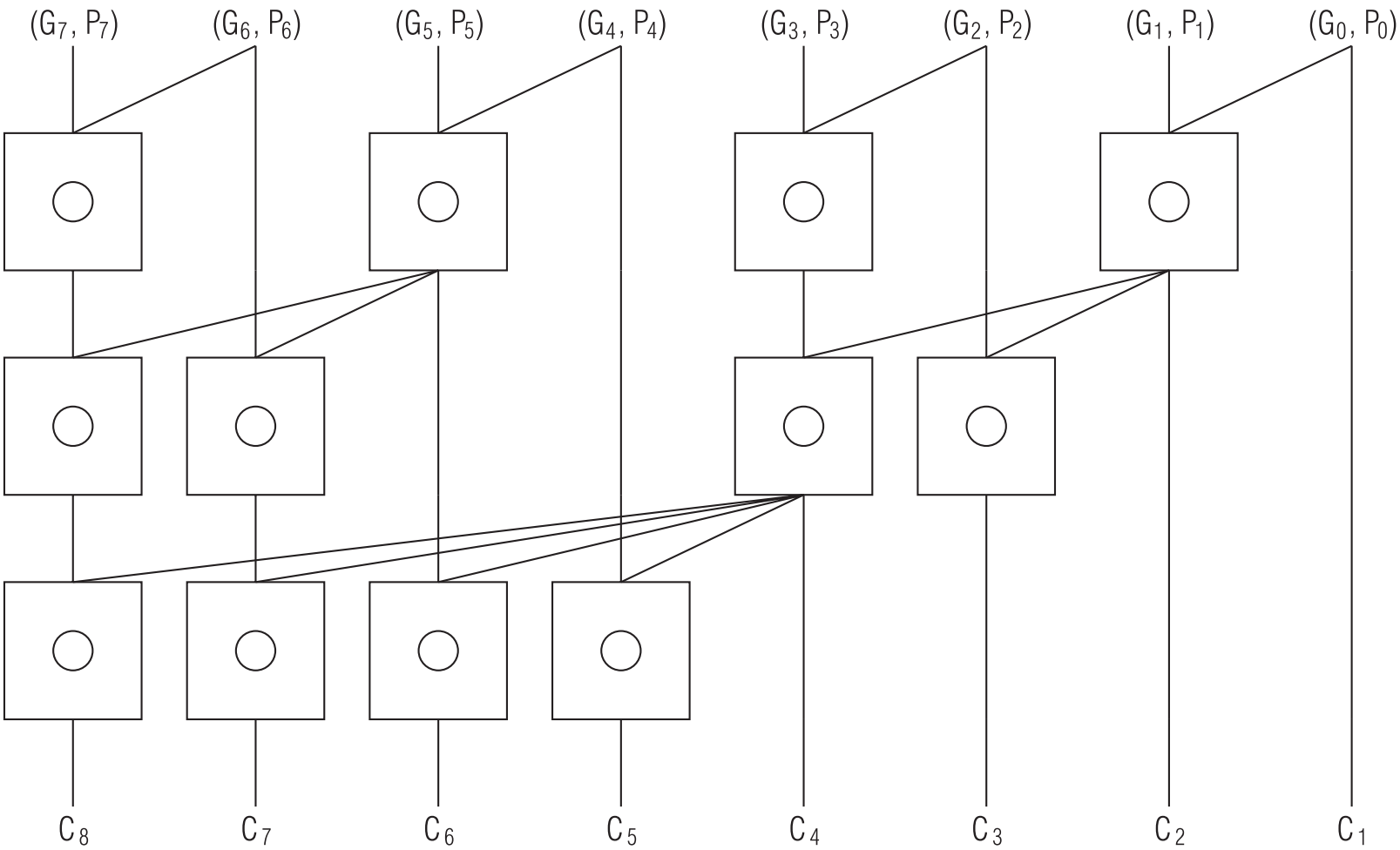
$$
\begin{aligned}
(G_{(0:1)}, P_{(0:1)}) &= (G_1, P_1) \circ (G_0, P_0) \\
(G_{(2:3)}, P_{(2:3)}) &= (G_3, P_3) \circ (G_2, P_2) \\
(G_{(4:5)}, P_{(4:5)}) &= (G_5, P_5) \circ (G_4, P_4) \\
(G_{(6:7)}, P_{(6:7)}) &= (G_7, P_7) \circ (G_6, P_6) \\
(G_{(0:3)}, P_{(0:3)}) &= (G_{(2:3)}, P_{(2:3)}) \circ (G_{(0:1)}, P_{(0:1)}) \\
(G_{(4:7)}, P_{(4:7)}) &= (G_{(6:7)}, P_{(6:7)}) \circ (G_{(4:5)}, P_{(4:5)}) \\
(G_{(0:7)}, P_{(0:7)}) &= (G_{(4:7)}, P_{(4:7)}) \circ (G_{(0:3)}, P_{(0:3)})
\end{aligned}
$$

Die Struktur des Schaltnetzes ergibt sich aus der Klammerstruktur der Ausdrücke zur Berechnung der C_i, z. B. für $(G_{(0:7)}, P_{(0:7)})$ zur Berechnung von $C_8 = G_{(0:7)} + P_{(0:7)} C_0$

$$(((G_7, P_7) \circ (G_6, P_6)) \circ ((G_5, P_5) \circ (G_4, P_4))) \circ (((G_3, P_3) \circ (G_2, P_2)) \circ ((G_1, P_1) \circ (G_0, P_0)))$$

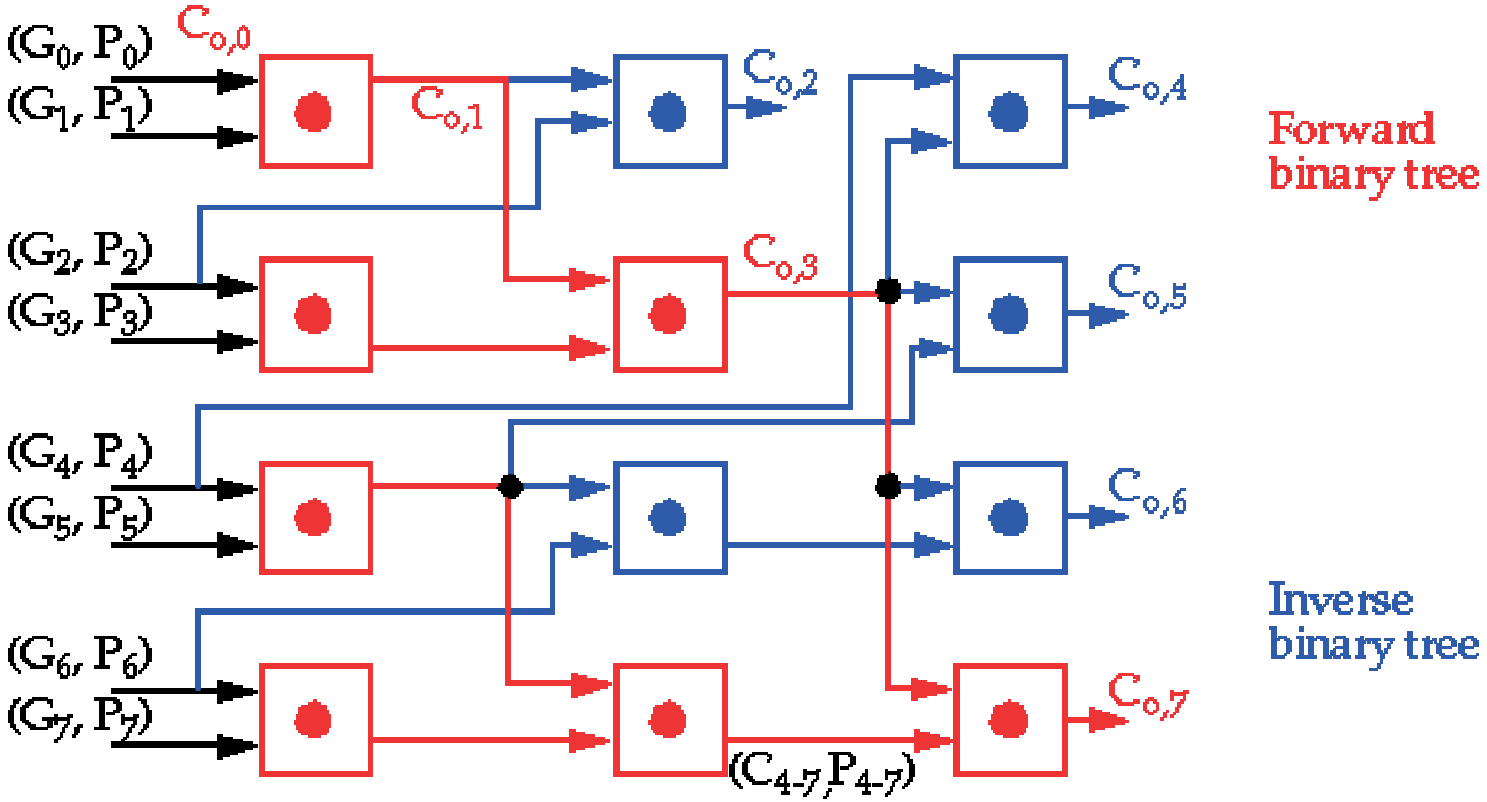
Da wegen der Assoziativität von $\circ$ die Klammerung beliebig gewählt werden kann, sind viele weitere Entwurfsstrategien möglich.

Baumartiger Carry-Lookahead-Generator (CLG) für $l = 8$



Schnellstmögliche Implementierung mit einheitlichen Verknüpfungsmodulen.
Probleme: hoher Aufwand ($l/2 \times \log_2 l$) und hoher Fan-out ($l/2$).

Kompakte Darstellung des baumartigen CLG



The dot operator (●) is defined as: $(g, p) \cdot (g', p') = (g + pg', pp')$

Alternative Schemata für baumartige CLG

Das Fan-out-Problem kann gelöst werden (Kogge-Stone)

- Aufwand $l \times \log_2 l - l + 1$
- Fan-out $\log_2 l$
- Latenz $\log_2 l$

oder (Brent-Kung)

- Aufwand $2 \times l - \log_2 l - 2$
- Fan-out $\log_2 l + 2$
- Latenz $2 \times \log_2 l - 2$

Hybride Formen sind möglich, um Kompromisse auszugestalten, z. B.

- Aufwand $l/2 \times \log_2 l$
- Fan-out $\log_2 l$
- Latenz $\log_2 l + 1$

Präfixberechnungen

Ist D eine endliche Menge, $\circ$ eine assoziative zweistellige Verknüpfung auf D und n eine natürliche Zahl, so heißt eine Abbildung $PRE^n : D^n \to D^n$ *Präfixfunktion*, wenn sie zu jedem n-Tupel $(x_{n-1}, \ldots, x_0) \in D^n$ alle n Anfangsprodukte $x_i \circ \ldots \circ x_0$ für $i = 0, \ldots, n-1$ liefert.

Der Ablauf der Berechnung einer Präfixfunktion kann ohne Kenntnis von D und $\circ$ entwickelt und beschrieben werden.

Die Addition in UInt kann als Spezialfall allgemeiner Präfixberechnungen aufgefasst werden.

Dabei ist D die Menge der Werte des Paars (G, P)
und $\circ$ die durch das Verknüpfungsmodul beschriebene Operation.

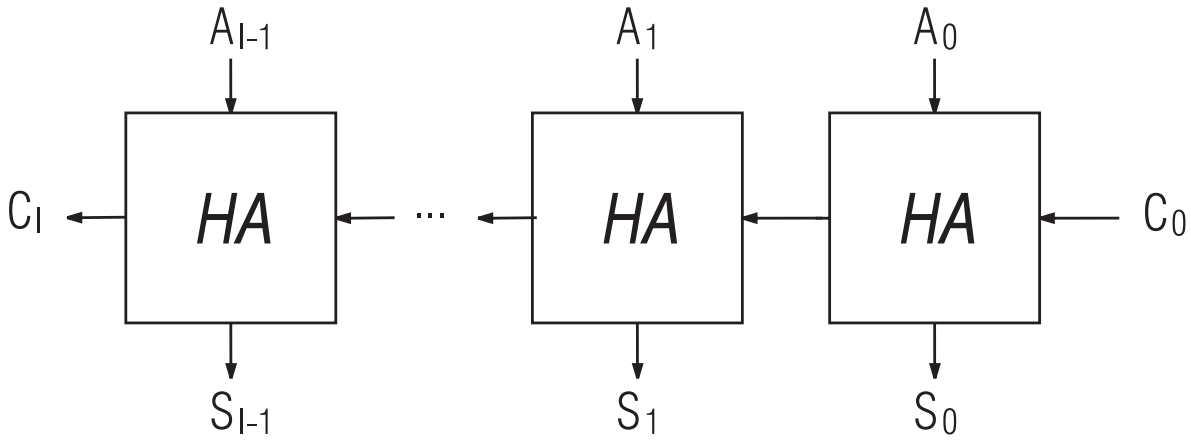
Es gilt $(C_{i+1}, ?) = (G_{(0:i)}, P_{(0:i)}) \circ (C_0, ?)$
bzw. $C_{i+1} = G_{(-1:i)}$ mit $G_{-1} = C_0$ und $P_{-1} = 0$.

Jedes Präfixberechnungsschema berechnet auch alle intermediären Überträge!

Inkrementierer

Häufig besitzt einer der Operanden der Addition nur die Wortlänge 1 Bit —
oder ist sogar konstant gleich Eins (z. B. beim INC-Maschinenbefehl oder in Zählern).

Ein Addierer vereinfacht sich dann zum Inkrementierer:



Schnelle Inkrementierer werden auch als Komponenten von Pyramiden-Addierer benutzt.

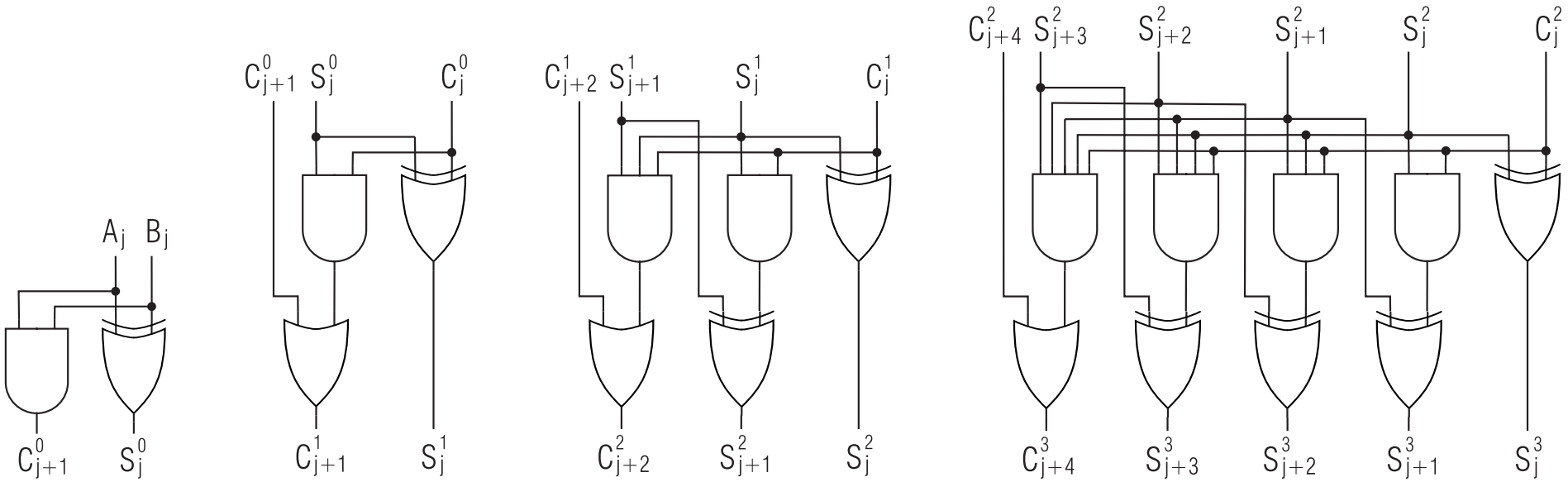
Pyramiden-Addierer

Der Pyramiden-Addierer ist in Stufen aufgeteilt, jede Stufe wiederum in Blöcke.

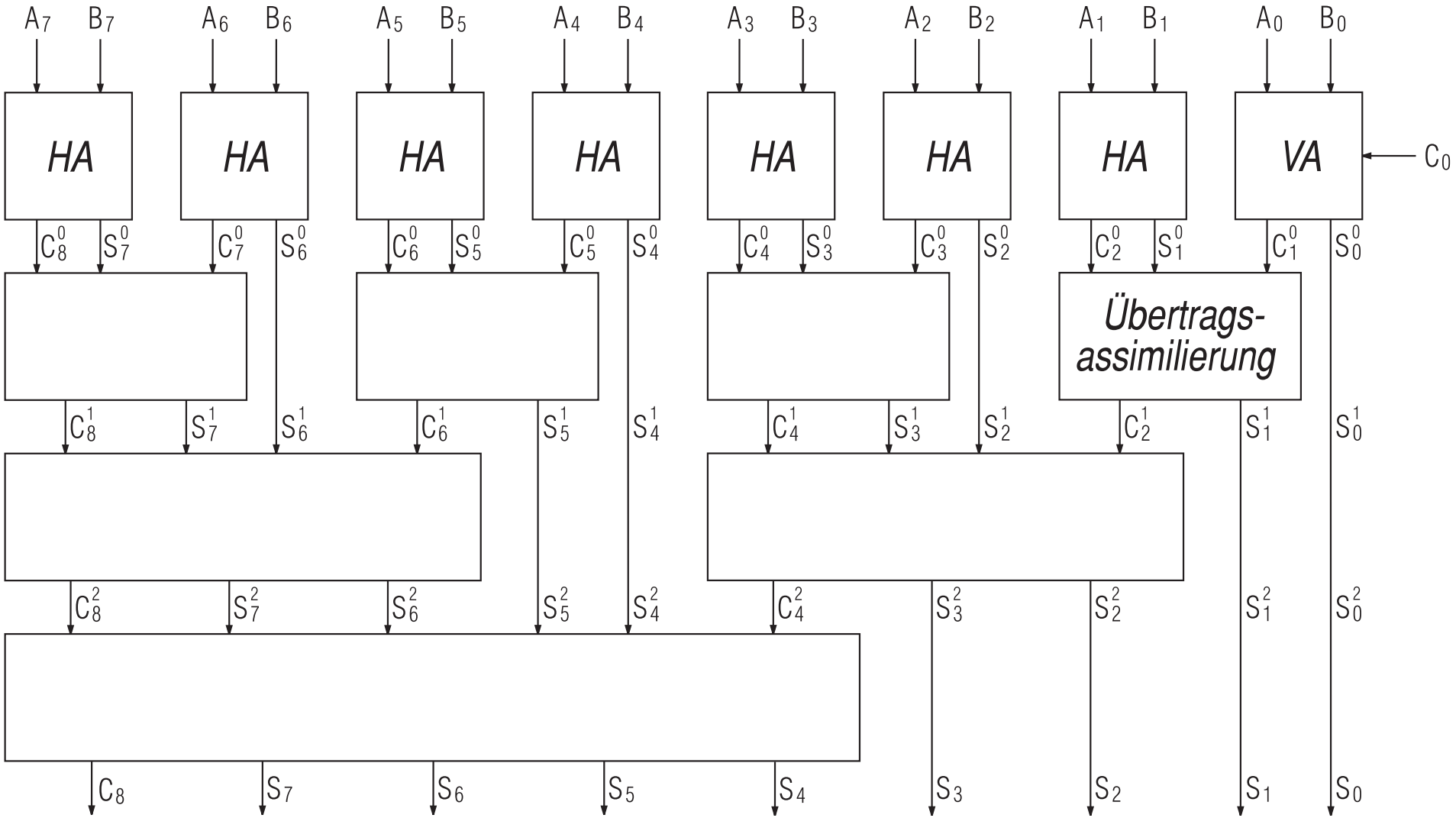
Jeder Block der Stufe 0 ist ein Halbaddierer
(in Bitposition 0 ein Volladdierer, falls zusätzlich ein Eingangsübertrag zu verarbeiten ist).

Jeder Block der Stufe $k \geq 1$ ist ein Inkrementierer (ohne Übertragsausgang)
für Wortlänge $1 + 2^{k-1}$.

Die Inkrementierer müssen mit möglichst geringer Latenz implementiert werden
(z. B. mit Carry-Look-Ahead-Techniken).



Prinzipieller Aufbau des Pyramiden-Addierers



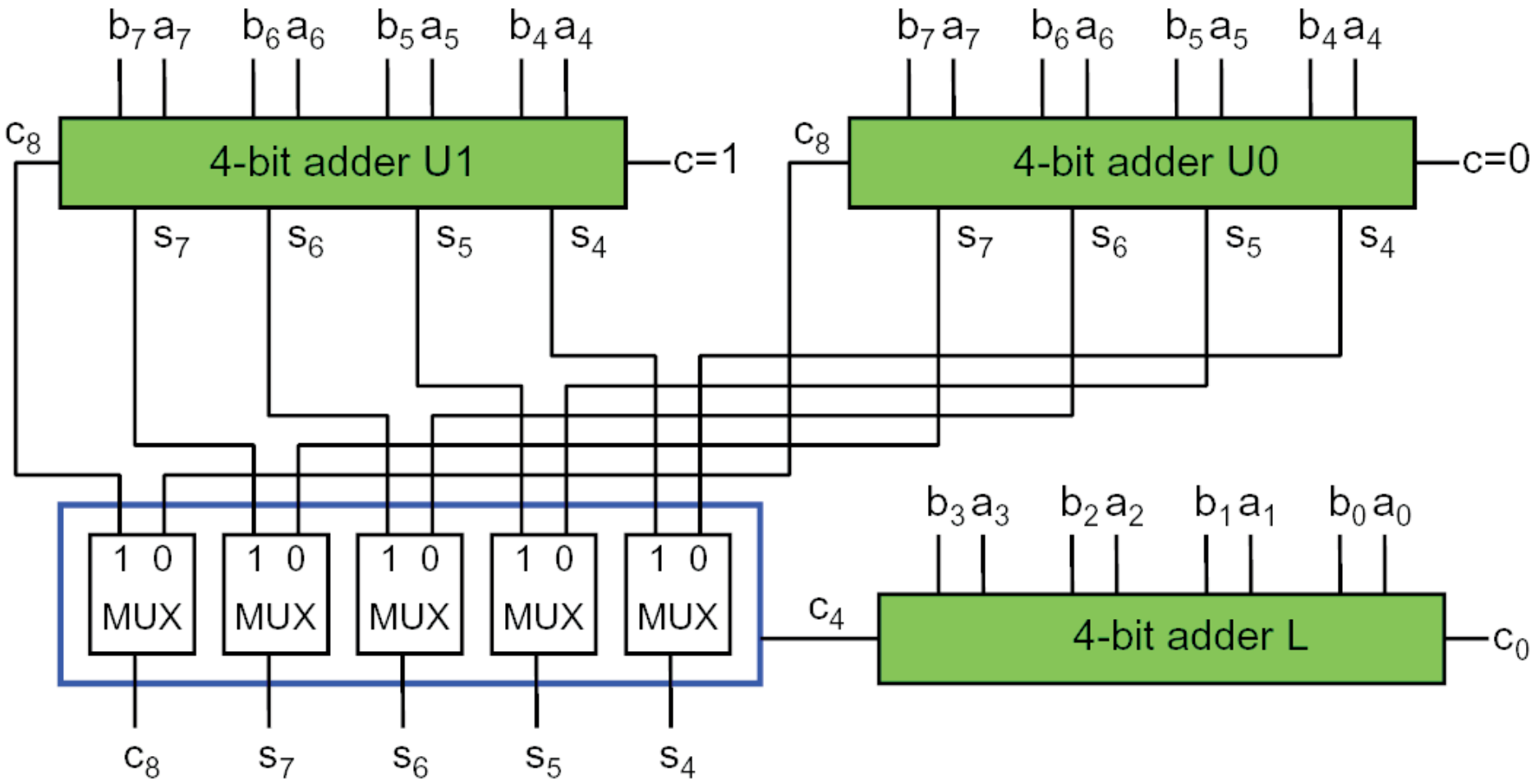
Berechnungsschema des Pyramiden-Addierers

i	8	7	6	5	4	3	2	1	0
A		1	0	0	1	0	1	0	1
B		0	0	1	0	1	1	0	1
S_i^0		1	0	1	1	1	0	0	0
C_i^0	0	0	0	0	0	1	0	1	
S_i^1		1	0	1	1	0	0	1	0
C_i^1	0		0		1		0		
S_i^2		1	0	1	1	0	0	1	0
C_i^2	0				1				
S_i^3		1	1	0	0	0	0	1	0
C_i^3	0								

Wesentliche Eigenschaften des Pyramiden-Addierers

- Latenz $O(\log_2\ l)$
- Aufwand $O(l \times \log_2\ l)$
- Pipelining möglich wegen des strengen Aufbaus in Stufen
- Fan-in der Und-Gatter wächst exponentiell von Stufe zu Stufe
- Fan-out der Ergebnisbits, insbesondere des Übertrags, wächst von Stufe zu Stufe

Das Carry-Select-Schema



Conditional-Sum-Addierer

In $1 + \log_2 l$ Stufen $(0, \ldots, \log_2 l)$ aufgebaut; in Stufe k beträgt die Blocklänge 2^k.

Pro Block werden zwei Ergebnisse berechnet,
jeweils bestehend aus den Summenbits des Blocks
und dem ausgehenden Blockübertrag:

Ein Ergebnis ist gültig unter der Annahme eines eingehenden Blockübertrags,
das andere unter der eines fehlenden Blockübertrags.

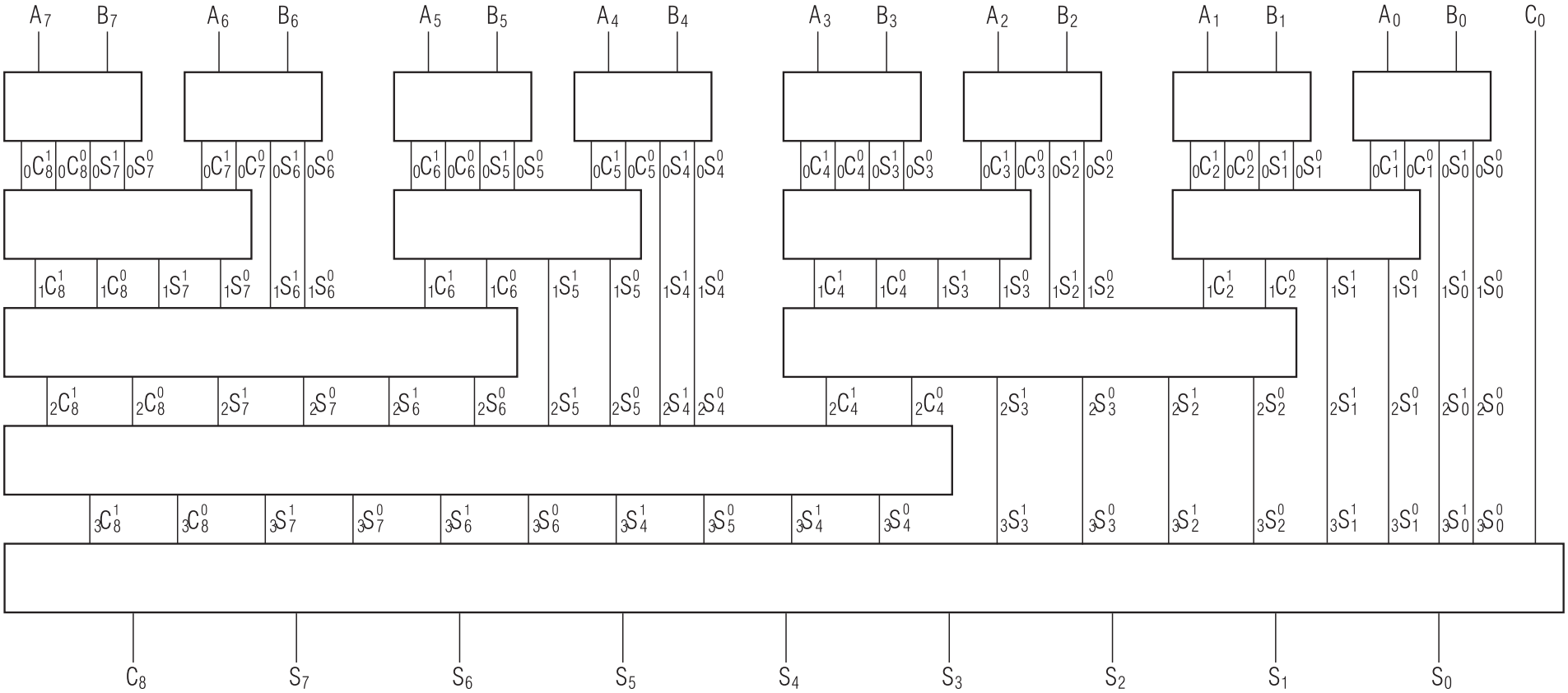
Steht der Wert des eingehenden Blockübertrags schließlich fest,
braucht nur noch das zutreffende Ergebnis ausgewählt werden.

Die Ergebnisse je eines Paars benachbarter Blöcke in Stufe k
werden zum Ergebnis eines Blocks in Stufe $(k + 1)$ zusammengesetzt:

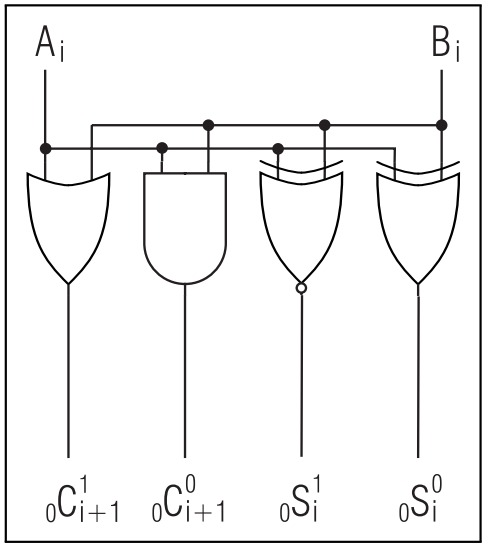
Die Summenteile des niederwertigen Blocks werden direkt weitergeleitet,
die Übertragsbits und die Summenteile des höherwertigen Blocks
durch Selektion mittels der Übertragsbits des niederwertigen Blocks gewonnen.

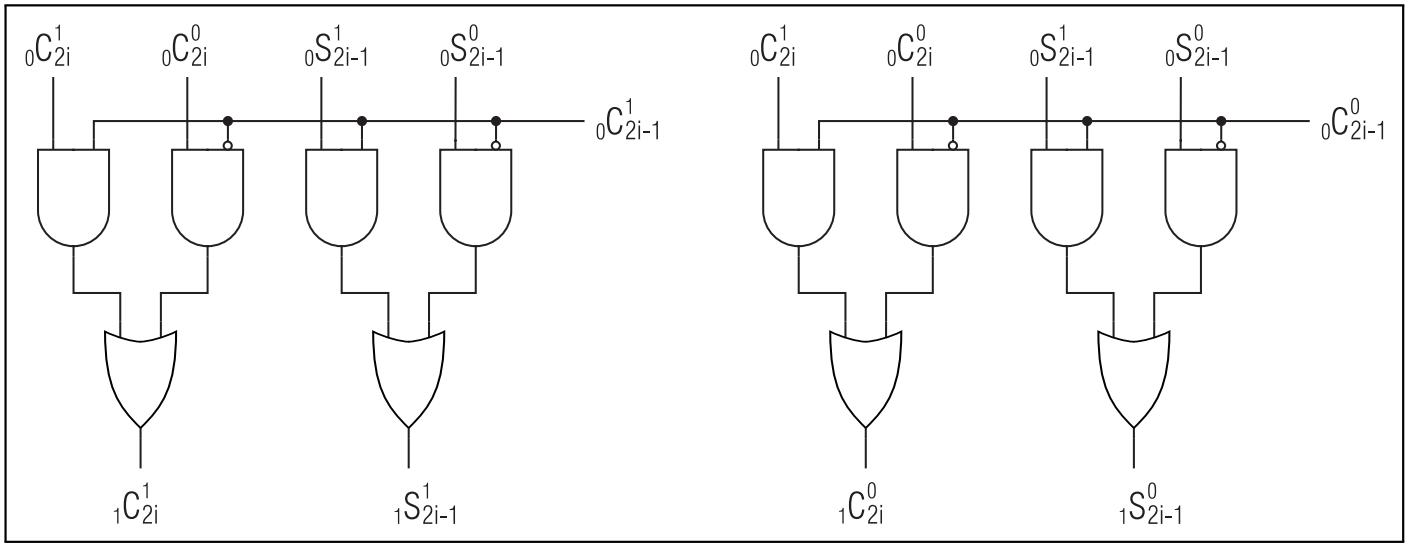
Im Prinzip werden bereits in Stufe 0 alle möglichen Summen- und Übertragsbits berechnet;
alle weiteren Stufen dienen nur der Auswahl aus diesen.

Schematischer Aufbau des Conditional-Sum-Addierers

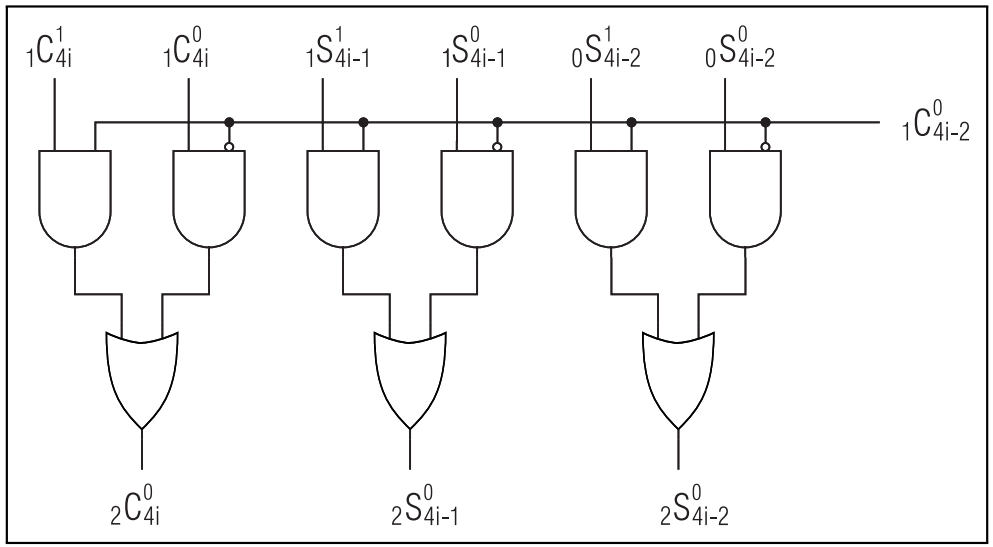


Module in den Stufen 0, 1 und 2 des Conditional-Sum-Addierers

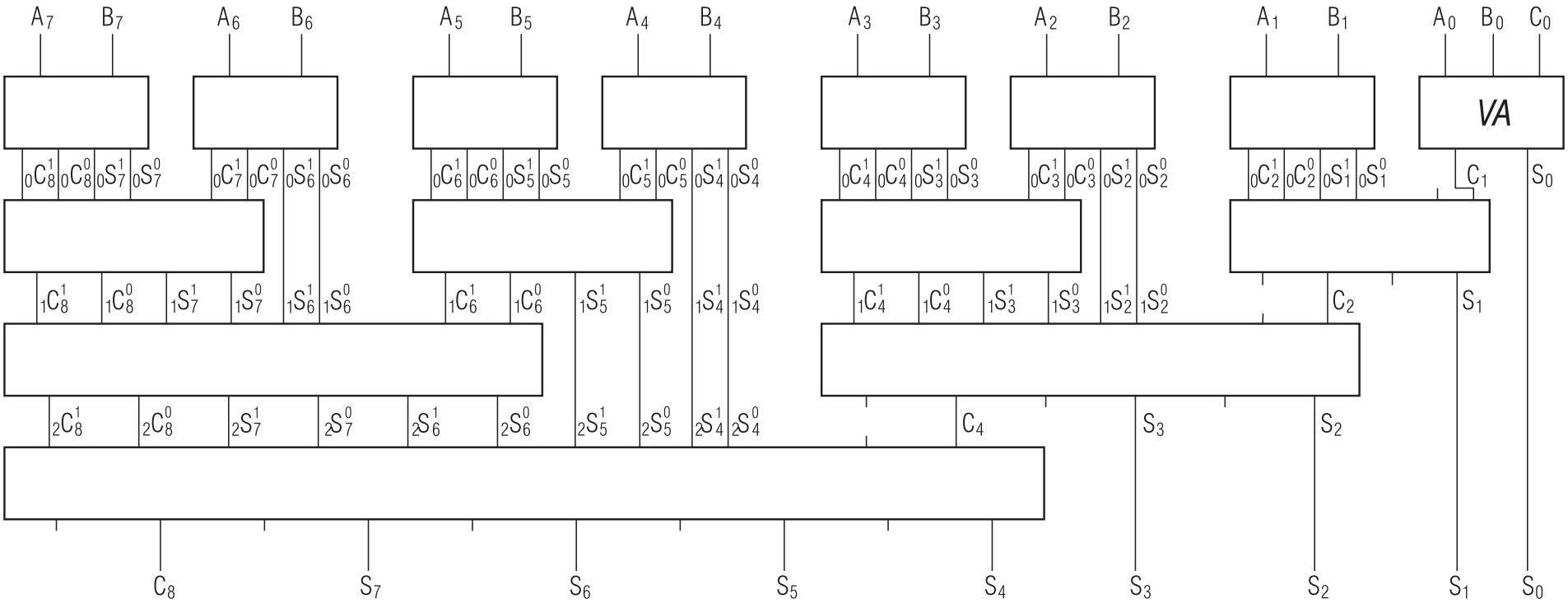




- Gatter mit Fan-in zwei genügen.
- Fan-out des Übertrags wächst exponentiell von Stufe zu Stufe.
- Pipelining gut machbar wegen des strengen Aufbaus in Stufen.
- Latenz $O(\log_2 l)$
- Aufwand $O(l \times \log_2 l)$



Optimierte Variante des Conditional-Sum-Addierers



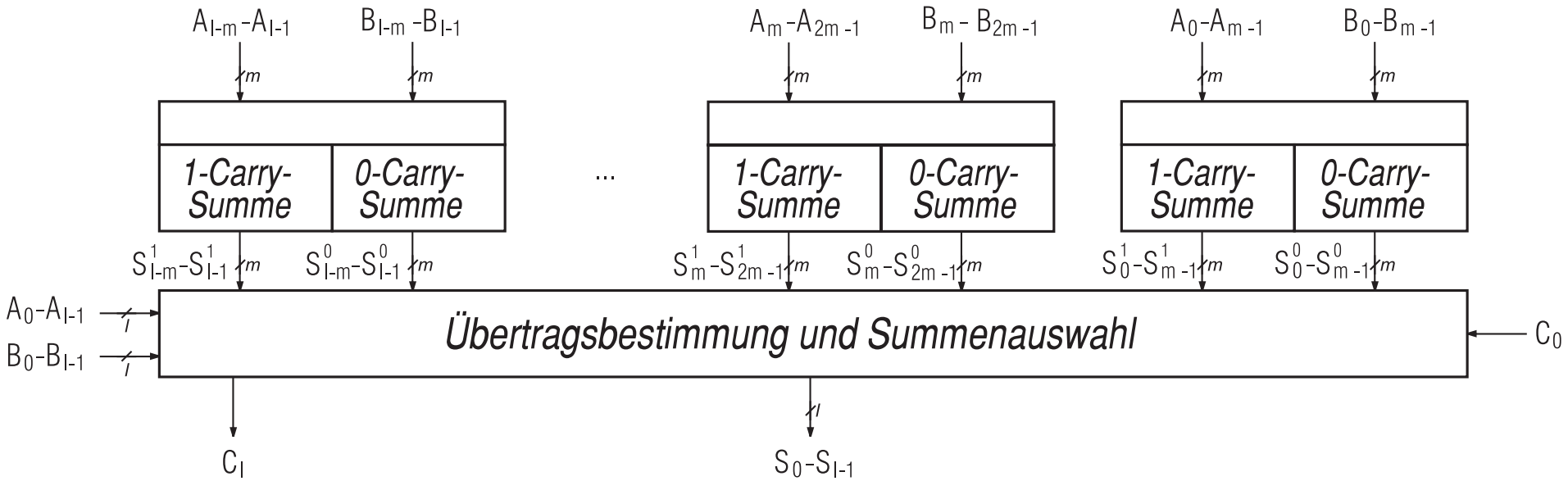
Berechnungsschema im optimierten Conditional-Sum-Addierer

i		7		6		5		4		3		2		1		0	
A		1		0		0		1		0		1		0		1	
B		0		0		1		0		1		1		0		1	
		C	S	C	S	C	S	C	S	C	S	C	S	C	S	C	S
$m = 1$	$C = 0$	0	1	0	0	0	1	0	1	0	1	1	0	0	0	1	0
	$C = 1$	1	0	0	1	1	0	1	0	1	0	1	1	0	1		
$m = 2$	$C = 0$	0	1		0	0	1		1	1	0		0	0	1		0
	$C = 1$	0	1		1	1	0		0	1	0		1				
$m = 4$	$C = 0$	0	1		0		1		1	1	0		0		1		0
	$C = 1$	0	1		1		0		0								
$m = 8$	$C = 0$	0	1		1		0		0		0		0		1		0

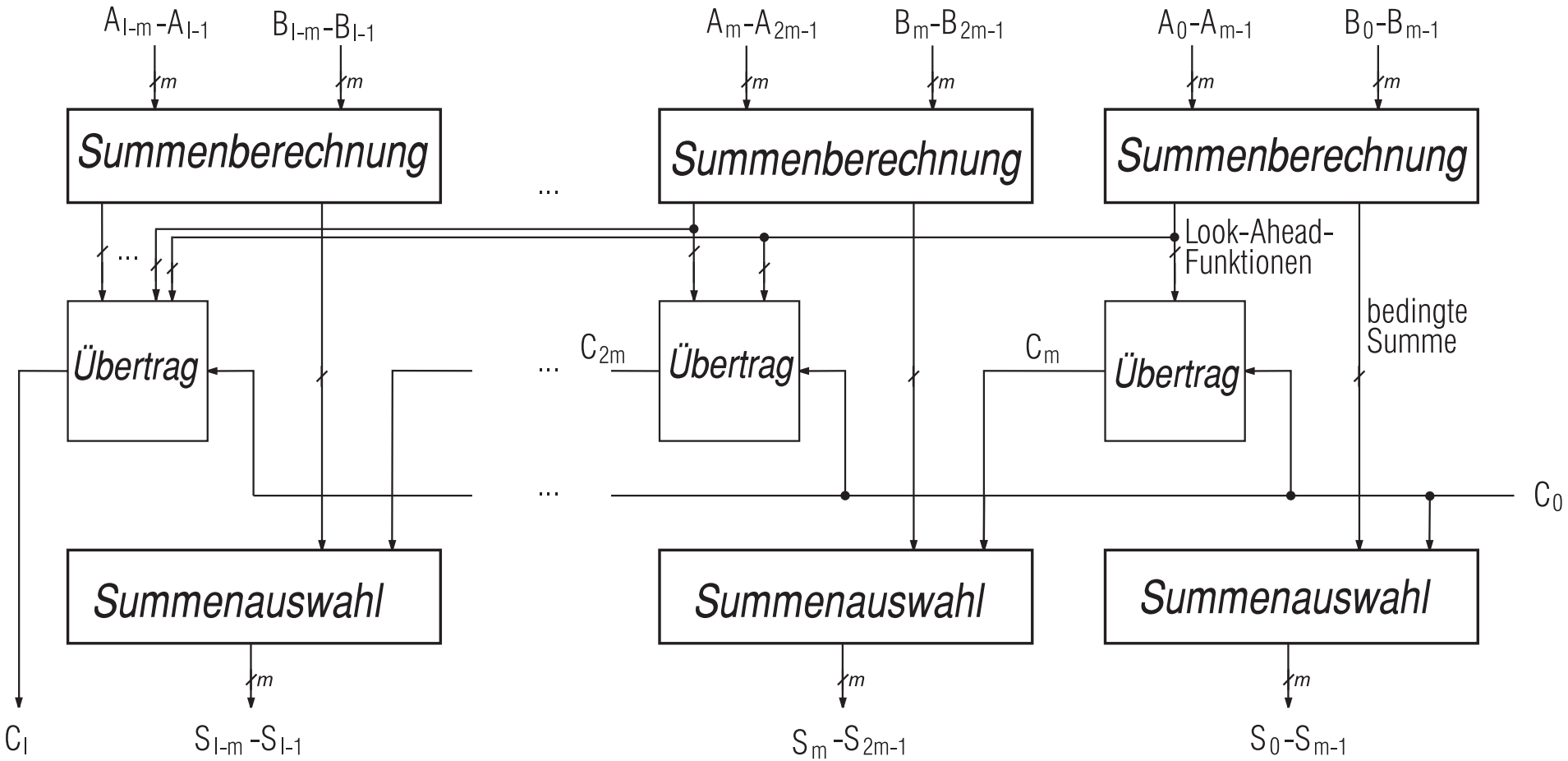
Carry-Select-Addierer

Der Carry-Select-Addierer verallgemeinert den vorgestellten Conditional-Sum-Addierer:

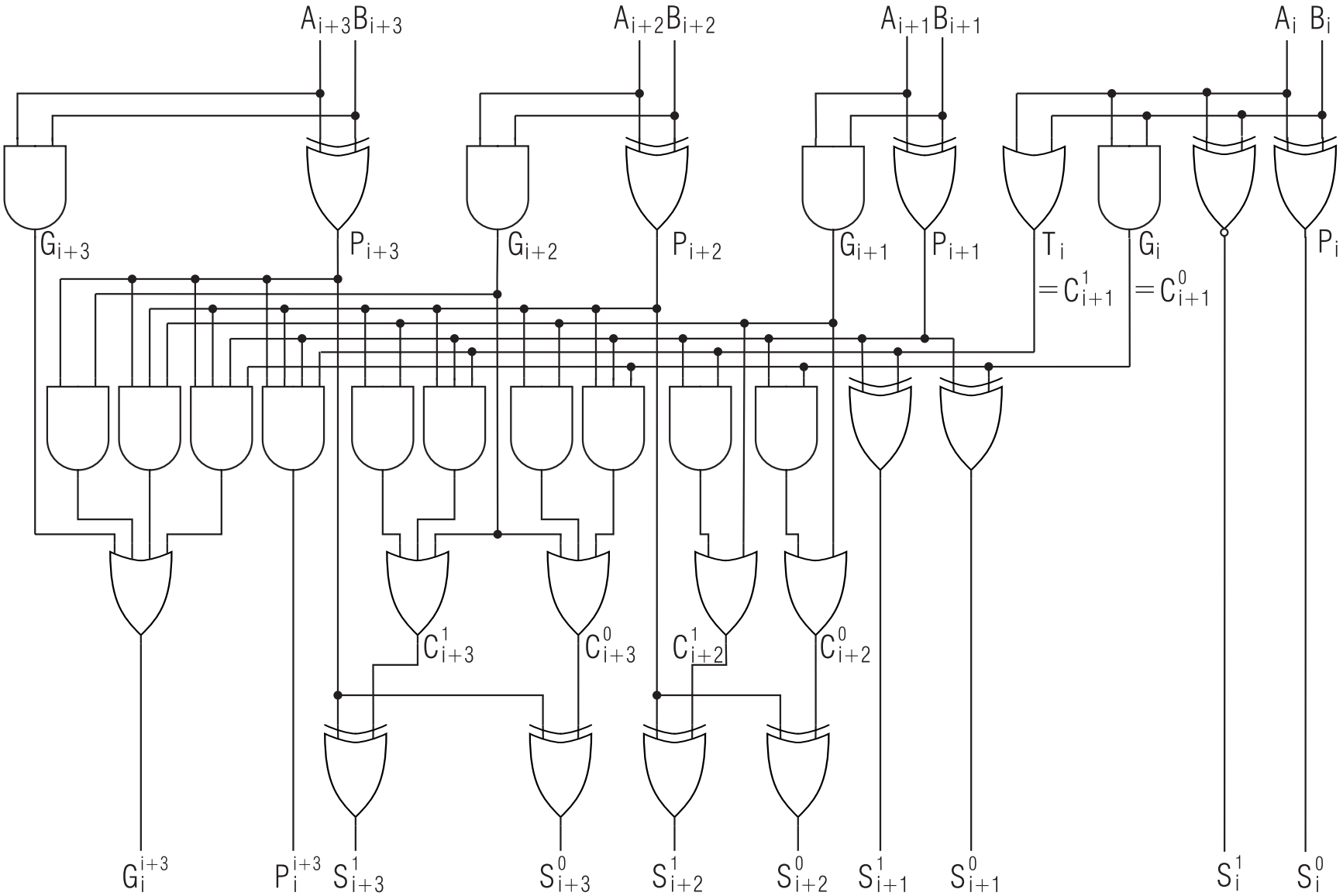
- Jeder Block kann auch mehr als zwei Vorgängerblöcke besitzen.
- Die Vorgängerblöcke können unterschiedliche Länge aufweisen.
- Die Berechnung der Überträge braucht nicht durch Selektion zu erfolgen, sondern kann mit jeder beliebigen Technik durchgeführt werden.



Detailansicht des Carry-Select-Addierers



Summen- und Lookahead-Logik im Carry-Select-Addierer



Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Addition/Subtraktion

Vergleich von Addierern für $\mathrm{UInt}_2(l)$

Typ	Latenz (t)
Einstufiger CLA	4
Einstufiger Carry-Select-Addierer	6
Pyramidenaddierer	$2 \times \log_2 l + 1$
Conditional-Sum-Addierer	$2 \times \log_2 l + 2$
SRCLA ($m = 4, n = 4$)	$l/8 + 6$
SRCLA ($m = 2, n = 8$)	$l/8 + 6$
SRCLA ($m = 4, n = 8$)	$l/16 + 6$
SBCLA ($m = n = \sqrt[3]{l}$)	$6 \times \sqrt[3]{l} - 2$
BCLA ($m = \sqrt{l}$)	$4 \times \sqrt{l} - 2$
Optimaler einstufiger Carry-Skip-Addierer	$\approx 4 \times \sqrt{l}$
RCLA ($m = 4$)	$l/2 + 1$
RCLA ($m = 8$)	$l/4 + 1$
Ripple-Carry-Addierer	$2 \times l - 1$
Serieller Addierer	$3 \times l$

(aus A. R. Omondi: Computer arithmetic systems, 1994, p. 98)

Vergleich von Addierern für $\mathrm{UInt}_2(16)$

Typ	Aufwand (a)	Latenz (t)	$a \times t \times 10^{-4}$
Einstufiger CLA	1264	4	0,51
Einstufiger Carry-Select-Addierer ($m = 4$)	440	6	0,26
RCLA ($m = 8$)	560	6	0,34
Pyramidenaddierer	342	9	0,31
RCLA ($m = 4$)	336	10	0,34
MSBCLA ($m = 2, n = 4$)	360	10	0,36
SRCLA ($m = 2, n = 4$)	392	10	0,39
Conditional-Sum-Addierer	698	10	0,70
Zweistufiger Carry-Skip-Addierer ($m = 2, n = 4$)	300	12	0,36
BCLA ($m = 4$)	300	14	0,42
SBCLA ($m = 2, n = 2$)	368	14	0,52
Einstufiger Carry-Skip-Addierer	240	15	0,36
Ripple-Carry-Addierer	224	31	0,69
Serieller Addierer	22	48	0,11

(aus A. R. Omondi: Computer arithmetic systems, 1994, p. 98)

Vergleich von Addierern für $\mathrm{UInt}_2(32)$

Typ	Aufwand (a)	Latenz (t)	$a \times t \times 10^{-4}$
Einstufiger CLA	7392	4	2,96
Einstufiger Carry-Select-Addierer ($m = 4$)	992	6	0,60
RCLA ($m = 8$)	700	10	1,12
SRCLA ($m = 2, n = 4$)	904	10	0,90
MSBCLA ($m = 2, n = 4$)	944	10	0,94
Pyramidenaddierer	774	11	0,85
Conditional-Sum-Addierer	1594	12	1,91
RCLA ($m = 4$)	672	18	1,21
SBCLA ($m = 2, n = 2$)	748	18	1,35
Zweistufiger Carry-Skip-Addierer ($m = 2, n = 4$)	572	20	1,14
BCLA ($m = 4$)	600	22	1,32
Einstufiger Carry-Skip-Addierer	480	23	1,10
Ripple-Carry-Addierer	448	63	2,82
Serieller Addierer	22	96	0,21

(aus A. R. Omondi: Computer arithmetic systems, 1994, p. 99)

Vergleich von Addierern für $\mathrm{UInt}_2(64)$

Typ	Aufwand (a)	Latenz (t)	$a \times t \times 10^{-4}$
Einstufiger CLA	50624	4	20,25
Einstufiger Carry-Select-Addierer ($m = 4$)	2688	6	1,61
SRCLA ($m = 4, n = 8$)	2032	10	2,03
Pyramidenaddierer	1734	13	2,25
MSBCLA ($m = 4, n = 8$)	1568	14	2,20
SRCLA ($m = 4, n = 4$)	1808	14	2,53
Conditional-Sum-Addierer	3578	14	5,01
MSBCLA ($m = 4, n = 4$)	1344	18	2,42
RCLA ($m = 8$)	1400	18	4,03
SBCLA ($m = 4, n = 4$)	1548	22	3,41
Zweistufiger Carry-Skip-Addierer ($m = 2, n = 4$)	1140	28	3,19
Einstufiger Carry-Skip-Addierer	960	29	2,50
BCLA ($m = 8$)	1320	30	3,96
RCLA ($m = 4$)	1344	34	4,57
BCLA ($m = 4$)	1299	38	4,56
Ripple-Carry-Addierer	896	127	11,38
Serieller Addierer	22	192	0,42

(aus A. R. Omondi: Computer arithmetic systems, 1994, p. 99)

Beurteilung von Übersichten und Kennwerten

Übersichten und Kennwerte sind meist gatterbezogen und geben dann nur eine idealisierte Sichtweise wieder, d. h. sie vernachlässigen z. B.

- Fan-in, Fan-out
- Layout und Leitungslängen
- Gatteraufbau
- Implementierungstechnologie
- Beschleunigungstechniken unterhalb der Gatterebene (Wired-Or, Manchester-Carry-Chain, Transistor-Dimensionierung)

Unterschiedliche Einschätzungen sind nicht ungewöhnlich, zum Vergleich:
Gute Addierer sind nach Einschätzung von O. Spaniol (1981)

- SRCLA
- Carry-Skip-Addierer mit variabler Blocklänge
- Conditional-Sum-Addierer
- Carry-Select-Addierer

Serienaddierer

Charakteristika

- Langsam
- Billigster Addierer
- Gutes Preis/Leistungsverhältnis

Sollte verwendet werden, wenn

- Geschwindigkeit unkritisch ist
- einfaches Routing erwünscht ist
- Preis oder Flächenbedarf klein gehalten werden müssen
- Parallelität ins Spiel kommt, aber Fläche bzw. Kosten beschränkt sind

Maximaler Durchsatz bei beschränkter Fläche bzw. Kosten:
Viele parallel zueinander betriebene langsame Addierer können höheren Durchsatz produzieren als wenige schnelle Addierer (deshalb beliebt in SIMD-Maschinen).

Beispiel aus der Tabelle für 16-Bit-Addierer:
8 Serienaddierer erreichen denselben Durchsatz wie ein Carry-Select-Addierer,
benötigen aber nur 40% der Fläche.

Weiterführende Literatur
P. Denyer, D. Renshaw: VLSI signal processing: a bit-serial approach.
S.G. Smith, P. Denyer: Serial-data computation.

Ripple-Carry-Addierer und Carry-Skip-Addierer

Ripple-Carry-Addierer

Nachteile:
Hohe Latenz.
Schlechtes Kosten/Nutzen-Verhältnis.

Vorteile:
Einfache, reguläre Struktur, daher gut für VLSI-Implementierung geeignet.

Dadurch werden Nachteile eventuell ausgeglichen.

Mit Manchester-Carry-Chain in vielen Fällen ausreichend schnell und relativ kostengünstig.

Carry-Skip-Addierer

Relativ geringe Zusatzkosten.

Geringste Latenz bei variierender Blocklänge.
Wegen geringerer Regularität aber eventuell schwierig zu implementieren.

Einheitliche Blocklänge mit Manchester-Carry-Chain ist daher meist günstiger.

Carry-Lookahead-Addierer

Für kleine Wortlängen CLA der schnellste Addierer, aber teuer.
Für größere Wortlängen wegen hohem Fan-in und Fan-out meist nicht direkt implementierbar.
Abhilfe: Stufenbildung und Kombination mit anderen Prinzipien (Ripple-Carry, Carry-Skip).

RCLA (CLA in Stufe 0, Ripple-Carry in Stufe 1):

Reguläre Struktur, geringe Verbindungsdichte zwischen Blöcken.
Gut geeignet für mittelgroße Wortlängen.

SRCLA (CLA in Stufe 0, CLA in Stufe 1, Ripple-Carry in Stufe 2):

Extrem schnell für große Wortlängen.
Geringere Regularität, höhere Verbindungsdichte.

Pyramiden-, Carry-Select- und Conditional-Sum-Addierer

Pyramiden-Addierer:

- Geringe Latenz, gutes Kosten/Nutzen-Verhältnis.
- Wegen hohem Fan-in und Fan-out nur für kleine Wortlängen implementierbar.
- In Kombination mit anderen Prinzipien evtl. gute Leistung ohne Fan-in-/Fan-out-Probleme.

Carry-Select-Addierer:

- Relativ schnell.
- Hoher Fan-in.
- Hoher Fan-out.
- Gutes Kosten/Nutzen-Verhältnis, aber aufwendig.

Conditional-Sum-Addierer:

- Relativ schnell.
- Hoher Fan-out.
- Schlechtes Kosten/Nutzen-Verhältnis.
- Gute Effizienz in Einsatzumgebungen, die beide bedingten Summen gleichzeitig benötigen, z. B. in schnellen Gleitkomma-Einheiten.

Pipelining von Addierern

Gut für Pipelining geeignet sind (in Klammern der maximale Pipeliningfaktor):

- Baumartige CLA ($1 + \log_2 l$)
- Pyramiden-Addierer ($1 + \log_2 l$)
- Conditional-Sum-Addierer ($1 + \log_2 l$)
- Carry-Select-Addierer (Stufenzahl, i. A. weniger als $\log_2 l$)
- Ripple-Carry-Addierer mit versetzter Dateneingabe (l)

Subtrahierer

Ein Subtrahierer nach dem Ripple-Prinzip entsteht aus einem Ripple-Carry-Addierer durch Ersetzen jeder Volladdiererzelle durch eine Subtraktionszelle.

Statt eines Übertrags (Carry) wird von Zelle zu Zelle (von der niederwertigsten zur höchstwertigen Stelle) ein Borge-Signal (Borrow) weitergereicht.

Trägt das höchstwertige ausgehende Borge-Signal C_l den Wert 1,
liegt ein Überlauf vor, d. h. es gilt $A - B < 0$.

Subtraktionszelle

Eingänge:

- Minuend (positives Vorzeichen)
- Subtrahend (negatives Vorzeichen)
- Borge-Signal von der Vorgängerstelle niedrigerer Wertigkeit (Borrow-in, negatives Vorzeichen)

Ausgänge:

- Differenzbit (positives Vorzeichen)
- Borge-Signal zur Nachfolgerstelle höherer Wertigkeit (Borrow-out, negatives Vorzeichen)

Die Vorzeichen sind nur fiktiv (werden nicht tatsächlich codiert).

$A_i - B_i - C_i = D_i - 2 \times C_{i+1}$				
A_i	B_i	C_i	D_i	C_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

$$
\begin{aligned}
D_i &= A_i \oplus B_i \oplus C_i \\
C_{i+1} &= (B_i \oplus C_i)\overline{A_i} + B_i C_i
\end{aligned}
$$

Vorzeichenwechsel

Vorzeichen/Betrag-Darstellung:

- Invertieren des Vorzeichen-Bits.

1-Komplement-Darstellung:

- Invertieren aller Bits der Darstellung.

2-Komplement-Darstellung:

- Invertieren aller Bits und inkrementieren ohne Berücksichtigung des Übertrags.
- Benutzt wird dabei die Beziehung $-[B]_2 = [\bar{B}]_2 + 1$.
- Für $[B]_2 = -2^{l-1}$ führt die Inkrementoperation zu einem Überlauf.

Vorzeichenwechsel in $\mathrm{Int}_2(l)$

Mit $B = B_{l-1}B_{l-2}\ldots B_0$, $\bar{B} = \bar{B}_{l-1}\bar{B}_{l-2}\ldots\bar{B}_0$, $\bar{B}_i = 1 - B_i$ und

$$[B]_2 = -B_{l-1} \times 2^{l-1} + \sum_{i=0}^{l-2} B_i \times 2^i$$

gilt

$$\begin{aligned} -[B]_2 &= B_{l-1} \times 2^{l-1} + \sum_{i=0}^{l-2} -B_i \times 2^i \\ &= -(1 - B_{l-1}) \times 2^{l-1} + \sum_{i=0}^{l-2}(1 - B_i) \times 2^i + 2^{l-1} - \sum_{i=0}^{l-2} 2^i \\ &= -\bar{B}_{l-1} \times 2^{l-1} + \sum_{i=0}^{l-2} \bar{B}_i \times 2^i + 1 = [\bar{B}]_2 + 1 \end{aligned}$$

Längenkonforme Konversion zwischen Zahlensystemen

$\mathrm{conv}_{l,l}(A_{l-1}, \ldots, A_0) = (B_{l-1}, \ldots, B_0)$

$\mathrm{Int}_1 \rightarrow \mathrm{Int}_2$: $B = A \oplus_l A_{l-1}$

$\mathrm{Int}_2 \rightarrow \mathrm{Int}_1$: $B = A \ominus_l A_{l-1}$ (Fehler für $10\ldots0$)

$\mathrm{Int}_{VB} \rightarrow \mathrm{Int}_1$: $B = (A_{l-1}, (A_{l-1} \oplus A_{l-2}), \ldots, (A_{l-1} \oplus A_0))$

$\mathrm{Int}_1 \rightarrow \mathrm{Int}_{VB}$: $B = (A_{l-1}, (A_{l-1} \oplus A_{l-2}), \ldots, (A_{l-1} \oplus A_0))$

$\mathrm{Int}_{VB} \rightarrow \mathrm{Int}_2$: $B = (A_{l-1}, (A_{l-1} \oplus A_{l-2}), \ldots, (A_{l-1} \oplus A_0)) \oplus_l A_{l-1}$

$\mathrm{Int}_2 \rightarrow \mathrm{Int}_{VB}$: $H = A \ominus_l A_{l-1}$; $B = (A_{l-1}, (H_{l-1} \oplus H_{l-2}), \ldots, (H_{l-1} \oplus H_0))$
(Fehler für $10\ldots0$)

Konversion innerhalb eines Zahlensystems

$\mathrm{conv}_{l',l}(A_{l'-1},\ldots,A_0) = (B_{l-1},\ldots,B_0)$

- In UInt:
  - $l' < l$: $\quad \forall i < l' : B_i = A_i \quad \forall i \geq l' : B_i = 0$
  - $l' > l$: $\quad \forall i < l : B_i = A_i \quad$ (nur gültig, falls $\forall i \geq l : A_i = 0$)
- In Int_{VB}:
  - $l' < l$: $\quad \forall i \leq l' - 2 : B_i = A_i \quad \forall i, l' - 1 \leq i \leq l - 2 : B_i = 0 \quad B_{l-1} = A_{l'-1}$
  - $l' > l$: $\quad \forall i \leq l - 2 : B_i = A_i \quad B_{l-1} = A_{l'-1}$
    (nur gültig, falls $\forall i, l - 1 \leq i \leq l' - 2 : A_i = 0$)
- In Int_2 und Int_1:
  - $l' < l$: $\quad \forall i < l' : B_i = A_i \quad \forall i \geq l' : B_i = A_{l'-1}$
  - $l' > l$: $\quad \forall i < l : B_i = A_i \quad$ (nur gültig, falls $\forall i \geq l : A_i = A_{l-1}$)

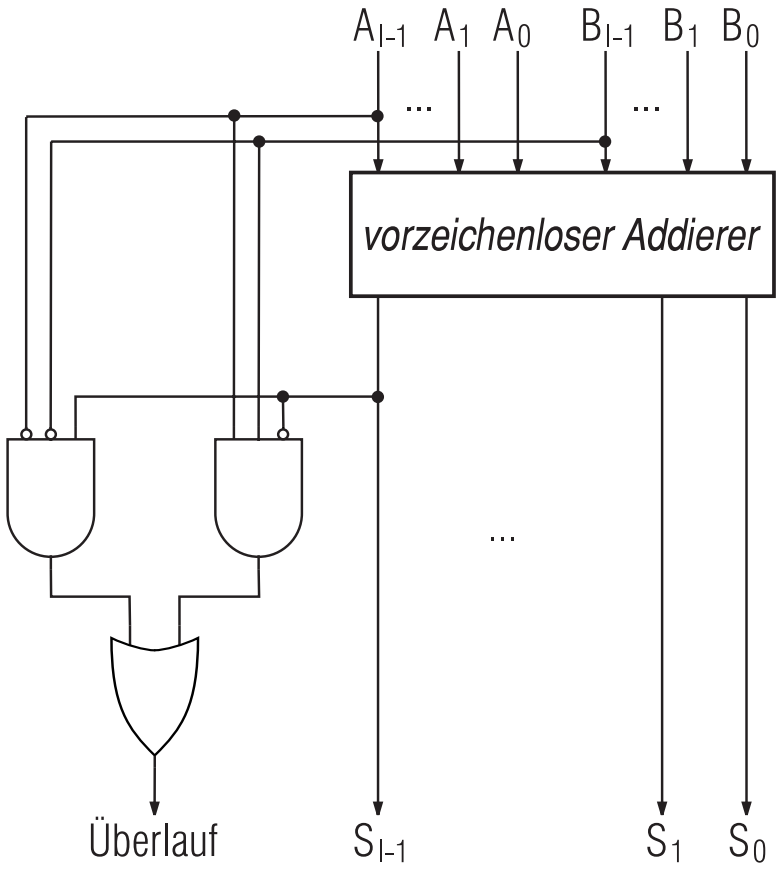
Addierer für $\mathrm{Int}_2(l)$

Zahlen in 2-Komplement-Darstellung können mit Verfahren für vorzeichenlose ganze Zahlen addiert werden, wenn modulo 2^l gerechnet wird:

$$(C_l, S_{l-1}, \ldots, S_0) = (A_{l-1}, A_{l-2}, \ldots, A_0) + (B_{l-1}, B_{l-2}, \ldots, B_0)$$

$A_{l-1} = B_{l-1} \neq S_{l-1}$ zeigt Überlauf an, die Ziffer C_l ist irrelevant.

Nachteilig an dieser Implementierung ist, dass die Überlaufinformation zugleich aus den beiden Operanden und dem Ergebnis abgeleitet werden muss.



Schutzstellentechnik für Addition in $\mathrm{Int}_2(l)$

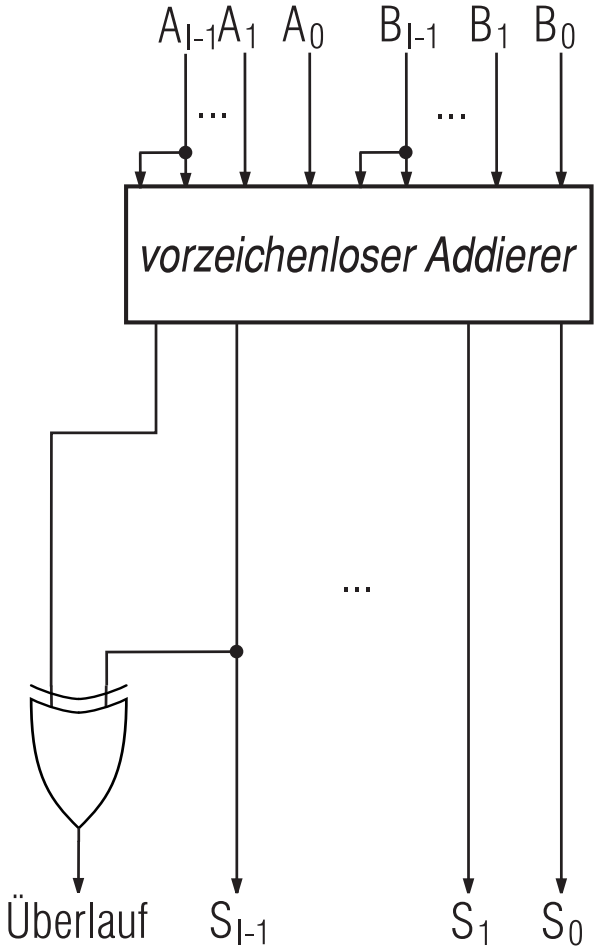
Die Operanden und das Ergebnis können um je eine Stelle erweitert werden:

$$(C_{l+1}, S_l, \ldots, S_0) = (A_{l-1}, A_{l-1}, \ldots, A_0) + (B_{l-1}, B_{l-1}, \ldots, B_0)$$

Überlauf ist dann an $S_l \neq S_{l-1}$ zu erkennen, die Ziffer C_{l+1} ist irrelevant.

Bei dieser Implementierung ergibt sich die Überlaufinformation ausschließlich aus dem Ergebnis.

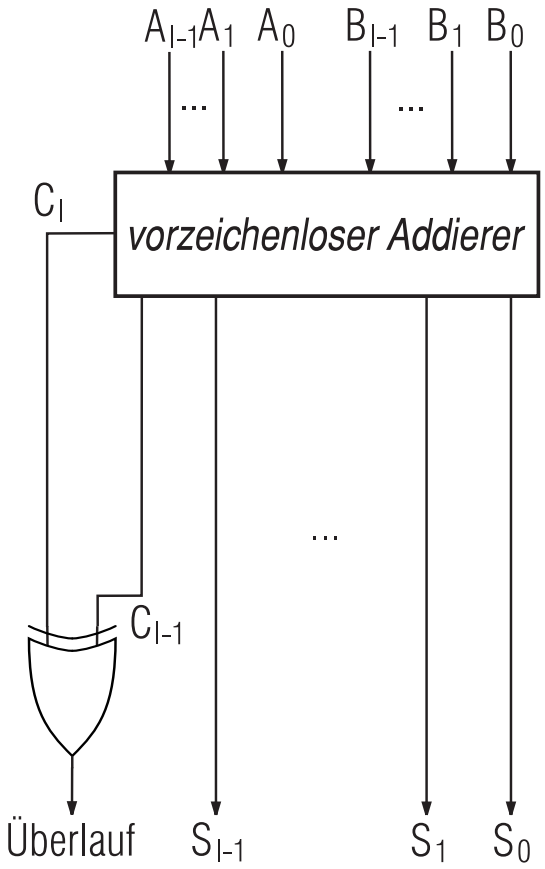
Allerdings muss der Addierer um eine Stelle verbreitert werden.



Überlauferkennung für Addition in $\mathrm{Int}_2(l)$ durch intermediäre Überträge

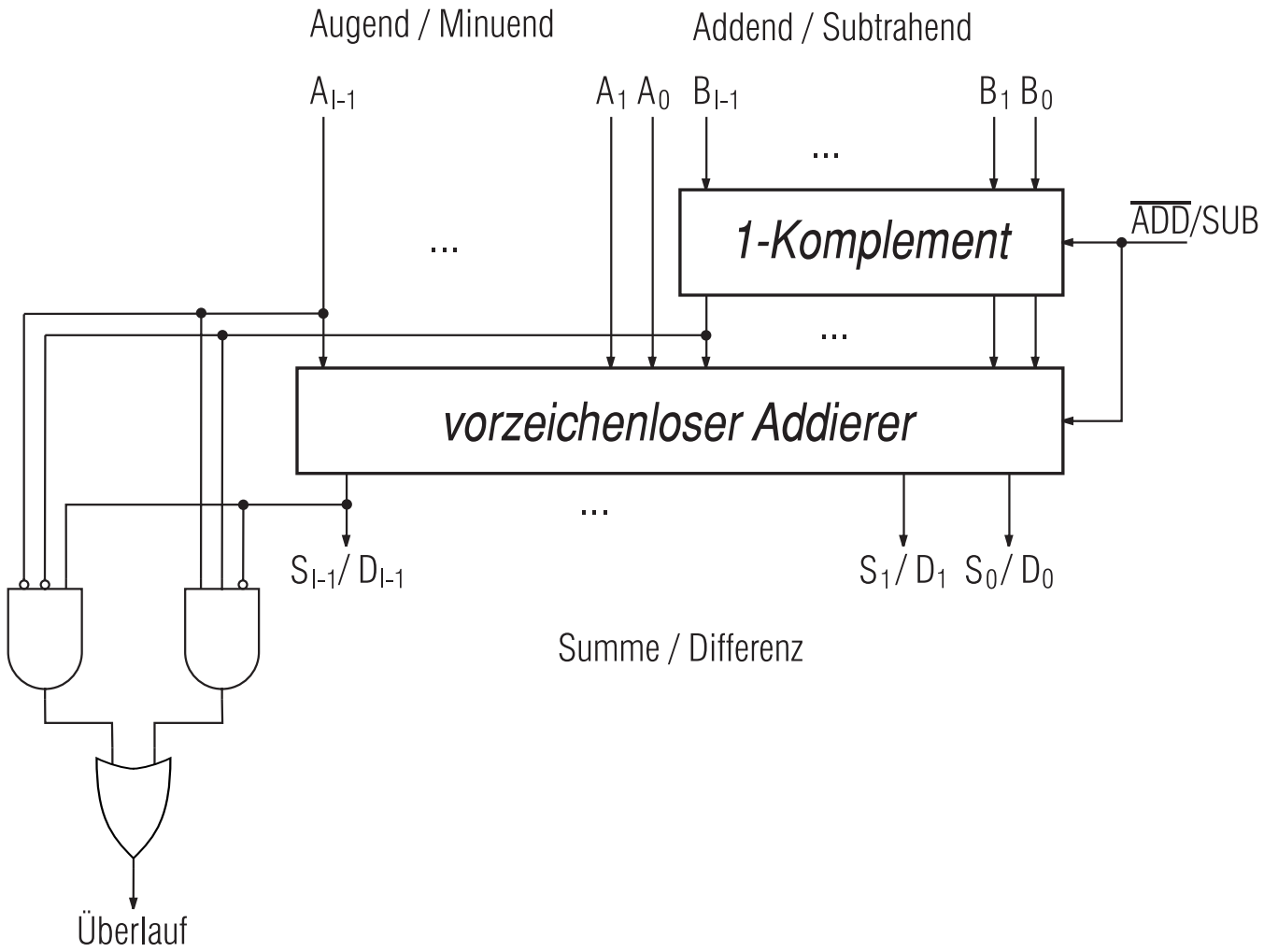
Als Alternative zur Schutzstellentechnik kann die Überlaufinformation ohne Verbreiterung des Addierers nur aus dem Ergebnis gewonnen werden, wenn der Übertrag C_{l-1} zugänglich ist: $C_{l-1} \neq C_l$ zeigt hier den Überlauf an.

Diese Lösung scheidet jedoch aus, falls das Signal C_{l-1} außerhalb des Addierers nicht verfügbar ist.



Kombinierter Addierer/Subtrahierer für $\mathrm{Int}_2(l)$

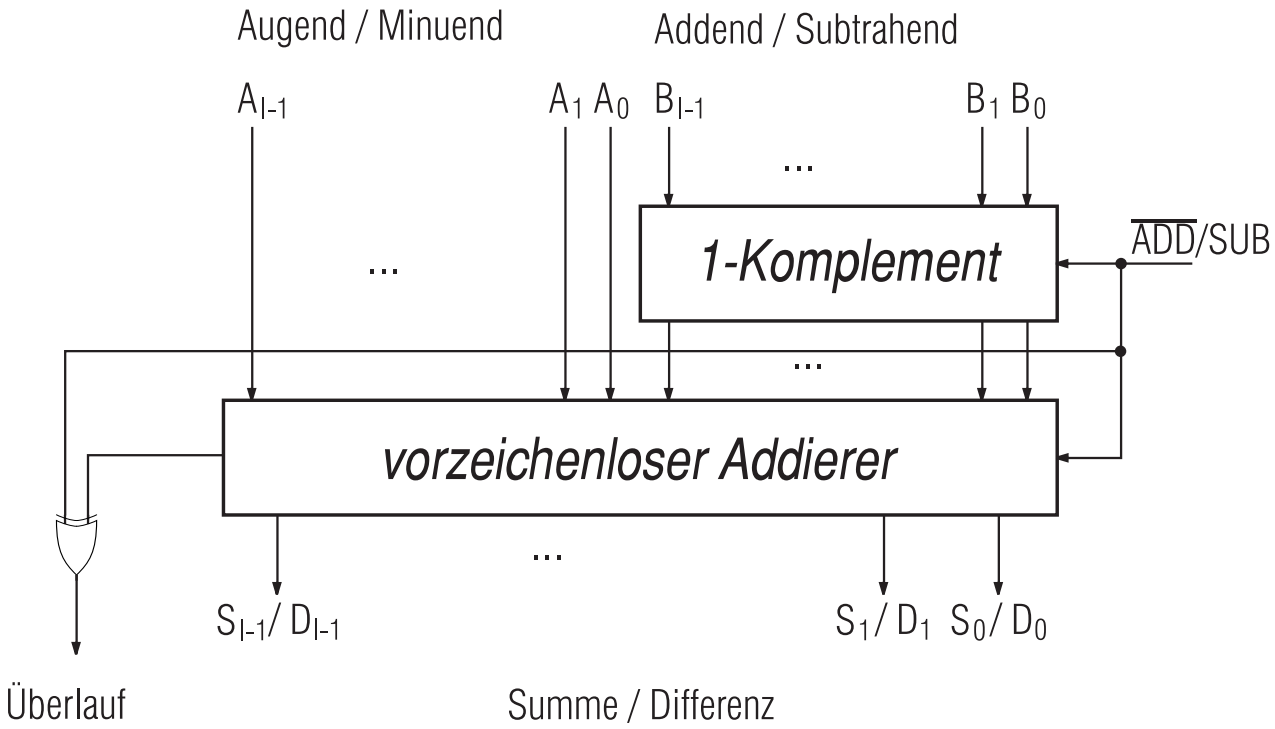
Die Subtraktion in $\mathrm{Int}_2(l)$ kann mittels der Beziehung $A \ominus B = A \oplus \bar{B} \oplus 1$ durchgeführt werden; die zusätzliche 1 kann implizit durch die Initialisierung $C_0 = 1$ berücksichtigt werden.



Kombinierter Addierer/Subtrahierer für $UInt_2(l)$

Dasselbe Verfahren wie für $Int_2(l)$ kann auch für $UInt_2(l)$ alternativ zur vorangehend beschriebenen Methode mit Subtraktionszellen benutzt werden.

Die Überlaufbestimmung unterscheidet sich jedoch von der in $Int_2(l)$.



Addition/Subtraktion in $\mathrm{Int}_1(l)$

Das Verfahren verläuft in zwei Schritten:

$$(\tilde{C}_l, \tilde{S}_{l-1}, \ldots, \tilde{S}_0) = (A_{l-1}, A_{l-2}, \ldots, A_0) + (B_{l-1}, B_{l-2}, \ldots, B_0) + \bar{C}_l$$

mit beliebigem $\bar{C}_l \in \{0, 1\}$

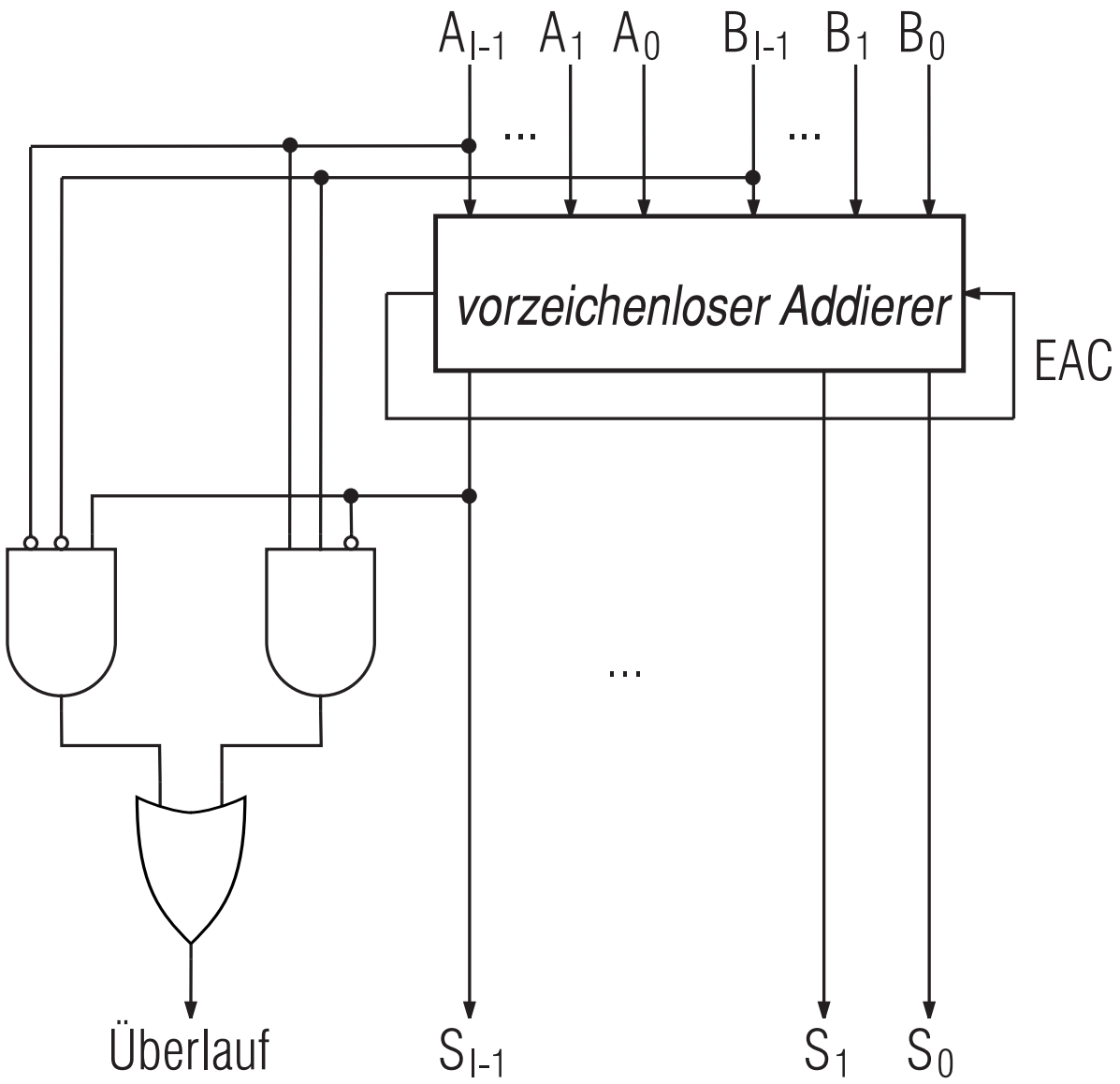
$$(S_{l-1}, \ldots, S_0) = (A_{l-1}, A_{l-2}, \ldots, A_0) + (B_{l-1}, B_{l-2}, \ldots, B_0) + \tilde{C}_l$$

Beide Schritte können mit einem einzigen vorzeichenlosen Addierer durch die Technik des *End-around-Carry* (EAC) integriert werden.

$A_{l-1} = B_{l-1} \neq S_{l-1}$ zeigt Überlauf an.

Subtraktion: $A \ominus B = A \oplus (\ominus B) = A \oplus \bar{B}$

Addierer für $\mathrm{Int}_1(l)$



Addition/Subtraktion in $\mathrm{Int}_{VB}(l)$

Vorzeichen und Beträge werden separat manipuliert.

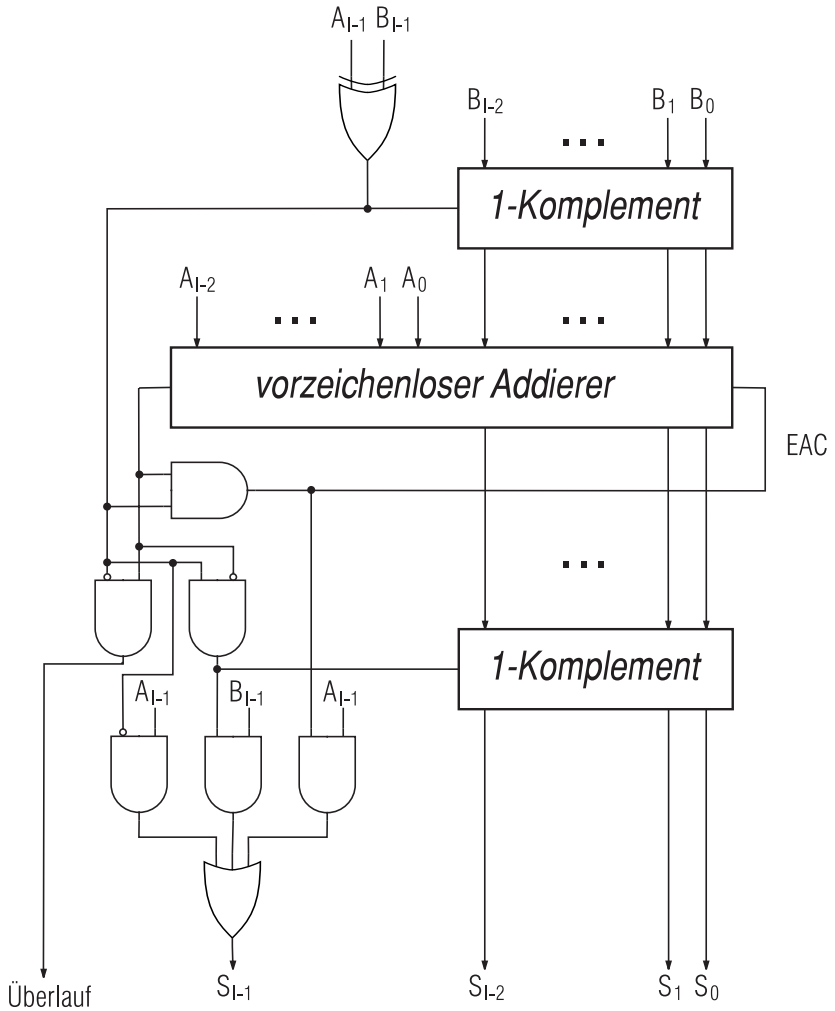
Addition:

$A_{l-1} = B_{l-1} \Rightarrow S_{l-1} = A_{l-1}, |S| = |A| + |B|$
(Überlauf möglich)

$A_{l-1} \neq B_{l-1}, |A| > |B| \Rightarrow S_{l-1} = A_{l-1}, |S| = |A| - |B|$
$A_{l-1} \neq B_{l-1}, |A| < |B| \Rightarrow S_{l-1} = B_{l-1}, |S| = |B| - |A|$
$A_{l-1} \neq B_{l-1}, |A| = |B| \Rightarrow S = \pm 0$

Rückführung der Subtraktion auf die Addition:

$(A_{l-1}, A_{l-2}, \ldots, A_0) - (B_{l-1}, B_{l-2}, \ldots, B_0) =$
$(A_{l-1}, A_{l-2}, \ldots, A_0) + (\bar{B}_{l-1}, B_{l-2}, \ldots, B_0)$



Voll parallele Addierer für Int(H, K)

- *Voll paralleler* Addierer: Dauer einer Addition *nicht* von der Stellenzahl abhängig.
- Einsatzfelder für voll parallele Addierer:
  - Als Komponenten schneller Multiplizierer und Dividierer,
    z. B. dem Verfahren von Schönhage und Strassen mittels diskreter Fourier-Transformation.
  - Echtzeitsysteme mit extrem hohen Datenraten:
    - Signalverarbeitung (lineare Filter, Fourier-Transformation).
    - Krypto-Equipment, z. B. für Public-Key-Codes wie RSA.
  - Hoch parallele Systeme sehr feiner Granularität
    (systolische Arrays, optische Rechner, ziffernserielle Systeme).
  - Systeme mit adaptiver Genauigkeit (exakte Arithmetik).
- Nur durch Verwendung einer redundanten Codierung erreichbar [Mazenc 1993]
- Benutzt wird i. d. R. eine Signed-Digit-Darstellung (SDNR).

Signed-Digit-Darstellungen

- Zur Berechnung einer bestimmten Ziffer einer Summe wird nur ein positional eng benachbarter Abschnitt der Summanden benötigt (*Lokalität*).
- Signed-Digit-Darstellungen treten auch implizit auf, z. B. in Booth-Verfahren, Non-restoring-Division und SRT-Division.
- Vorteile:
  - Voll parallele (und damit extrem schnelle) Addition.
  - Alle Operationen können ziffernseriell (im MSDF-Stil) durchgeführt werden.
  - Keine Sonderbehandlung für Vorzeichenstelle nötig.
- Nachteile:
  - Höherer Speicheraufwand durch Redundanz, größere Anzahl von Anschlüssen.
  - Höherer Hardware-Aufwand zur Implementierung der Operationen.
  - Vorzeichentest, Vergleich, Überlaufbehandlung und Division schwierig.
  - Rückkonvertierung in Standard-Darstellung aufwendig.

Signed-Digit-Number-Representations (SDNR)

- Stellenwertsysteme zu fester Basis R.
- Jede Ziffer besitzt ein eigenes Vorzeichen.
- Fortlaufende Ziffernmenge $\{z \in \mathbb{Z} : -\alpha \leq z \leq \beta\}$, wobei $\alpha, \beta > 0$.
- Redundanz: $\alpha + \beta + 1 > R$.
- Redundanzindex: $\rho = \alpha + \beta + 1 - R > 0$.
- Zahlenbereich: Int(H, K) mit $H = \alpha \times \frac{R^l - 1}{R - 1}$ und $K = 1 + \beta \times \frac{R^l - 1}{R - 1}$

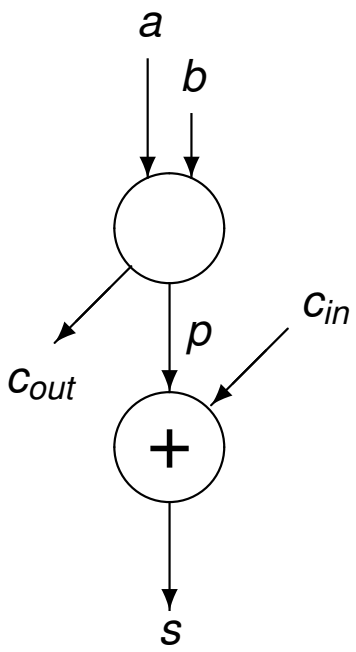
Signed-Binary-Darstellung (SB oder BSD, $R = 2$, $\alpha = \beta = 1$): E. Collignon, 1897!

Ordinary-Signed-Digit-Darstellungen (OSD, $R/2 < \alpha = \beta < R$): A. Avizienis, 1961

General-Signed-Digit-Darstellungen (GSD): B. Parhami, 1990

Für vorzeichenlose Addition genügt die Carry-Save-Darstellung (Metze und Robertson, 1959)

Beispiel eines voll parallelen Addierers für $R = 10$, $\alpha = \beta = 7$



$$\text{Für } a < b \text{ setze } c_{out} = \begin{cases} -1: & a+b < -4 \\ 0: & -4 \le a+b \le 6 \\ 1: & 6 < a+b \end{cases}$$

$$\text{Für } a \ge b \text{ setze } c_{out} = \begin{cases} -1: & a+b < -6 \\ 0: & -6 \le a+b \le 4 \\ 1: & 4 < a+b \end{cases}$$

$s = p + c_{in}$ mit $p = a + b - 10 \times c_{out}$ (garantiert $-6 \le p \le 6$)

	2	7	3	8	a
+	3	2	5	3	b
0	1	1	1	0	c
0	5	-1	-2	1	p
0	6	0	-1	1	s

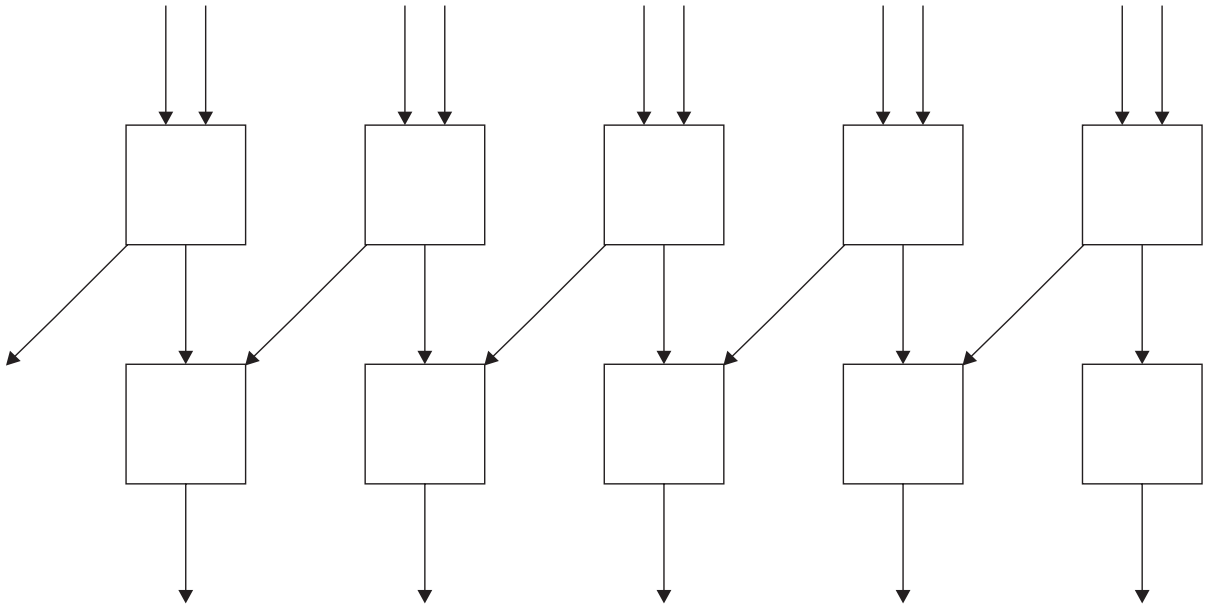
$0 \times 10^4 + 6 \times 10^3 = 6 \times 10^3$

	3	2	5	3	a
+	2	7	3	8	b
1	1	1	1	0	c
0	-5	-1	-2	1	p
1	-4	0	-1	1	s

$1 \times 10^4 + (-4) \times 10^3 = 6 \times 10^3$

„Übertragsfreie“ Addition

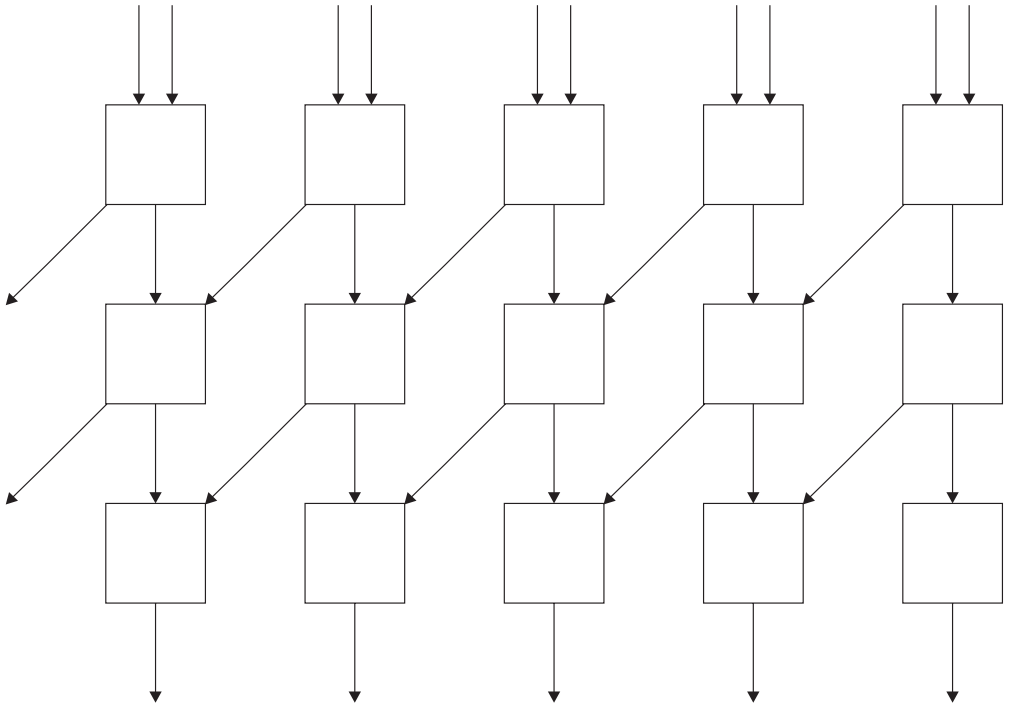
$R > 2, \alpha + \beta \geq R + 2$ oder $R > 2, \alpha + \beta = R + 1, \alpha > 1, \beta > 1$



Zellen müssen nicht unbedingt algebraische Dekomposition leisten!

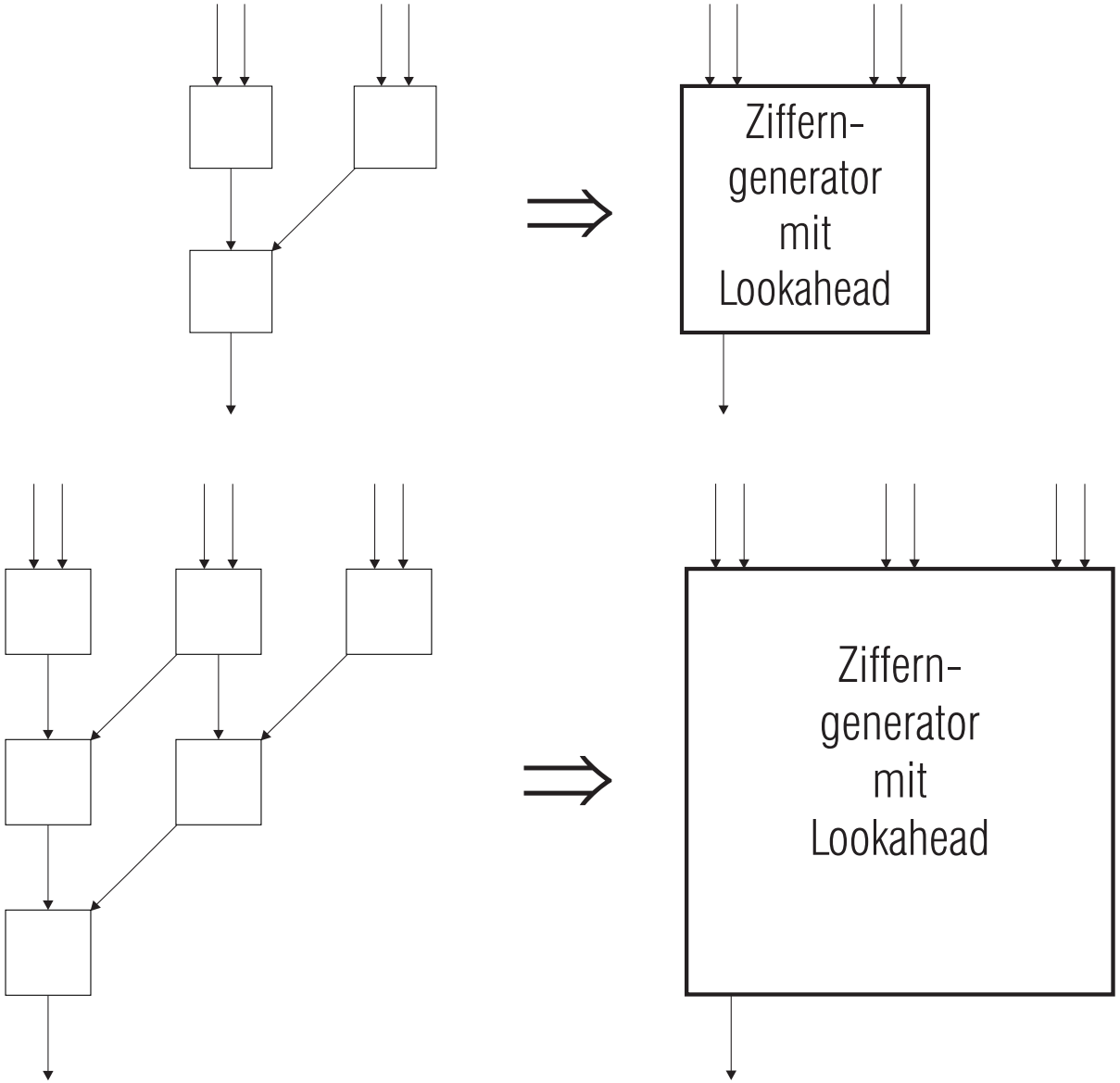
„Übertragsbeschränkte“ Addition

$R = 2$ oder $\alpha + \beta = R$ oder $\alpha = 1, \beta = R$ oder $\alpha = R, \beta = 1$

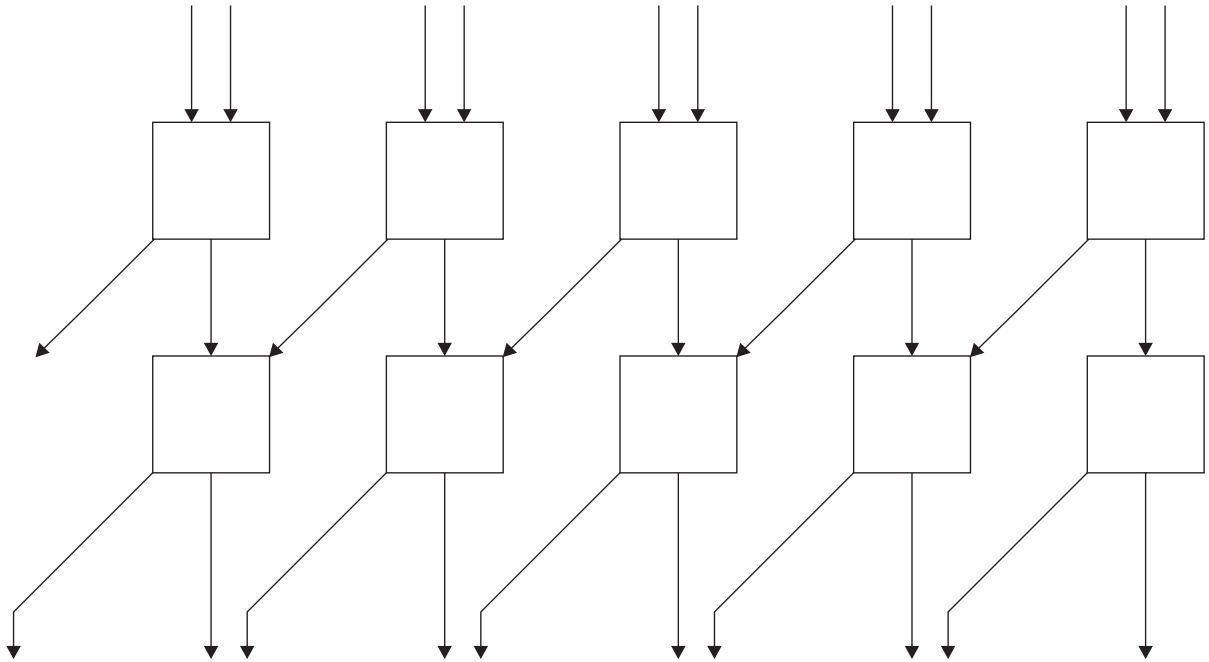


Übertragsbeschränkte Addition auch möglich für Carry-Save-Darstellungen mit $S = [0, R] \cap \mathbb{Z}$.

Beschleunigung durch Lookahead



Übertragsbeschränkte Addition mit trivialer dritter Stufe



Addierer für $R = 2, \alpha = \beta = 1$ von J. Duprat und J.-M. Muller, 1991

Spezielle Codierung der Ziffern spielt entscheidende Rolle: (p, m) mit $s = I(p, m) = p - m$.

Die Repräsentation entsteht aus der 1-Komplement-Darstellung der Länge $l = 2$ durch Vertauschung der Bits.

Zifferncodierung: Fallstudie für $R = 2, \alpha = \beta = 1$ (Zehendner, 1992)

Jeder SDNR-Addierer kann durch boolesche Formeln auf Bit-Ebene beschrieben werden.

Aber nicht jeder SDNR-Addierer kann auch auf Ziffern-Ebene beschrieben werden.

Von den möglichen 2-Bit-Codierungen der Ziffern sind 36 redundant,
die übrigen 24 lassen je eine Bit-Kombination ungenutzt.

Folgende Transformationen führen auf insgesamt 6 Klassen:

- Vertauschung der Repräsentationsbits
- Invertierung eines oder beider Repräsentationsbits
- Vertauschung der Repräsentationen für 1 und -1

Die Klassen zeigen bezüglich der Implementierung recht unterschiedliches Verhalten. Beispiel:

Die (p, m)-Darstellung berechnet jedes Resultatbit aus nur je 7 Operandenbits;
es gibt aber auch Darstellungen, die alle 12 Operandenbits für jedes Resultatbit benötigen.

NB: 1K-, 2K- und VB-Darstellung liegen in drei unterschiedlichen Klassen.

Die Anzahl möglicher Implementierungen ist gigantisch!!

Weitere Operationen

- Subtraktion: Über Vorzeichenwechsel und Addition unter Nutzung von $x - y = x + (-y)$.
- Vorzeichenwechsel:
  - Nur Vorzeichenwechsel jeder einzelnen Ziffer, falls $\alpha = \beta$.
  - GSD: Kann durch überlauffreie Umcodierung realisiert werden.
- Konvertierungen:
  Allgemeine Vorgehensweise bei der Konvertierung: Jede im Zielsystem nicht existente Ziffer wird in eine dort existente Ziffer derselben Position und einen zugehörigen Übertrag (positiv oder negativ) umgeschrieben. Dieser Prozess muss evtl. iteriert werden.
  - Binärdarstellung $\longrightarrow$ BSD: Einfügen des globalen Vorzeichens in jede Ziffer.
  - Beliebige Darstellungen $\longrightarrow$ GSD: Durch Umcodierung mit beschränktem Überlauf realisierbar.
  - BSD $\longrightarrow$ Binärdarstellung: Trennung der negativen und positiven Anteile (besonders einfach für (p, m)-Darstellung), dann gewöhnliche Subtraktion.
  - GSD $\longrightarrow$ Binärdarstellung: Erfordert meist zusätzlich ziffernweise Umcodierung.

Prädikate

- Test auf Null bzw. auf Gleichheit:
  - Ziffernweiser Test funktioniert nicht immer!
  - Für $\alpha, \beta < R$ ist die Null eindeutig; Test einfach, aber langsam.
  - Test auf Gleichheit: $a = b \Leftrightarrow (a - b) =_0$; Subtraktion parallel.
- Vorzeichentest und Größenvergleiche:
  - Schwierig, da Vorzeichen i. A. von allen Ziffern abhängen kann.
  - Für $\alpha, \beta < R$ gibt die höchstwertige, von Null verschiedene Ziffer das Vorzeichen an.
  - Reduzierung der Latenz durch „unscharfe“ Vergleiche.
- Überlauf:
  - Tritt evtl. unnötigerweise auf.
  - Erkennung und Korrektur fiktiver Überläufe aufwendig und langsam.
  - Überlaufarithmetik und Sättigungsarithmetik schlecht realisierbar.
  - Zirkuläre Arithmetik besser realisierbar (aber Ergebnisse redundant).

Komplexität von Addierern für Int(H, K)

Bei gleicher Implementierungstechnik ist die Addition in Vorzeichen/Betrag-Darstellung am aufwendigsten, in 2-Komplement-Darstellung am einfachsten.

Der Aufwand für die 1-Komplement-Darstellung liegt gewöhnlich dazwischen, außer beim Conditional-Sum-Addierer, für den sich kein wesentlicher Unterschied zwischen der 1-Komplement-Darstellung und der 2-Komplement-Darstellung ergibt.

Ein paralleler Addierer für eine Signed-Digit-Darstellung verursacht etwa den doppelten Aufwand eines Ripple-Carry-Addierers.

Ein serieller Addierer für eine Signed-Digit-Darstellung verursacht etwa den doppelten Aufwand eines Volladdierers.

Weiterführende Literatur zu Signed-Digit-Darstellungen

A. Avizienis: Signed-digit number representations for fast parallel arithmetic. IRE Transactions on Electronic Computers, Vol. 10, pp. 389–400, 1961.

J. Duprat und J.-M. Muller: Ecrire les nombres autrement pour calculer plus vite. Technique et Science Informatiques, Vol. 10, pp. 211–224, 1991.

B. Parhami: Generalized signed-digit number systems: A unifying framework for redundant number representations. IEEE Transactions on Computers, Vol. 39, No. 1, pp. 89–98, 1990.

B. Parhami: Computer arithmetic. Algorithms and hardware designs. Oxford University Press, New York, 2000.

E. Zehendner: Efficient implementation of regular parallel adders for binary signed digit number representations. Microprocessing and Microprogramming, Vol. 35, pp. 319–326, 1992.

Carry-Save-Addierer (CSA)

Summierung der Teilprodukte einer Multiplikation oder eines inneren Produkts (Skalarprodukt, Berechnung von Durchschnitten, etc.) erfordert Mehr-Operanden-Addition.

Mehr-Operanden-Addierer benutzen häufig Carry-Save-Addierer (CSA) als Teilstrukturen.

Ein l-Bit-CSA besitzt drei Operanden der Wortlänge l Bit
und erzeugt zwei Resultate der Wortlänge l Bit (Partialsumme und Übertragswort).

Jedes dieser Resultate kann als Operand weiterer CSA oder gewöhnlicher Addierer dienen.

Ein CSA wird implementiert durch ein Array von Volladdierern, die parallel zueinander arbeiten, also ohne Verbindung zwischen den einzelnen Addierern.

Addierer für mehr als drei Operanden können aus mehreren CSA aufgebaut werden.

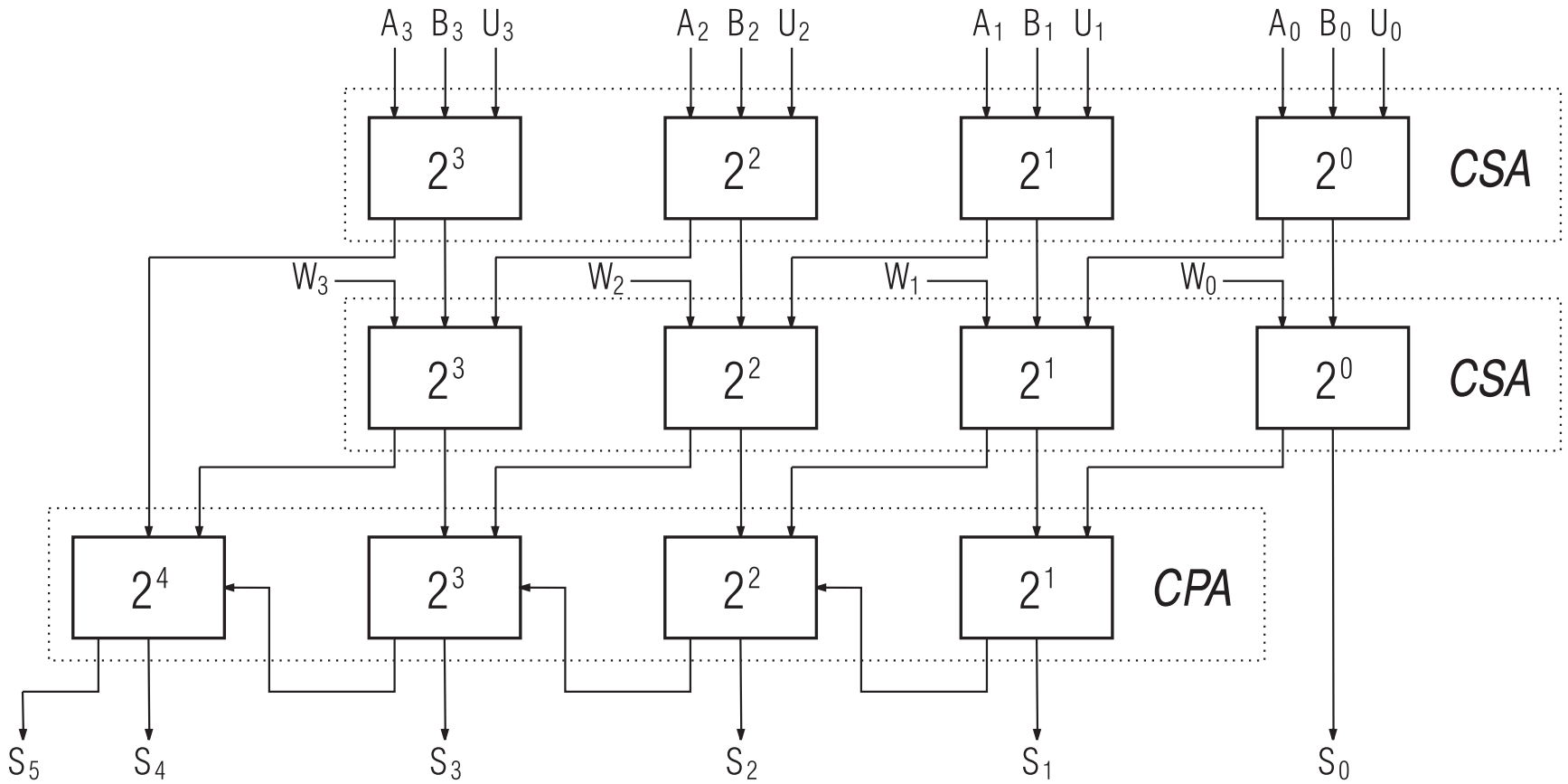
Den Abschluss eines Netzes von CSA bildet ein (meist schneller) *Carry-Propagate-Addierer* (CPA), z. B. ein Carry-Lookahead-Addierer.

Vorteil: Die Propagierung von Überträgen tritt nur in einer abschließenden Phase des Verfahrens auf, alle vorhergehenden Additionsschritte erfolgen mit sehr geringer Latenz.

Kaskaden von Carry-Save-Addierern

Zur Addition von k Operanden werden $(k-2)$ CSA und ein CPA benötigt.

Die Latenz bei Schaltung der CSA als Kaskade beträgt $t = (k-2) \times t_{CSA} + t_{CPA}$
mit $t_{CSA} = t_{VA}$ und $t_{CPA} \leq (\lceil \log_2 k \rceil + l - 1) \times t_{VA}$

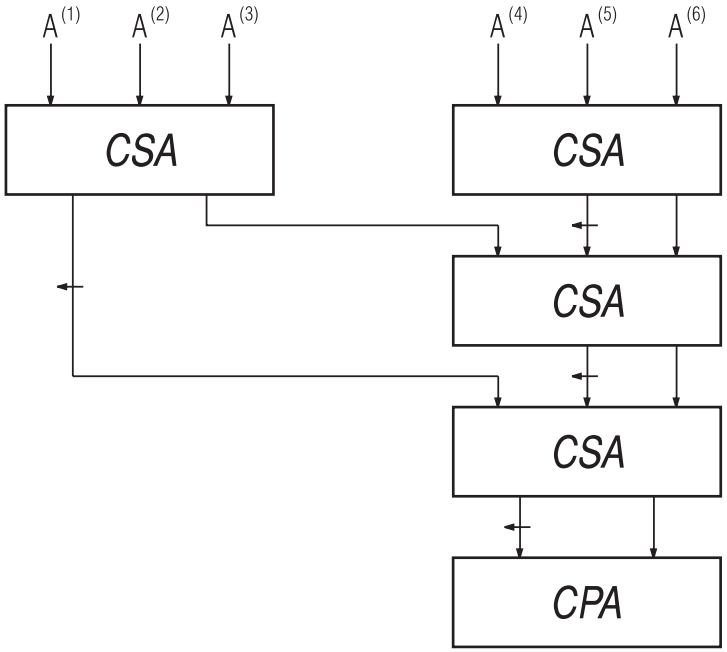


Wallace-Bäume

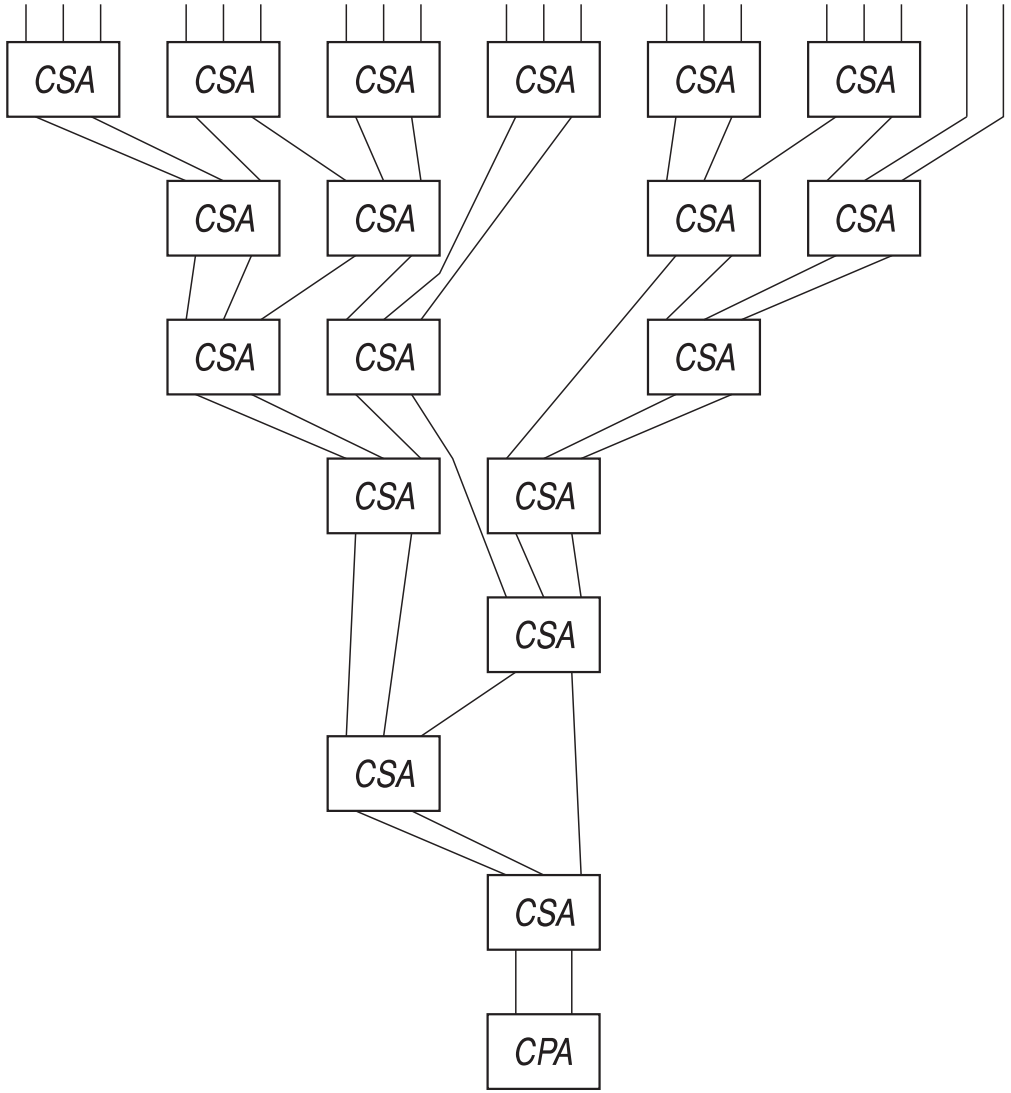
Dieselbe Anzahl von CSA wie in der Kaskade für k Operanden kann zu einer Baumstruktur mit minimaler Latenz verschaltet werden: $t = h \times t_{CSA} + t_{CPA}$ mit $h \approx \log_{3/2}(k/2)$

Anzahl k von Operanden	Optimale Stufenzahl h
3	1
4	2
5–6	3
7–9	4
10–13	5
14–19	6
20–28	7
29–42	8
43–63	9

Wallace-Bäume (Beispiele)



Wallace-Baum für 6 Summanden

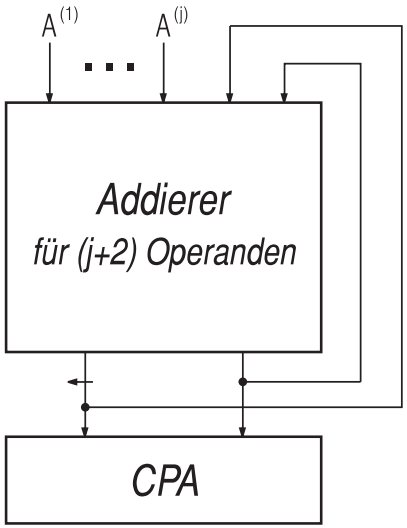


Wallace-Baum für 20 Summanden

Iterative Mehr-Operanden-Addierer

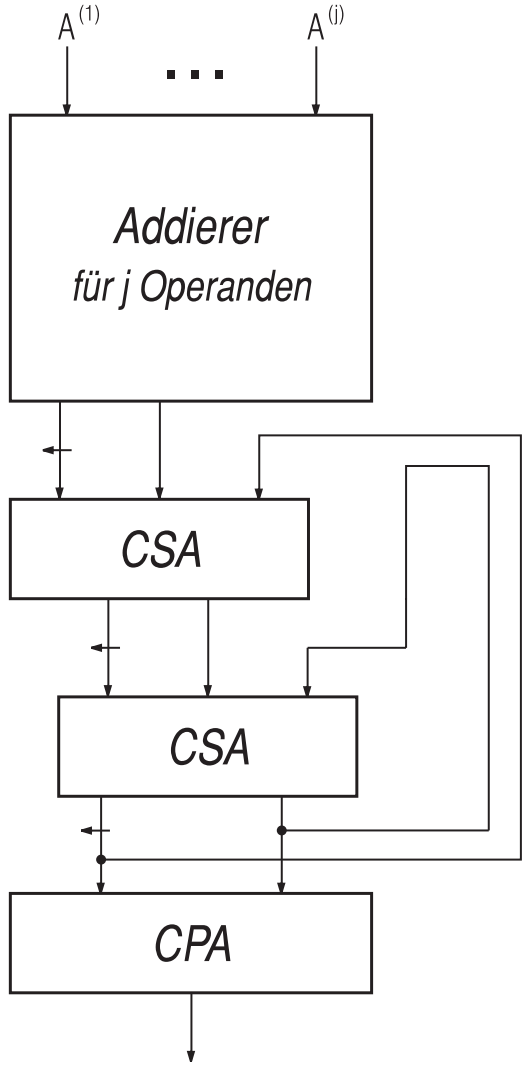
Mehr-Operanden-Addierer für eine große Anzahl von Operanden können eventuell für eine direkte Implementierung zu aufwendig sein.

Stattdessen können die Ausgänge eines kleineren Mehr-Operanden-Addierers auf seine Eingänge rückgekoppelt werden; der Addierer wird dabei iterativ genutzt.



Eingabe in den Addierer für $(j + 2)$ Operanden in Gruppen zu je j Stück.

Optimierte Beschaltung: $(j + 2)$ Operanden beim ersten Durchlauf.



Iterativer Mehr-Operanden-Addierer mit Pipelining

(r, m)-Zähler

Ein Volladdierer heißt im Kontext von Carry-Save-Addition auch $(3, 2)$-Zähler, da er drei 1-Bit-Eingaben addiert und die Summe als 2-Bit-Zahl darstellt.

Statt mit $(3, 2)$-Zählern kann auch allgemeiner mit (r, m)-Zählern gearbeitet werden: Die Ausgabe ist eine m-Bit-Codierung der Anzahl der Einsen in den r 1-Bit-Eingaben. Folglich hat $m \geq \log_2(r + 1)$ zu gelten.

Schaltungen aus (r, m)-Zählern mit $r > 3$ haben meist kürzere Latenzen als entsprechende Schaltungen aus $(3, 2)$-Zählern, falls Gatter mit ausreichendem Fan-in zur Verfügung stehen.

Ein (r, m)-Zähler kann, außer durch ein Schaltnetz, auch durch ein ROM implementiert werden, das eine Größe von $m \times 2^r$ Bits besitzt.

Geschwindigkeitsvorteile ergeben sich dabei aber erst für großes r.

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Multiplikation

Multiplikation in $\mathrm{UInt}_2(l), \mathrm{Int}_2(l), \mathrm{Int}_1(l), \mathrm{Int}_{VB}(l)$

Multiplikand $A = A_{l-1}\, A_{l-2} \ldots A_0$

Multiplikator $B = B_{m-1}\, B_{m-2} \ldots B_0$

Produkt $A \times B = P = P_{n-1}\, P_{n-2} \ldots P_0$

Multiply-Add-Operation: $A \times B + E = P = P_{n-1}\, P_{n-2} \ldots P_0$

- Vorzeichenlose Multiplikation:
  Unter der Voraussetzung $n \geq l + m$ erfolgt die Multiplikation stets überlauffrei.
  Für Multiply-Add muss zusätzlich $E \leq 2^l + 2^m - 2$ gelten.
- Multiplikation mit Vorzeichen:
  Unter der Voraussetzung $n \geq l + m$ erfolgt die Multiplikation stets überlauffrei;
  für die 2-Komplement-Darstellung ist diese Stellenzahl auch notwendig.
  Für Vorzeichen/Betrag-Darstellung und 1-Komplement-Darstellung genügen sogar
  $n = l + m - 1$ Stellen; im Falle $m \leq 2$ oder $l \leq 2$ reichen bereits $n = l + m - 2$ Stellen.

Linksverschiebung als Multiplikation mit der Zahlenbasis

$(A_{l-1}, \ldots, A_0) \rightarrow (B_{l-1}, \ldots, B_0) = (B_{l-1}, A_{l-3}, \ldots, A_0, B_0)$

- Vorzeichenlose Multiplikation: $B_{l-1} = A_{l-2}$, $B_0 = 0$ (Überlauf für $A_{l-1} \neq 0$).
- 2-Komplement-Darstellung: $B_{l-1} = A_{l-2}$, $B_0 = 0$ (Überlauf für $A_{l-1} \neq A_{l-2}$).
- 1-Komplement-Darstellung: $B_{l-1} = A_{l-2}$, $B_0 = A_{l-1}$ (Überlauf für $A_{l-1} \neq A_{l-2}$).
- Vorzeichen/Betrag-Darstellung: $B_{l-1} = A_{l-1}$, $B_0 = 0$ (Überlauf für $A_{l-2} \neq 0$).

Multiplizierer

Der Vorgang der Multiplikation besteht aus zwei grundlegenden Operationen:

- Erzeugung von Teilprodukten,
- Aufsummierung der Teilprodukte (nach Verschieben entsprechend Gewichtung).

Eine Beschleunigung der Multiplikation kann deshalb erreicht werden durch

- Reduzierung der Anzahl der Teilprodukte,
- Beschleunigung des Summationsprozesses.

Typen von Multiplizierern:

- Ein *sequentieller Multiplizierer* erzeugt die Teilprodukte nacheinander und addiert jedes solche Teilprodukt zur bisherigen Zwischensumme.
- Ein *paralleler Multiplizierer* erzeugt alle Teilprodukte simultan und benutzt dann einen schnellen Mehr-Operanden-Addierer für die Summierung.
- Ein *Array-Multiplizierer* besteht aus einem Array identischer Zellen, die Teilprodukte gleichzeitig erzeugen und summieren.

Serieller Multiplizierer

Bei serieller Multiplikation werden beide Operanden (Multiplikator und Multiplikand) seriell abgearbeitet, üblicherweise von den niederwertigsten zu den höchstwertigen Stellen.

In jedem Multiplikationszyklus wird ein Bit des Multiplikators zur seriellen Generierung eines Teilprodukts benutzt, das durch einen seriellen Addierer zur Zwischensumme addiert wird.

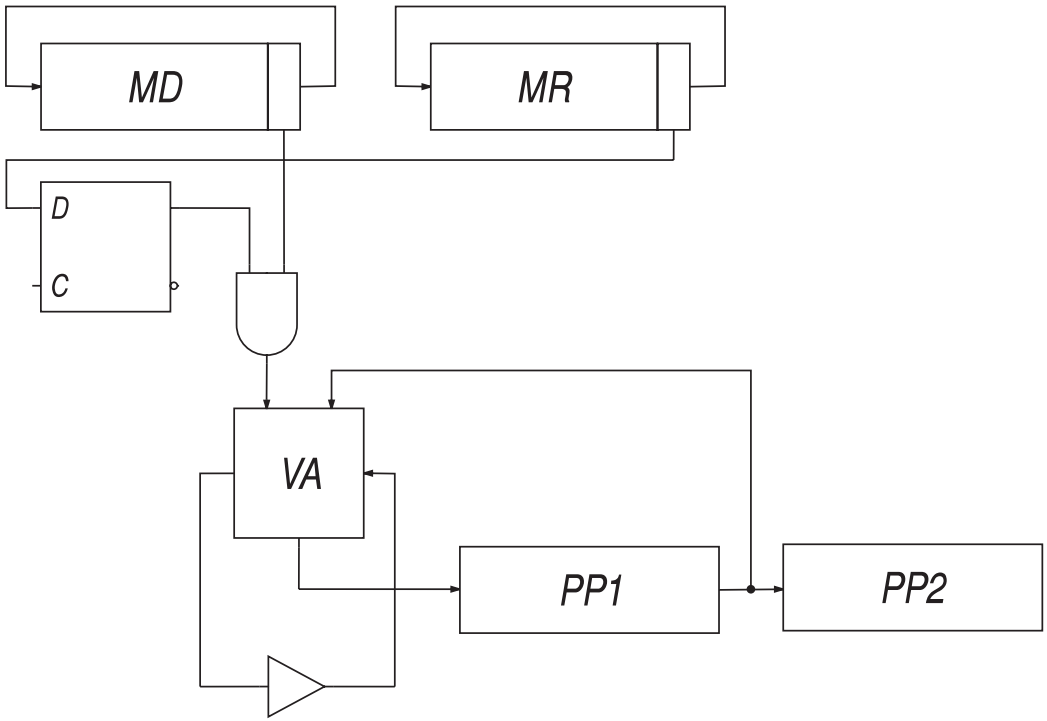
Der Multiplikand wird bei dieser Technik mehrfach seriell durchlaufen.

Das AND-Gatter stellt einen (1 Bit)$\times$(1 Bit)-Multiplizierer dar.

Für Operandenlängen l und m dauert die Multiplikation $l \times m$ Takte.

Der Aufwand (ohne die Register) ist unabhängig von l und m.

In der Zeichnung fehlt die Reset-Logik!



Naive Multiplikation für vorzeichenlose binäre Multiplikatoren

Bei „Multiplikation von Hand“ werden alle Teilprodukte sequentiell berechnet und anschließend spaltenweise von rechts sequentiell addiert (mit Übertrag).

Bei sequentieller Multiplikation durch eine Maschine werden die Teilprodukte zwar sequentiell berechnet, nach jeder solchen Berechnung aber sofort zur Zwischensumme summiert.

$$
\begin{array}{lcll}
P^{(0)} & = & 0 & \text{(bzw. } P^{(0)} = E \text{ für Multiply-Add)} \\
P^{(j+1)} & = & P^{(j)} + A \times B_j \times 2^j, & j = 0, 1, 2, \ldots, m-1 \\
P & = & P^{(m)} &
\end{array}
$$

Die Multiplikation mit 2^j erfolgt durch arithmetisches Verschieben um j Stellen nach links.

Negative Summanden $A \times B_j \times 2^j$ bzw. E sind auf die Breite des Akkumulators zu erweitern.

Nachteil dieses Verfahrens: Zu addierendes Vielfaches fällt an unterschiedlichen Stellen an.

- Entweder muss der Addierer für $l + m - 1$ Stellen ausgelegt werden.
- Oder der Addierer ist zwar nur für l Stellen ausgelegt, wird aber in jedem Schritt anders mit dem Akkumulator verbunden.

Naive Multiplikation (Beispiel)

			21 =	1	0	1	0	1		Multiplikand
			27 =	1	1	0	1	1		Multiplikator
				0	0	0	0	0		initiale Zwischensumme
				1	0	1	0	1		+ 1. Vielfaches
			0	1	0	1	0	1		
			1	0	1	0	1			+ 2. Vielfaches
		0	1	1	1	1	1	1		
		0	0	0	0	0				+ 3. Vielfaches
	0	0	1	1	1	1	1	1		
	1	0	1	0	1					+ 4. Vielfaches
0	1	1	1	0	0	1	1	1		
1	0	1	0	1						+ 5. Vielfaches
567 = 1	0	0	0	1	1	0	1	1	1	vollständiges Produkt

Optimierter Algorithmus für die sequentielle Multiplikation

$$
\begin{aligned}
P^{(0)} &= 0 && \text{(bzw. } P^{(0)} = E \text{ für Multiply-Add)} \\
P^{(j+1)} &= (P^{(j)} + A \times B_j)/2, && j = 0, 1, 2, \ldots, m-1 \\
P &= P^{(m)} \times 2^m
\end{aligned}
$$

Division durch 2 wird durch arithmetisches Verschieben um eine Stelle nach rechts realisiert.

Das Register P muss dazu an der rechten Seite um m „Nachkommastellen“ verlängert werden; die entsprechenden m Stellen an der linken Seite des Registers entfallen dabei.

Der Operand B kann zu Beginn der Multiplikation komplett in den m Nachkommastellen von P untergebracht werden; er verkürzt sich dann in jedem Schritt um ein Bit.

Die Multiplikation mit 2^m ist nicht wirklich erforderlich (virtuelle Skalierung durch Uminterpretation des Ergebnisregisters, d. h. Entfernung des gedachten Kommas).

Produktbildung $A \times B_j$ durch Auswahl von 0 bzw. A als zweitem Operanden der Addition.

Für $B_j = 0$ kann die Addition auch einfach unterdrückt werden.

Optimierter Algorithmus für die sequentielle Multiplikation (Beispiel)

21 =		1	0	1	0	1				Multiplikand
27 =		1	1	0	1	1				Multiplikator
		0	0	0	0	0				initiale Zwischensumme
		1	0	1	0	1				+ 1. Vielfaches
	0	1	0	1	0	1				
		0	1	0	1	0	1			Ergebnis halbieren
		1	0	1	0	1				+ 2. Vielfaches
	0	1	1	1	1	1	1			
		0	1	1	1	1	1	1		Ergebnis halbieren
		0	0	0	0	0				+ 3. Vielfaches
	0	0	1	1	1	1	1	1		
		0	0	1	1	1	1	1	1	Ergebnis halbieren
		1	0	1	0	1				+ 4. Vielfaches
	0	1	1	1	0	0	1	1	1	
		0	1	1	1	0	0	1	1	1	Ergebnis halbieren
		1	0	1	0	1				+ 5. Vielfaches
567 =	1	0	0	0	1	1	0	1	1	1	vollständiges Produkt

Alternativer Algorithmus für die sequentielle Multiplikation

$$
\begin{aligned}
P^{(0)} &= 0 && \text{(bzw. } P^{(0)} = E \times 2^{-m} \text{ für Multiply-Add)} \\
P^{(j+1)} &= 2 \times P^{(j)} + A \times B_{m-j-1}, && j = 0, 1, 2, \ldots, m-1 \\
P &= P^{(m)}
\end{aligned}
$$

Die Multiplikation mit 2 erfolgt durch Verschieben um eine Stelle nach links.

Der Operand B kann zu Beginn der Multiplikation komplett in den m höchstwertigen Bits von P untergebracht werden; er verkürzt sich dann in jedem Schritt um ein Bit.

Nachteile dieses Verfahrens:

Die Addition erstreckt sich über $l + m - 1$ Stellen
(für die höchstwertigen $m - 1$ Stellen des Addierers genügen allerdings Inkrementierzellen).

Da die Linksverschiebung von P vor Beginn der Addition erfolgt,
muss das jeweils aktuelle Multiplikatorbit zwischengespeichert werden.

Bei einer Multiply-Add-Operation muss der Summand E bit-seriell zugeführt werden.

Alternativer Algorithmus für die sequentielle Multiplikation (Beispiel)

				21 =	1	0	1	0	1	Multiplikand
				27 =	1	1	0	1	1	Multiplikator
					0	0	0	0	0	initiale Zwischensumme
				0	0	0	0	0	0	Ergebnis verdoppeln
					1	0	1	0	1	+ 1. Vielfaches
				0	1	0	1	0	1	
			0	1	0	1	0	1	0	Ergebnis verdoppeln
					1	0	1	0	1	+ 2. Vielfaches
			0	1	1	1	1	1	1	
		0	1	1	1	1	1	1	0	Ergebnis verdoppeln
					0	0	0	0	0	+ 3. Vielfaches
		0	1	1	1	1	1	1	0	
	0	1	1	1	1	1	1	0	0	Ergebnis verdoppeln
					1	0	1	0	1	+ 4. Vielfaches
	1	0	0	0	1	0	0	0	1	
1	0	0	0	1	0	0	0	1	0	Ergebnis verdoppeln
					1	0	1	0	1	+ 5. Vielfaches
567 =	1	0	0	0	1	1	0	1	1	1	vollständiges Produkt

Verfahren für vorzeichenbehafteten Multiplikator

- Multiplikator in Vorzeichen/Betrag-Darstellung

$$
\begin{aligned}
P^{(0)} &= 0 && \text{(bzw. } P^{(0)} = E \text{ für Multiply-Add)} \\
P^{(j+1)} &= (P^{(j)} + A \times B_j)/2, && j = 0, 1, 2, \ldots, m-2 \\
P &= (-1)^{B_{m-1}} \times P^{(m-1)} \times 2^{m-1}
\end{aligned}
$$

  Alternativ Beträge vorzeichenlos und Vorzeichen durch XOR multiplizieren.

- Multiplikator in 2-Komplement-Darstellung

$$
\begin{aligned}
P^{(0)} &= 0 && \text{(bzw. } P^{(0)} = E \text{ für Multiply-Add)} \\
P^{(j+1)} &= (P^{(j)} + A \times B_j)/2, && j = 0, 1, 2, \ldots, m-2 \\
P^{(m)} &= (P^{(m-1)} - A \times B_{m-1})/2 \\
P &= P^{(m)} \times 2^m
\end{aligned}
$$

- Multiplikator in 1-Komplement-Darstellung

$$
\begin{aligned}
P^{(0)} &= A \times B_{m-1} \\
P^{(j+1)} &= (P^{(j)} + A \times B_j)/2, && j = 0, 1, 2, \ldots, m-2 \\
P^{(m)} &= (P^{(m-1)} - A \times B_{m-1})/2 \\
P &= P^{(m)} \times 2^m
\end{aligned}
$$

Radix-4-Multiplikation

Die Anzahl der Teilprodukte wird auf $\lceil m/2 \rceil$ reduziert, wenn jeweils 2 Bits des Multiplikators gemeinsam ausgewertet werden. Beispiel:

$A = 5 = 0000101$	0000000	initiale Zwischensumme
$B = 114 = 1110010$	000001010	+ 1. Vielfaches ($2 \times A$)
	000001010	
	000001010	Zwischensumme 2 Stellen rechtsschieben
	000000000	+ 2. Vielfaches ($0 \times A$)
	00000001010	
	00000001010	Zwischensumme 2 Stellen rechtsschieben
	000001111	+ 3. Vielfaches ($3 \times A$)
	000011111010	
	000011111010	Zwischensumme 2 Stellen rechtsschieben
	000000101	+ 4. Vielfaches ($1 \times A$)
$5 \times 114 = 570 =$	0001000111010	vollständiges Produkt

Radix-2^k-Multiplikation

Werden gleichzeitig 3 Bits des Multiplikators betrachtet, resultiert Radix-8-Multiplikation, usw.

Radix-2^k-Multiplikation erfordert die Vielfachen $h \times A, h = 0, 1, \ldots, 2^k - 1$ des Multiplikanden; diese entstehen zum Teil durch Linksverschiebung direkt aus dem Multiplikanden A.

Weitere Vielfache ergeben sich daraus durch Summen-/Differenzenbildung;
aus diesen wiederum werden weitere Vielfache durch erneute Linksverschiebung gewonnen.
Dieser Vorgang kann dann noch beliebig iteriert werden.

Aus Zeitgründen erfolgt die Summen-/Differenzenbildung vor Generierung des ersten Teilprodukts; die so gewonnenen Vielfachen werden in Registern abgespeichert.

Verschieben kostet keine Zeit, da topologisch implementierbar.

Vielfache, die sich durch Verschieben entweder direkt aus dem Multiplikanden oder aus den abgespeicherten Vielfachen ergeben, brauchen nicht abgespeichert zu werden (*Wired-Shift*, Generierung *on-the-fly*).

Radix-8-Multiplikation

Multiplikatorbits $B_{3i+2}, B_{3i+1}, B_{3i}$	Aktion	Bildungsgesetz
000	$+ 0 \times A; \gg 3$	
001	$+ 1 \times A; \gg 3$	
010	$+ 2 \times A; \gg 3$	$A \ll 1$
011	$+ 3 \times A; \gg 3$	$A + A \ll 1$
100	$+ 4 \times A; \gg 3$	$A \ll 2$
101	$+ 5 \times A; \gg 3$	$A + A \ll 2$
110	$+ 6 \times A; \gg 3$	$A \ll 1 + A \ll 2$ oder $(A + A \ll 1) \ll 1$
111	$+ 7 \times A; \gg 3$	$A + A \ll 1 + A \ll 2$ oder $A \ll 3$ - A

Nachteile der Radix-2^k-Multiplikation

- Erhöhter Hardware-Aufwand.
- Größere Latenz der Einzelschritte.

Zusatzaufwand bei Radix-2^k-Multiplikation gegenüber Radix-2-Multiplikation durch:

- Erzeugung von Vielfachen des Multiplikanden.
- Auswahl eines Vielfachen (1-aus-2^k-Multiplexer).
- Verlängerung des Addierers um k Stellen.
- Ergänzung des Multiplikators auf eine ohne Rest durch k teilbare Stellenzahl.
- Um $(k-1)$ Stellen vergrößerte Verschiebedistanzen.

Schieben über Gruppen von Nullen und Einsen im Multiplikator

Eine durchschnittliche Beschleunigung lässt sich erzielen, indem für eine Gruppe aufeinanderfolgender Nullen des Multiplikators zwar die Zwischensumme um die jeweilige Anzahl von Bits nach rechts verschoben, aber nichts addiert wird.

Auch für eine Gruppe aufeinanderfolgender Einsen des Multiplikators können weniger Teilprodukte erzeugt werden. Hierzu benutzt man die Beziehung

$$\sum_{i=j}^{h} 2^i = 2^{h+1} - 2^j$$

Statt $h - j + 1$ Additionen (entsprechend der Anzahl aufeinanderfolgender Einsen) benötigt man dann nur noch eine Addition und eine Subtraktion.

Verfahren von Booth für vorzeichenlose ganze Binärzahlen

Das Verfahren von Booth (1951!) kombiniert das Schieben über Folgen von Nullen bzw. Einsen auf elegante Weise.

Die jeweilige Aktion an der Bitposition i hängt von den Multiplikatorbits B_i und B_{i-1} ab:

B_i	B_{i-1}	Aktion
0	0	$\gg 1$
0	1	$+ A; \gg 1$
1	0	$- A; \gg 1$
1	1	$\gg 1$

Formal muss $B_{-1} = 0$ gesetzt werden, da B keine Nachkommastellen besitzt, also zunächst von einer laufenden Folge von Nullen ausgegangen wird.

Außerdem muss eine Stelle $B_m = 0$ ergänzt werden, da ansonsten irrtümlich eine Subtraktion in der höchstwertigen Position von B auftreten könnte.

Redundante Codierung des Multiplikators

Die Wirkungsweise des Booth-Verfahrens wird noch deutlicher, wenn man den Multiplikator zunächst in einem ternären Code mit den Symbolen 0, 1 und $\bar{1}$ (für -1) darstellt (*Recoding*).

Der Multiplikator B wird folgendermaßen in einen neuen Multiplikator B' umcodiert:

B_i	B_{i-1}	B'_i	Aktion	Bemerkung
0	0	0	Verschieben	laufende Folge von Nullen
0	1	1	Addieren und Verschieben	Ende einer Folge von Einsen
1	0	$\bar{1}$	Subtrahieren und Verschieben	Anfang einer Folge von Einsen
1	1	0	Verschieben	laufende Folge von Einsen

Die B'_i können parallel oder on-the-fly berechnet werden.

Beispiel: 6 Additionen können ersetzt werden durch 2 Additionen und 2 Subtraktionen.

i	8	7	6	5	4	3	2	1	0	-1
B_i	0	1	1	1	1	0	0	1	1	0
B'_i	1	0	0	0	$\bar{1}$	0	1	0	$\bar{1}$	—

Verfahren von Booth für 2-Komplement-Darstellung

Für Multiplikatoren in 2-Komplement-Darstellung darf keine Ziffer B_m ergänzt werden, da nur B_{m-1} das Vorzeichen von B trägt, in B' dagegen jede Ziffer ihr eigenes Vorzeichen besitzt. Beispiel:

A		1	0	1	1				Multiplikand -5
B	$\times$	1	1	0	1				Multiplikator -3
B'		0	$\bar{1}$	1	$\bar{1}$				umcodierter Multiplikator
initialisieren		0	0	0	0				
$-A$	+	0	1	0	1				
		0	1	0	1				
rechtsschieben		0	0	1	0	1			
$+A$	+	1	0	1	1				
		1	1	0	1	1			
rechtsschieben		1	1	1	0	1	1		arithmetisch verschieben!
$-A$	+	0	1	0	1				
		0	0	1	1	1	1		
rechtsschieben		0	0	0	1	1	1	1	

Ineffizienzen im Radix-2-Booth-Verfahren

Beim Auftreten von isolierten Einsen im Multiplikator verhält sich der Algorithmus ineffizient.

Beispiel: 00101 ergibt $01\bar{1}1\bar{1}$; 2 Additionen ersetzt durch 2 Additionen und 2 Subtraktionen.

Um isolierte Einsen im Multiplikator zu erkennen, müssen statt Paaren Tripel von Bits betrachtet werden. Paare von Bits werden gemeinsam umcodiert (Radix-4-Booth-Verfahren).

Für jedes Paar von Bits wird nur noch höchstens eine Addition oder eine Subtraktion benötigt.

Bei synchroner Implementierung wird durch Addition bzw. Subtraktion eines Nulloperanden erreicht, dass für jedes Paar von Bits genau eine Addition oder Subtraktion durchgeführt wird.

Radix-4-Booth-Verfahren für vorzeichenlose ganze Binärzahlen

B_{2i+1}	B_{2i}	B_{2i-1}	B'_{2i+1}	B'_{2i}	Aktion	Bemerkung
0	0	0	0	0	$+\,0;\gg 2$	Folge von Nullen
0	0	1	0	1	$+\,A;\gg 2$	Ende einer Folge von Einsen
0	1	0	0	1	$+\,A;\gg 2$	isolierte 1
0	1	1	1	0	$+\,2\times A;\gg 2$	Ende einer Folge von Einsen
1	0	0	$\bar{1}$	0	$-\,2\times A;\gg 2$	Anfang einer Folge von Einsen
1	0	1	0	$\bar{1}$	$-\,A;\gg 2$	isolierte Null
1	1	0	0	$\bar{1}$	$-\,A;\gg 2$	Anfang einer Folge von Einsen
1	1	1	0	0	$-\,0;\gg 2$	Folge von Einsen

Für ungerades m muss eine Vorzeichenstelle $B_m = 0$ ergänzt werden,
für gerades m sogar zwei identische Vorzeichenstellen $B_{m+1} = B_m = 0$.

Ebenso muss wieder $B_{-1} = 0$ gesetzt werden.

Radix-4-Booth-Verfahren für 2-Komplement-Darstellung

A			01	00	01			Multiplikand 17
B	$\times$		11	01	11			Multiplikator -9
B'			$0\bar{1}$	10	$0\bar{1}$			umcodierter Multiplikator
initialisieren			00	00	00			
$-A$	+		10	11	11			
			10	11	11			
rechtsschieben		1	11	10	11	11		arithmetischer Doppel-Shift!
$+2 \times A$	+	0	10	00	10			
			01	11	01	11		
rechtsschieben			00	01	11	01	11	arithmetischer Doppel-Shift!
$-A$	+		10	11	11			
			11	01	10	01	11	-153

Liegt der Multiplikator in 2-Komplement-Darstellung mit ungerader Stellenzahl m vor, so wird das Vorzeichen verdoppelt, $B_m = B_{m-1}$.

Radix-2^k-Booth-Verfahren

Radix-4-Booth generiert weniger Teilprodukte als Radix-2-Booth,
aber dennoch in manchen Fällen mehr als gewöhnliche sequentielle Multiplikation.

Beispiel: 001010 ergibt $010\bar{1}\bar{1}0$.

Codiert man Tripel benachbarter Bits des Multiplikators gemeinsam um,
ergibt sich ein Radix-8-Booth-Verfahren.

Interpretiert man ein entstandenes Tripel 011 als Faktor 3
und stellt neben A, $2 \times A$ und $4 \times A$ auch $3 \times A$ bereit (was etwas aufwendiger zu berechnen ist),
kommt man mit höchstens $\lceil (m+1)/3 \rceil$ Teilprodukten aus.

Das synchron implementierte Radix-2^k-Booth-Verfahren benötigt wie die gewöhnliche
Radix-2^k-Multiplikation genau $\lceil (m+1)/k \rceil$ Additionen/Subtraktionen.

Statt der Teilprodukte $h \times A$ für $h = 0, 1, \ldots, 2^k - 1$
werden jedoch nur die für $h = 0, 1, \ldots, 2^{k-1}$ benötigt.

In der Praxis werden Verfahren mit Basis 16 und höher selten benutzt, da die Einzelschritte immer teurer werden und die Zahl der auszuführenden Schritte nur noch langsam abnimmt.

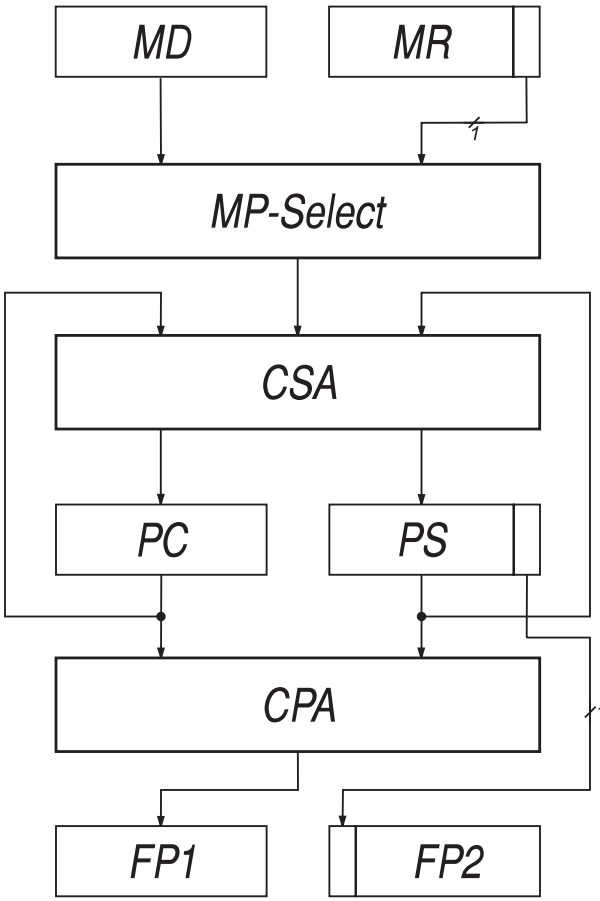
Minimierung der Anzahl der Teilprodukte

Mit Hilfe der sogenannten *kanonischen Umcodierung* kann ein ternär codierter Multiplikator mit einer maximalen Anzahl von Nullziffern hergestellt werden.

Wegen der vom Wert des Multiplikators abhängigen Anzahl von Additionen/Subtraktionen ist ein darauf basierender Multiplizierer für eine synchrone Umgebung jedoch meist nicht geeignet.

Entsprechendes gilt für die Technik, den Multipliziervorgang abzubrechen, sobald alle noch nicht verarbeiteten Stellen des Multiplikators Null sind.

Sequentieller Multiplizierer mit CSA



Eine Beschleunigung der Akkumulierung eines Teilprodukts ergibt sich durch Verwendung eines CSA statt eines CPA.

Zur Berechnung des abschließenden Ergebnisses ist entweder ein CPA nötig oder der CSA muss mehrfach durchlaufen werden (entspricht einem Ripple-Carry-Addierer).

Für Radix-2^k-Multiplikation müssen zusätzlich die niederwertigen k Stellen des CSA als CPA ausgeführt werden, oder es muss ein k-Bit-CPA dem CSA nachgeschaltet werden.

Für umcodierte Multiplikatoren muss der CSA auch Subtraktionen durchführen können.

Statt einen CPA zu verwenden, können wir den Multiplikator links um l Nullbits verlängern und damit aus der Carry-Save-Darstellung der höherwertigen l Stellen des Ergebnisses mittels des CSA die kanonische Binärdarstellung errechnen.

Sequentielle Multiplikation mit CSA (Beispiel)

7 =	00111	Multiplikand
15 =	01111	Multiplikator
	00000	initiale Teilsumme
	00000	initialer Teilübertrag
	00111	+ 1. Vielfaches
	00111	1. Teilsumme
	00000	1. Teilübertrag
	000111	Teilsumme rechtsschieben
	00111	+ 2. Vielfaches
	00000	+ 1. Teilübertrag
	001001	2. Teilsumme
	00011	2. Teilübertrag
	0001001	Teilsumme rechtsschieben
	00111	+ 3. Vielfaches
	00011	+ 2. Teilübertrag
	0011001	3. Teilsumme
	00011	3. Teilübertrag
	00011001	Teilsumme rechtsschieben
	00111	+ 4. Vielfaches
	00011	+ 3. Teilübertrag
	00111001	4. Teilsumme
	00011	4. Teilübertrag
	000111001	Teilsumme rechtsschieben
	00000	+ 5. Vielfaches
	00011	+ 4. Teilübertrag
	000001001	5. Teilsumme
	00011	5. Teilübertrag
	0000001001	Teilsumme rechtsschieben
	00011	propagiere Restübertrag
105 =	001101001	

Vergleich von sequentiellen Multiplizierern

		ohne CSA			mit CSA		
$l = m$	CPA	t	a	$a \times t \times 10^{-4}$	t	a	$a \times t \times 10^{-4}$
16	RCLA	144	608	8,76	103	1040	10,71
	MSBCLA	208	408	8,49	107	840	8,99
	SRCLA	208	440	9,15	107	872	9,33
	2-Level-Carry-Skip	240	348	8,35	109	780	8,50
	Carry-Completion	240	416	9,98	109	848	9,24
	BCLA	272	348	9,47	111	780	8,66
	SBCLA	272	416	11,32	111	848	9,41
	1-Level-Carry-Skip	288	288	8,29	112	720	8,06
	Carry-Ripple	544	272	14,80	128	704	9,01
32	SRCLA	416	1000	41,60	203	1864	37,83
	MSBCLA	416	1040	43,26	203	1904	38,65
	Carry-Completion	544	832	45,26	207	1696	35,11
	RCLA	672	796	53,49	211	1660	35,02
	SBCLA	672	844	56,72	211	1708	36,04
	2-Level-Carry-Skip	736	668	49,16	213	1532	32,63
	BCLA	800	696	55,68	215	1460	31,39
	1-Level-Carry-Skip	832	576	47,92	216	1640	35,42
	Carry-Ripple	2112	544	144,89	256	1308	33,48
64	SRCLA	832	2224	185,04	395	3952	156,10
	MSBCLA	1088	1760	191,49	399	3488	139,17
	Carry-Completion	1216	1664	202,34	401	3392	136,02
	RCLA	1344	1592	213,96	403	3320	133,80
	SBCLA	1600	1740	278,40	407	3468	141,15
	2-Level-Carry-Skip	1984	1332	264,27	413	3060	126,38
	1-Level-Carry-Skip	2046	1152	235,70	414	2880	119,23
	BCLA	2112	1512	319,33	415	3240	134,46
	Carry-Ripple	8320	1088	905,21	512	2816	144,18

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

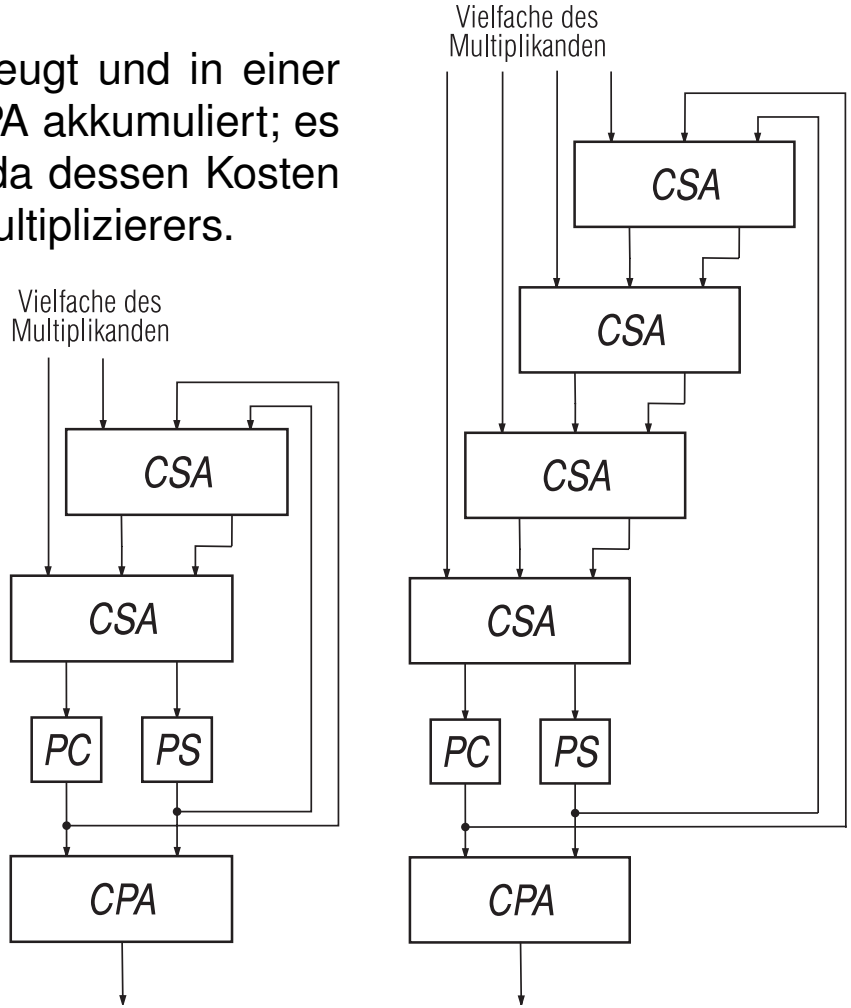
FSU Jena

Thema: Parallele Multiplizierer

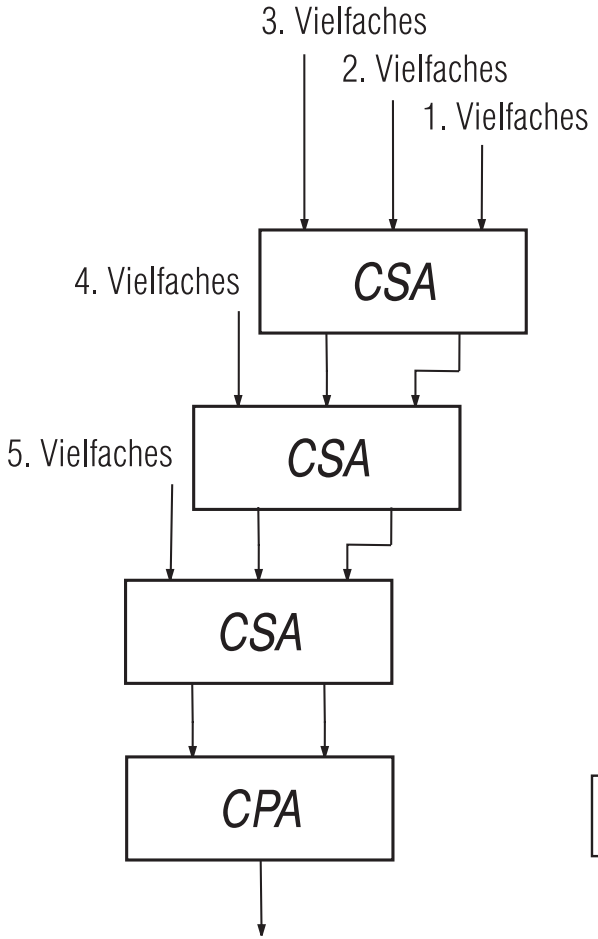
Parallele Multiplizierer mit CSA

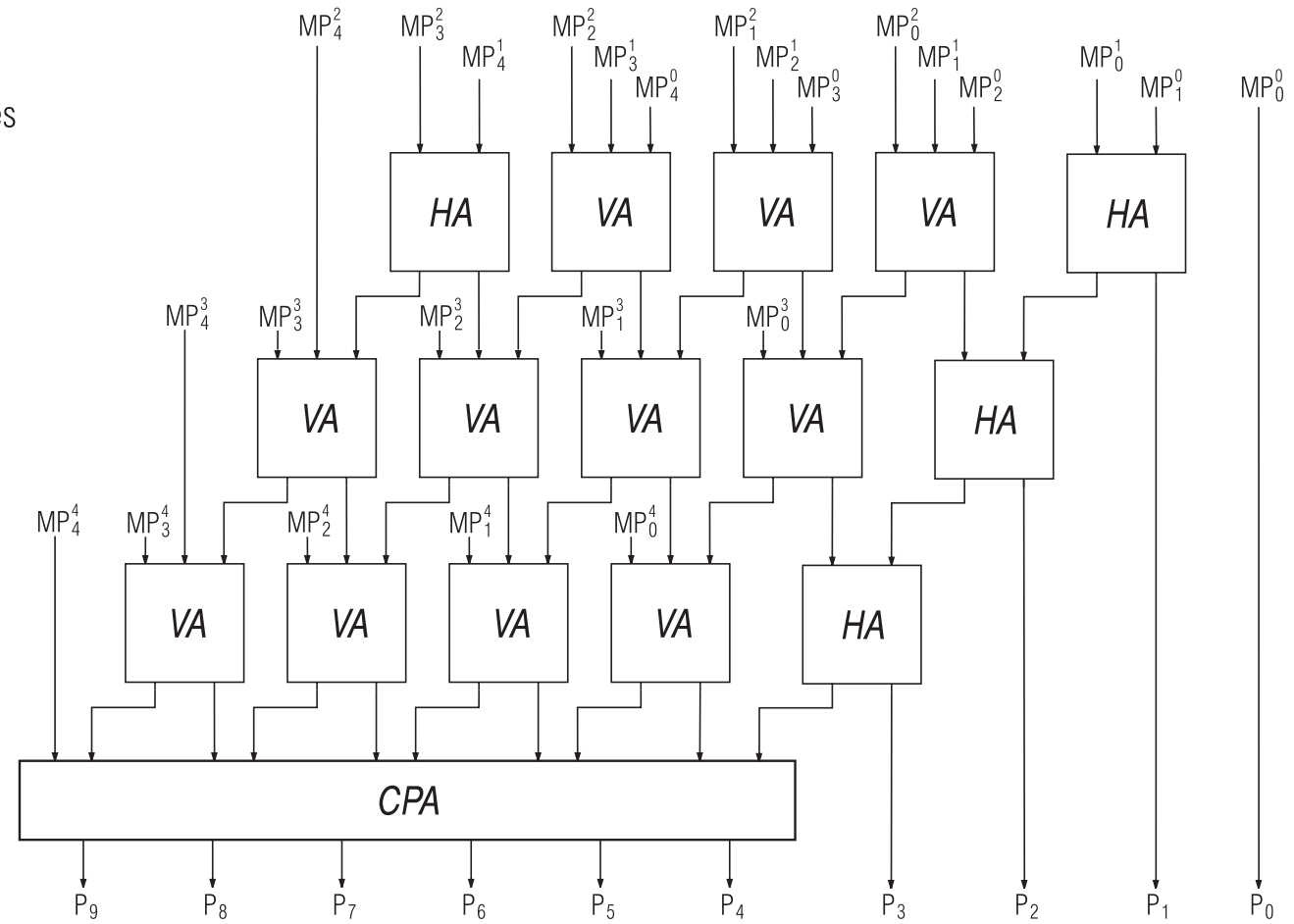
In parallelen Multiplizierern werden Teilprodukte simultan erzeugt und akkumuliert; dies geschieht in der Regel mit Hilfe mehrerer CSA.

Im Extremfall werden alle Teilprodukte simultan erzeugt und in einer nicht-iterativen CSA-Struktur mit abschließendem CPA akkumuliert; es kann dann ein sehr schneller CPA benutzt werden, da dessen Kosten gering sind im Vergleich zum Gesamtaufwand des Multiplizierers.



Akkumulation im Array-Multiplizierer





Vergleich von Array-Multiplizierern

$l = m$	CPA	t	a	$a \times t \times 10^{-4}$
16	RCLA	49	4012	19,66
	MSBCLA	53	3812	20,20
	SRCLA	53	3844	20,37
	2-Level-Carry-Skip	55	3752	20,63
	BCLA	57	3752	21,39
	SBCLA	57	3820	21,77
	1-Level-Carry-Skip	58	3692	21,41
	Carry-Ripple	74	3676	27,20
	Seriell	91	3474	31,61
32	SRCLA	101	15972	161,31
	MSBCLA	101	16012	161,72
	RCLA	109	15768	171,87
	SBCLA	109	15816	172,39
	2-Level-Carry-Skip	111	15640	173,60
	BCLA	113	15668	177,05
	1-Level-Carry-Skip	114	15548	177,24
	Carry-Ripple	154	15516	238,95
	Seriell	187	15090	282,18
64	SRCLA	197	64908	1278,68
	MSBCLA	201	64444	1295,32
	RCLA	205	64276	1317,66
	SBCLA	209	64424	1346,46
	2-Level-Carry-Skip	215	64016	1376,34
	1-Level-Carry-Skip	216	63836	1378,86
	BCLA	217	64196	1393,05
	Carry-Ripple	314	63772	2002,24
	Seriell	379	62898	2383,83

Array-Multiplizierer

In Array-Multiplizierern wird die Erzeugung der Teilprodukte und ihre Akkumulierung simultan vorgenommen; dadurch entfällt der zweifache Aufwand für die Steuerung separater Einheiten.

Jede Zeile des Array-Multiplizierers berechnet ein neues Teilprodukt (oder einen Ausschnitt daraus) und addiert es zur bisherigen Zwischensumme.

Array-Multiplizierer werden in der Regel in Pipelining-Technik ausgeführt:

Der Abschluss des Array-Multiplizierers durch einen CPA begrenzt den Durchsatz.

Der CPA wird deshalb durch weitere Zeilen ohne Propagierung innerhalb einer Zeile ersetzt.

Für die Realisierung der Zellen der zusätzlichen Zeilen genügen Halbaddierer (statt der Volladdierer in den oberen Zeilen).

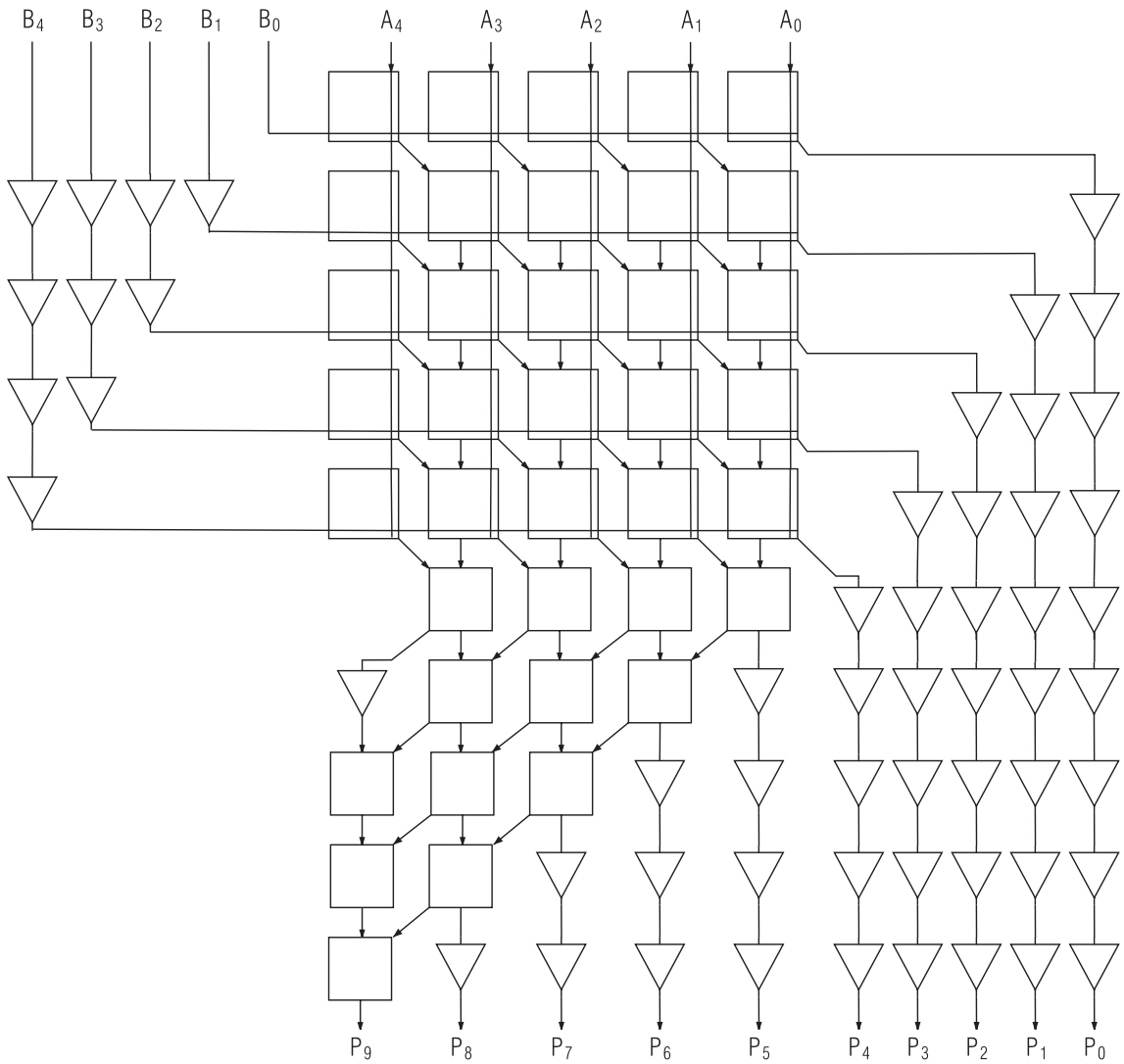
Zusätzlich erfordert das Pipelining ein Array von Puffern für die getaktete Weitergabe der Multiplikatorbits und der Produktbits.

Array-Multiplizierer in Pipelining-Technik

Die oberen und linken Randzellen des rechteckigen Multiplikationsarrays enthalten nur jeweils ein AND-Gatter zur Multiplikation zweier Bits.
Jede andere Zelle der zweiten Zeile beinhaltet zusätzlich einen Halbaddierer.

Jede weitere Zelle des rechteckigen Multiplikationsarrays enthält das AND-Gatter sowie einen Volladdierer.

Jede rechteckige Zelle der linken Spalte des dreieckigen Abschlussarrays enthält ein OR-Gatter zur Addition zweier Bits ohne Übertrag.
Jede weitere rechteckige Zelle des Abschlussarrays enthält einen Halbaddierer.



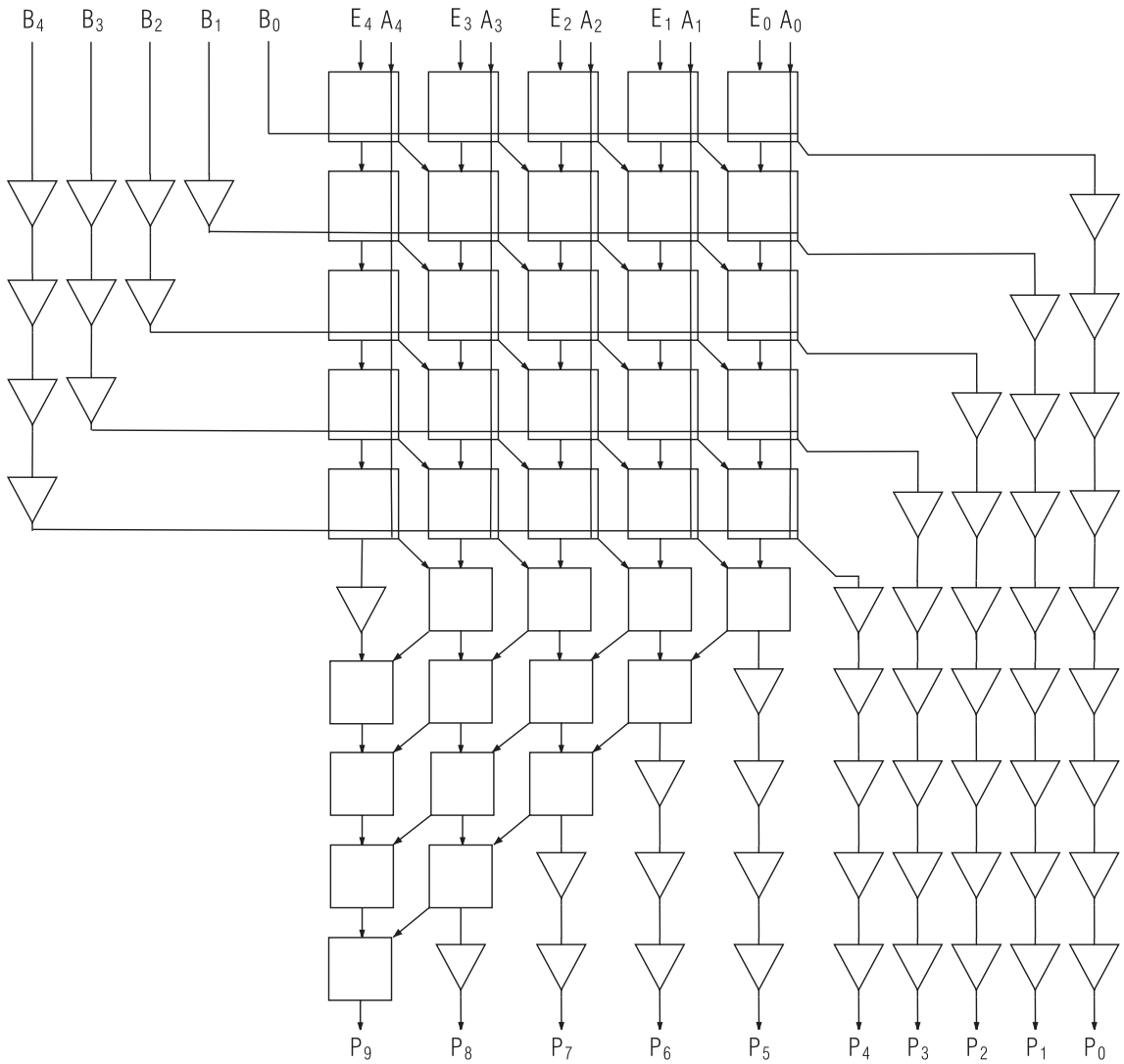
Erweiterung zur Multiply-Add-Einheit

Die oberen und linken Randzellen des rechteckigen Multiplikationsarrays enthalten jeweils ein AND-Gatter zur Multiplikation zweier Bits sowie einen Halbaddierer.

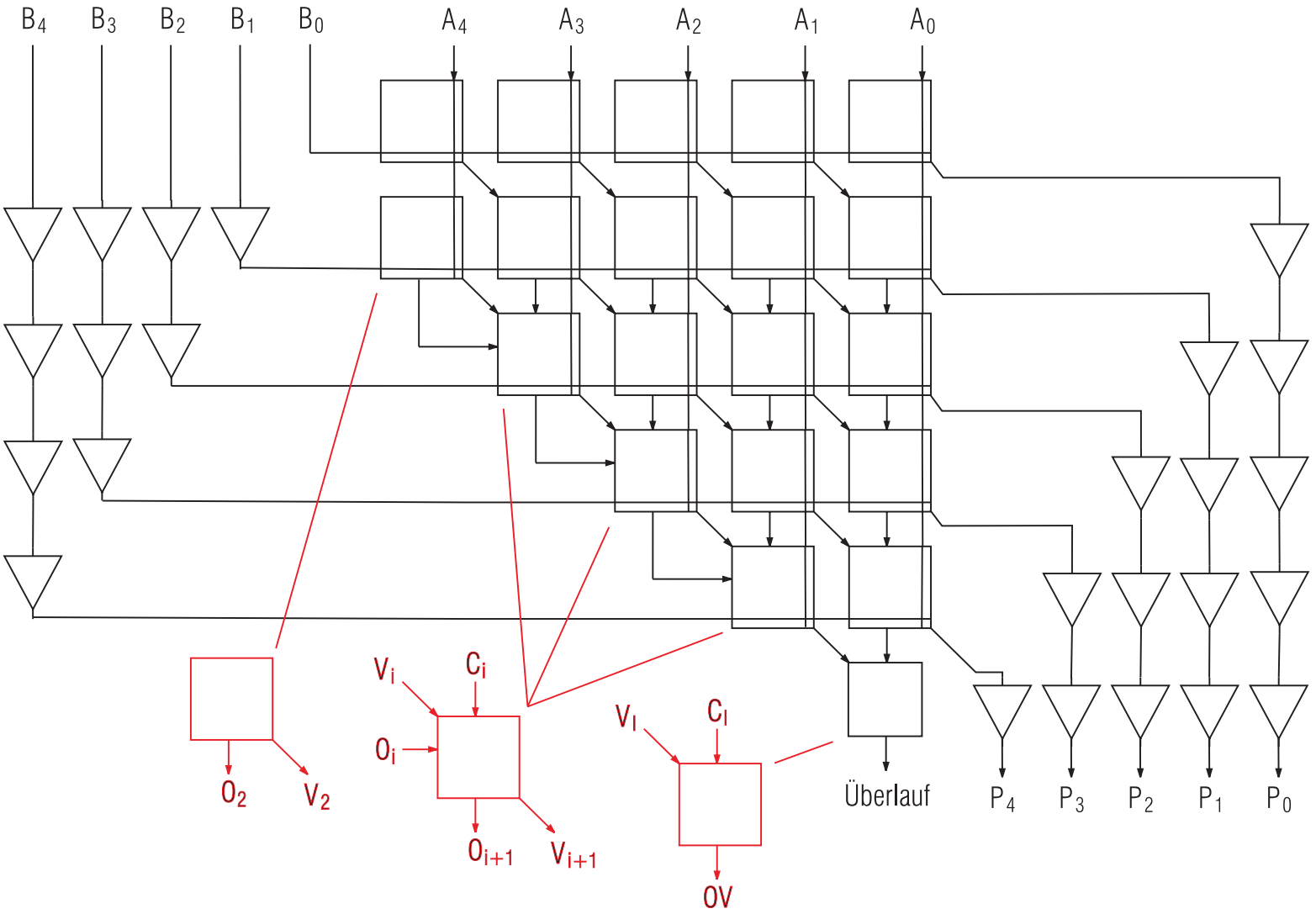
Jede weitere Zelle des rechteckigen Multiplikationsarrays enthält das AND-Gatter sowie einen Volladdierer.

Jede rechteckige Zelle der linken Spalte des dreieckigen Abschlussarrays enthält ein OR-Gatter zur Addition zweier Bits ohne Übertrag.

Jede weitere rechteckige Zelle des Abschlussarrays enthält einen Halbaddierer.



Array-Multiplizierer mit Überlauferkennung

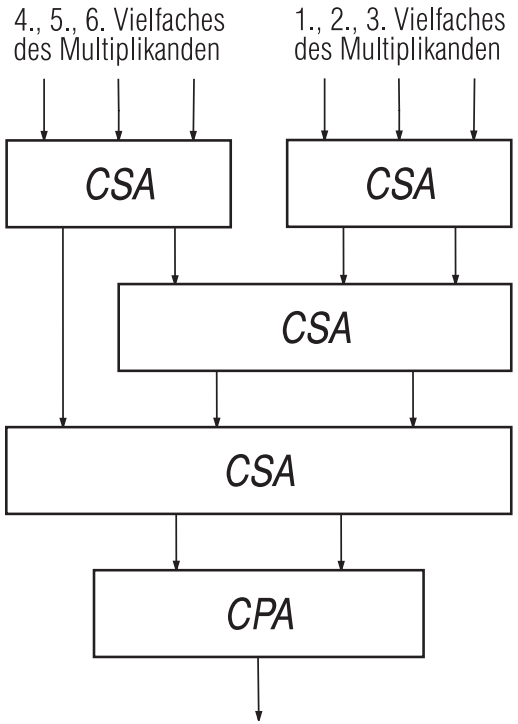


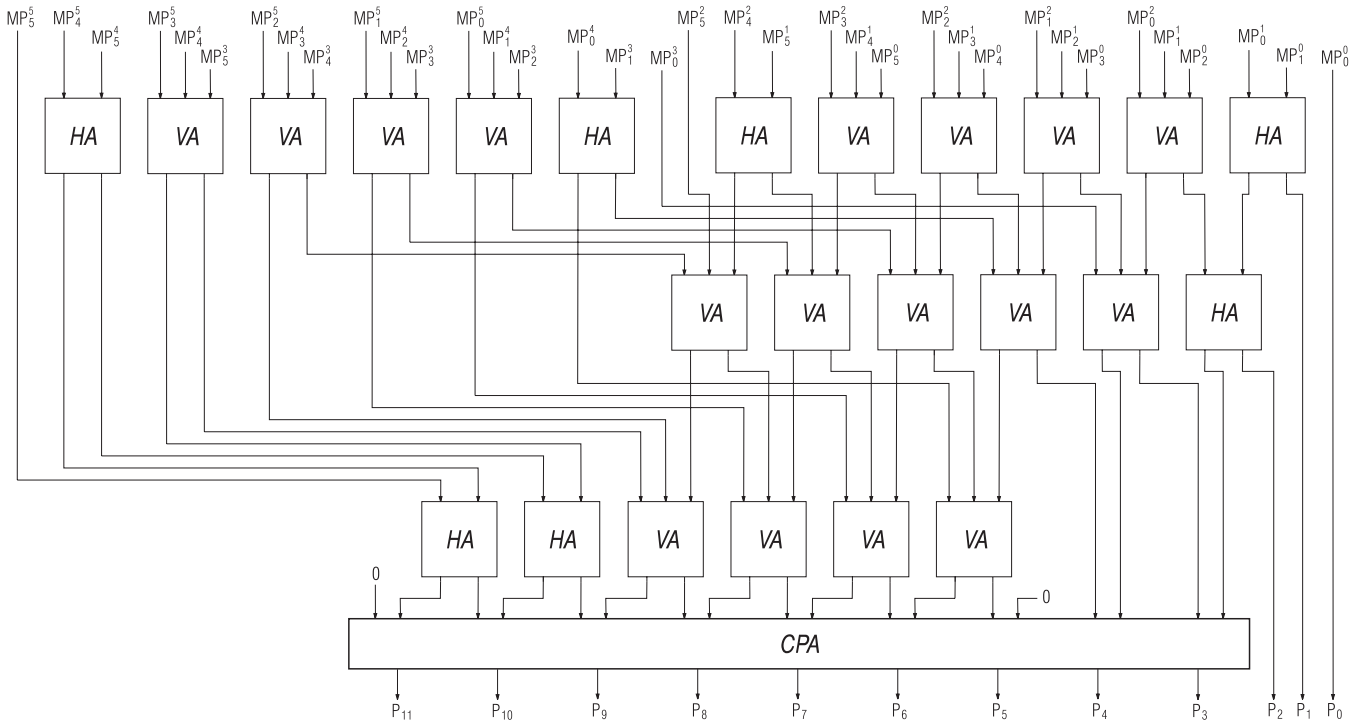
$$O_2 = A_{l-1} \qquad V_2 = A_{l-1} \wedge B_1 \qquad OV = V_l \vee C_l$$
$$O_{i+1} = O_i \vee A_{l-i} \qquad V_{i+1} = V_i \vee C_i \vee O_{i+1} \wedge B_i$$

Baum-Multiplizierer

Statt einer Kaskade von CSA-Addierern kann auch ein CSA-Baum benutzt werden.

Die Latenz eines Baum-Multiplizierers ist kleiner als die eines Array-Multiplizierers;
dies wird durch einen höheren Hardware-Aufwand erkauft
(breiterer CPA, irregulärer Aufbau, komplexere Leitungsführung).



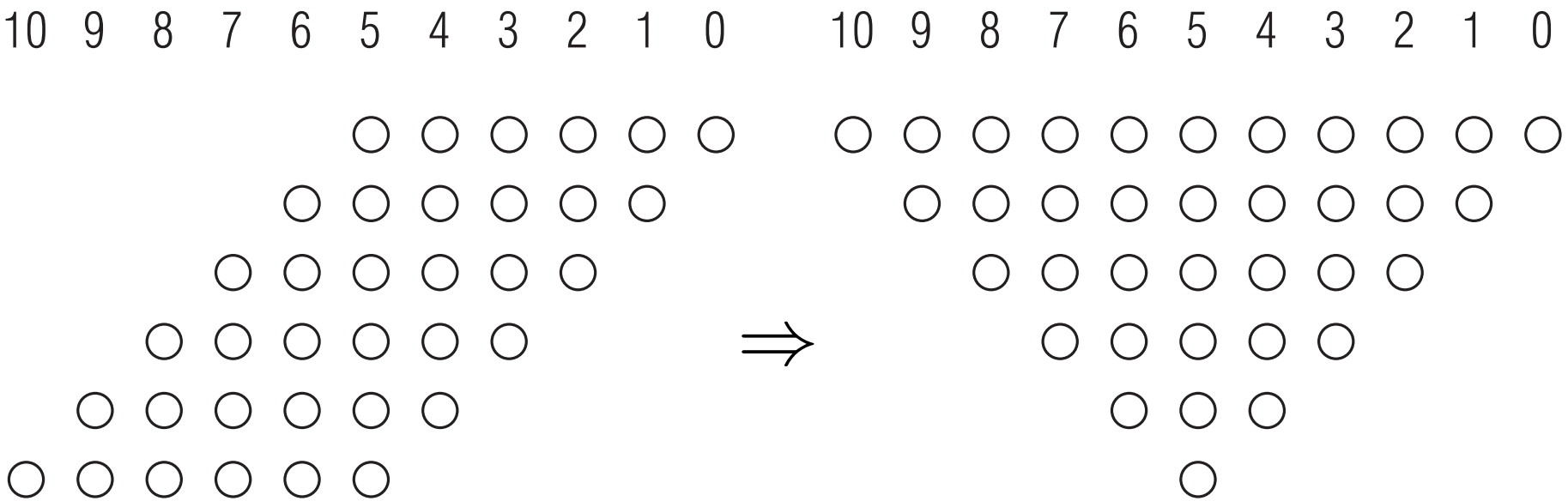


Effiziente Summierung der Teilprodukte

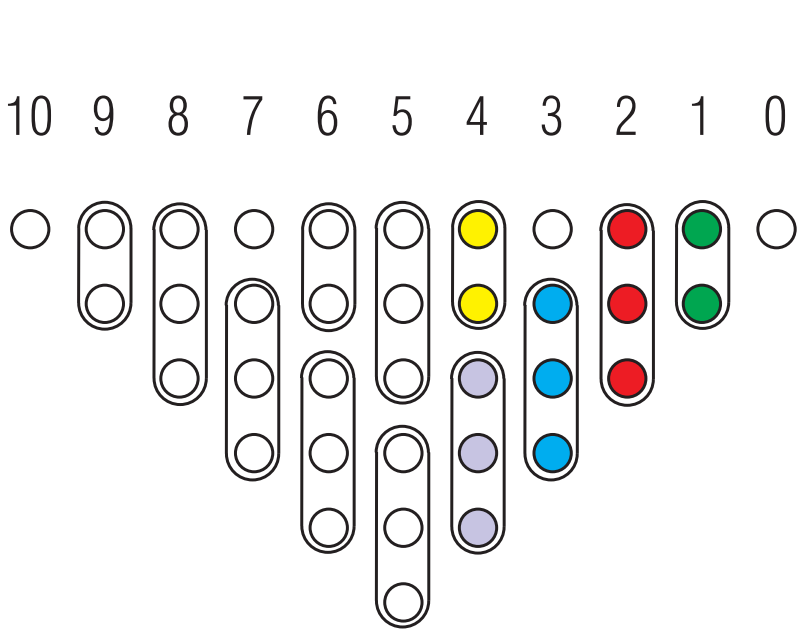
Durch geschickte Beschaltung kann die Anzahl der Volladdierer eines CSA-Baumes zur Summierung der Teilprodukte verringert werden (nicht aber die Anzahl der Stufen).

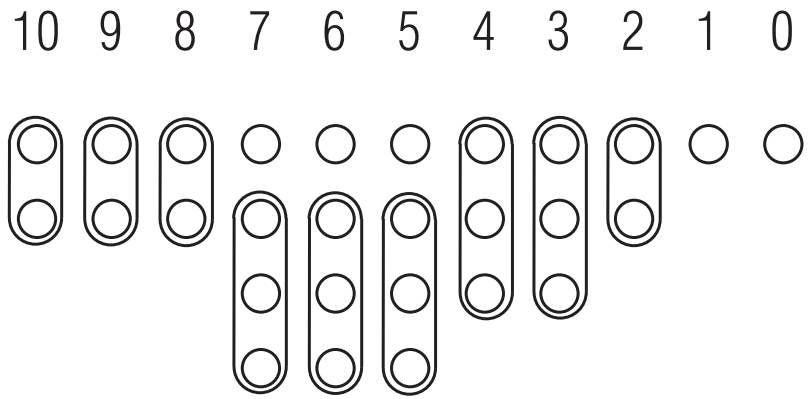
Außerdem können einige Volladdierer durch Halbaddierer ersetzt werden.

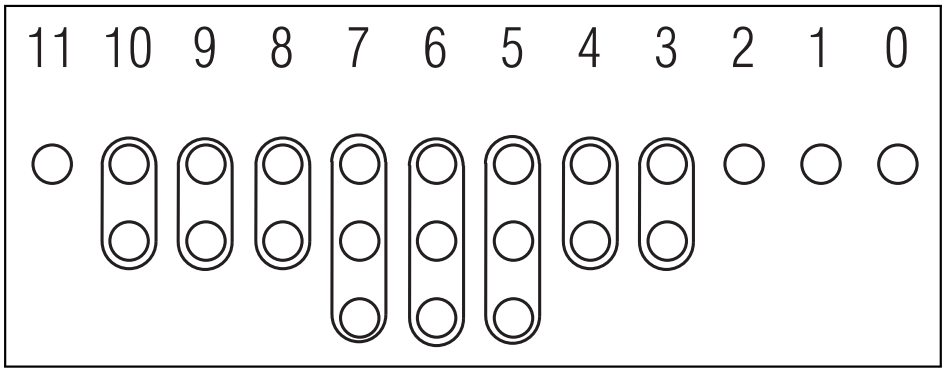
Initiales Bitschema für einen (6×6)-Multiplizierer:

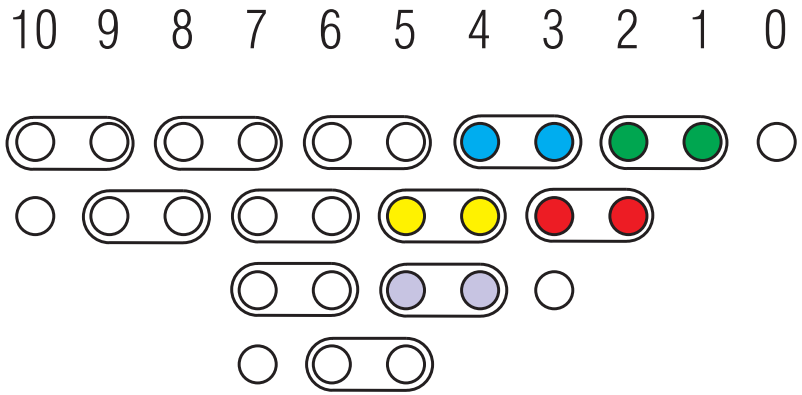


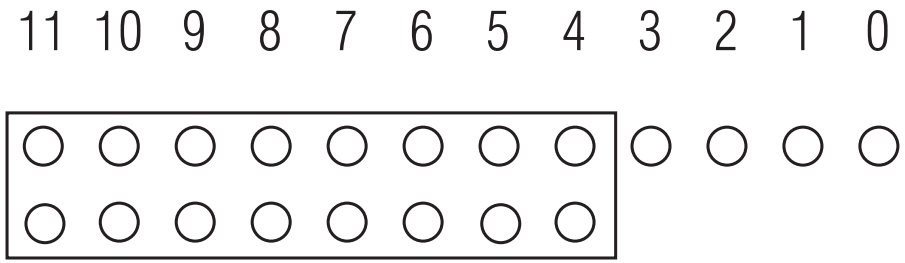
Frühestmögliche Reduktion: Wallace-Multiplizierer











Vergleich von Wallace-Multiplizierern

$l = m$	CPA	t	a	$a \times t \times 10^{-4}$
16	RCLA	25	4642	11,61
	MSBCLA	29	4442	12,88
	SRCLA	29	4472	12,97
	2-Level-Carry-Skip	31	4382	13,58
	BCLA	33	4382	14,46
	SBCLA	33	4450	14,67
	1-Level-Carry-Skip	34	4322	14,69
	Carry-Ripple	50	4306	21,53
	Seriell	67	4104	27,50
32	SRCLA	32	17274	55,28
	MSBCLA	32	17314	55,40
	RCLA	38	17070	64,87
	SBCLA	40	17118	68,47
	2-Level-Carry-Skip	42	16942	71,56
	BCLA	44	16970	74,67
	1-Level-Carry-Skip	45	16850	75,83
	Carry-Ripple	85	16818	142,95
	Seriell	118	16392	193,43
64	SRCLA	35	67554	236,44
	MSBCLA	39	67090	261,65
	RCLA	43	66922	287,76
	SBCLA	47	67070	315,23
	2-Level-Carry-Skip	53	66662	353,30
	1-Level-Carry-Skip	54	66482	359,00
	BCLA	55	66842	367,63
	Carry-Ripple	152	66418	1009,55
	Seriell	217	65544	1422,30

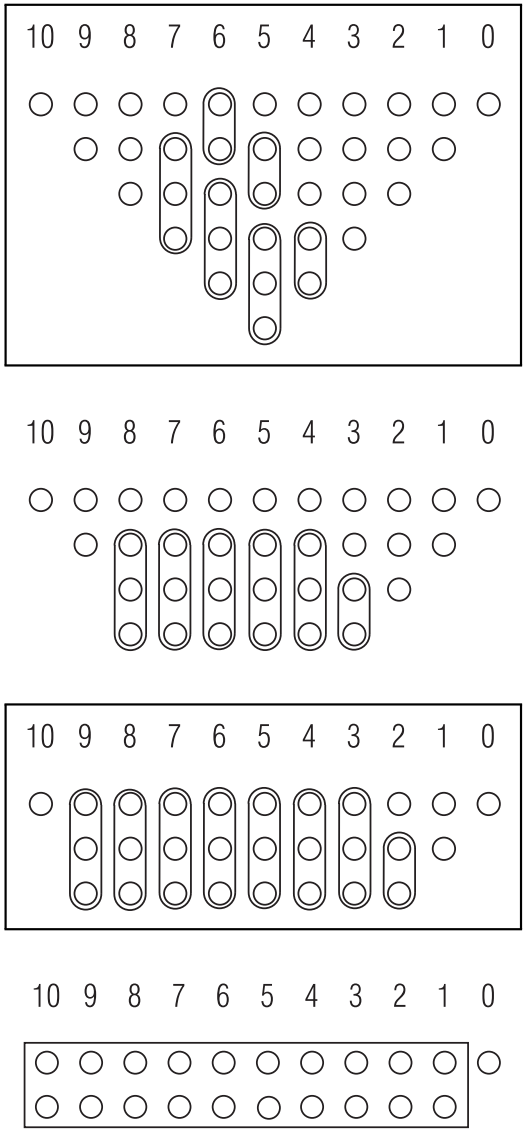
Einsparungen durch verzögerte Reduktion: Dadda-Multiplizierer

Im Dadda-Multiplizierer werden Volladdierer und Halbaddierer so eingesetzt, dass die Anzahl der Ergebnisbits derselben Gewichtung möglichst nahe an den bezüglich der Stufenzahl eines CSA-Baumes optimalen Wert herankommen.

Die Anzahl der Ergebnisbits sollte damit am besten einen der Werte $3, 4, 6, 9, 13, 19, \ldots$ annehmen.

Aufwand (ohne CPA) für parallele (6×6)-Multiplizierer:

Standard-CSA-Baum:	4×12 VA
Zwei niederwertigste Stellen direkt in CPA geleitet:	4×10 VA
Wallace-Multiplizierer (schnellster Multiplizierer):	16 VA, 13 HA
Dadda-Multiplizierer:	15 VA, 5 HA
Optimierter Array-Multiplizierer:	19 VA, 5 HA



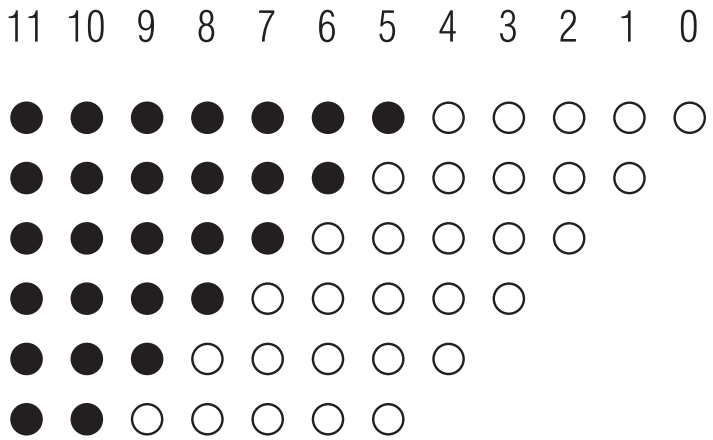
Summierung von Teilprodukten mit Vorzeichen

- In Vorzeichen/Betrag-Darstellung läuft eine $(l \times m)$-Multiplikation praktisch wie eine vorzeichenlose $((l-1) \times (m-1))$-Multiplikation ab.

  Das Vorzeichen wird abgetrennt und separat verarbeitet: $P_{n-1} = A_{l-1} \oplus B_{m-1}$
- In 1- oder 2-Komplement-Darstellung müssen negative Teilprodukte vor ihrer Summierung durch Vervielfachung des Vorzeichens auf die Länge des Ergebnisses erweitert werden.

  Da es wenig Sinn macht, zur Laufzeit zwischen positiven und negativen Teilprodukten zu unterscheiden, erfolgt die Vorzeichenvervielfachung auch für positive Teilprodukte.

  Die Anzahl der zu addierenden Bits steigt hierdurch signifikant an.



Summierung von Teilprodukten in 2-Komplement-Darstellung (Beispiel)

A						0	1	0	1	1	0	Multiplikand 22
B				$\times$		0	0	1	0	1	1	Multiplikator 11
B'						0	1	0	$\bar{1}$	0	$\bar{1}$	umcodierter Multiplikator
1	**1**	**1**	**1**	**1**	**1**	**1**	0	1	0	1	0	
0	**0**	**0**	**0**	**0**	**0**	0	0	0	0	0		
1	**1**	**1**	**1**	**1**	0	1	0	1	0			
0	**0**	**0**	**0**	0	0	0	0	0				
0	**0**	**0**	1	0	1	1	0					
0	**0**	0	0	0	0	0						
0	0	0	0	1	1	1	1	0	0	1	0	Ergebnis 242

Umcodieren der Teilprodukte

Durch Umcodieren der Teilprodukte kann die Anzahl der zusätzlichen Bits stark verringert werden. Beispielsweise gilt mit beliebigem $S \in \{0, 1\}$:

$$SSSSSSSZ_4Z_3Z_2Z_1Z_0 \equiv 000000\bar{S}Z_4Z_3Z_2Z_1Z_0 \pmod{2^{12}}$$

Durch einen Trick lässt sich die Ziffer $\bar{S}$ ohne Verwendung eines ternären Codes darstellen:

- Komplementieren des Vorzeichenbits S liefert die Ziffer $(1 - S)$, die durch Addieren einer zusätzlichen 1 an der Position des Vorzeichenbits in den Wert $(2 - S)$ überführt wird.
- Der Wert $+2$ kann eine Stelle höher als dieselbe zu addierende 1 für das folgende Teilprodukt interpretiert werden.
- Die gewünschten Ziffern $\bar{S}$ ergeben sich also durch Komplementierung des Vorzeichenbits aller Teilprodukte, Addition einer einzigen 1 an der Position des Vorzeichens des am niedrigsten gewichteten Teilprodukts und Komplementieren des Ergebnisvorzeichenbits.

Umcodieren der Teilprodukte (Beispiel)

A						0	1	0	1	1	0	Multiplikand 22
B				$\times$		0	0	1	0	1	1	Multiplikator 11
B'						0	1	0	$\bar{1}$	0	$\bar{1}$	umcodierter Multiplikator
						1						
						0	0	1	0	1	0	
					1	0	0	0	0	0		
				0	0	1	0	1	0			
			1	0	0	0	0	0				
		1	1	0	1	1	0					
	1	0	0	0	0	0						
1	0	0	0	1	1	1	1	0	0	1	0	vorläufiges Ergebnis
0	0	0	0	1	1	1	1	0	0	1	0	Ergebnis 242

Direkte Multiplikation in $\mathrm{Int}_2(l)$ nach Baugh-Wooley

Das Multiplikationsschema für $\mathrm{UInt}_2(l)$ wird zu einem Multiplikationsschema für $\mathrm{Int}_2(l)$ abgeändert, in dem die negativen Gewichte der beiden Vorzeichenbits berücksichtigt sind.

Darauf werden folgende Regeln zur Vereinfachung systematisch angewandt:

- $-A_{l-1}B_j = A_{l-1}(1 - B_j) - A_{l-1} = A_{l-1}\overline{B}_j - A_{l-1}$.
- $-A_{l-1}$ in Spalte k wird ersetzt durch A_{l-1} in Spalte k und $-A_{l-1}$ in Spalte $k+1$.
- $-A_{l-1}$ in Spalte $2l-2$ wird ersetzt durch $\overline{A}_{l-1} - 1$.
- $-A_iB_{l-1} = (1 - A_i)B_{l-1} - B_{l-1} = \overline{A}_iB_{l-1} - B_{l-1}$.
- $-B_{l-1}$ in Spalte k wird ersetzt durch B_{l-1} in Spalte k und $-B_{l-1}$ in Spalte $k+1$.
- $-B_{l-1}$ in Spalte $2l-2$ wird ersetzt durch $\overline{B}_{l-1} - 1$.
- -2 in Spalte $2l-2$ wird ersetzt durch 1 in Spalte $2l-1$.

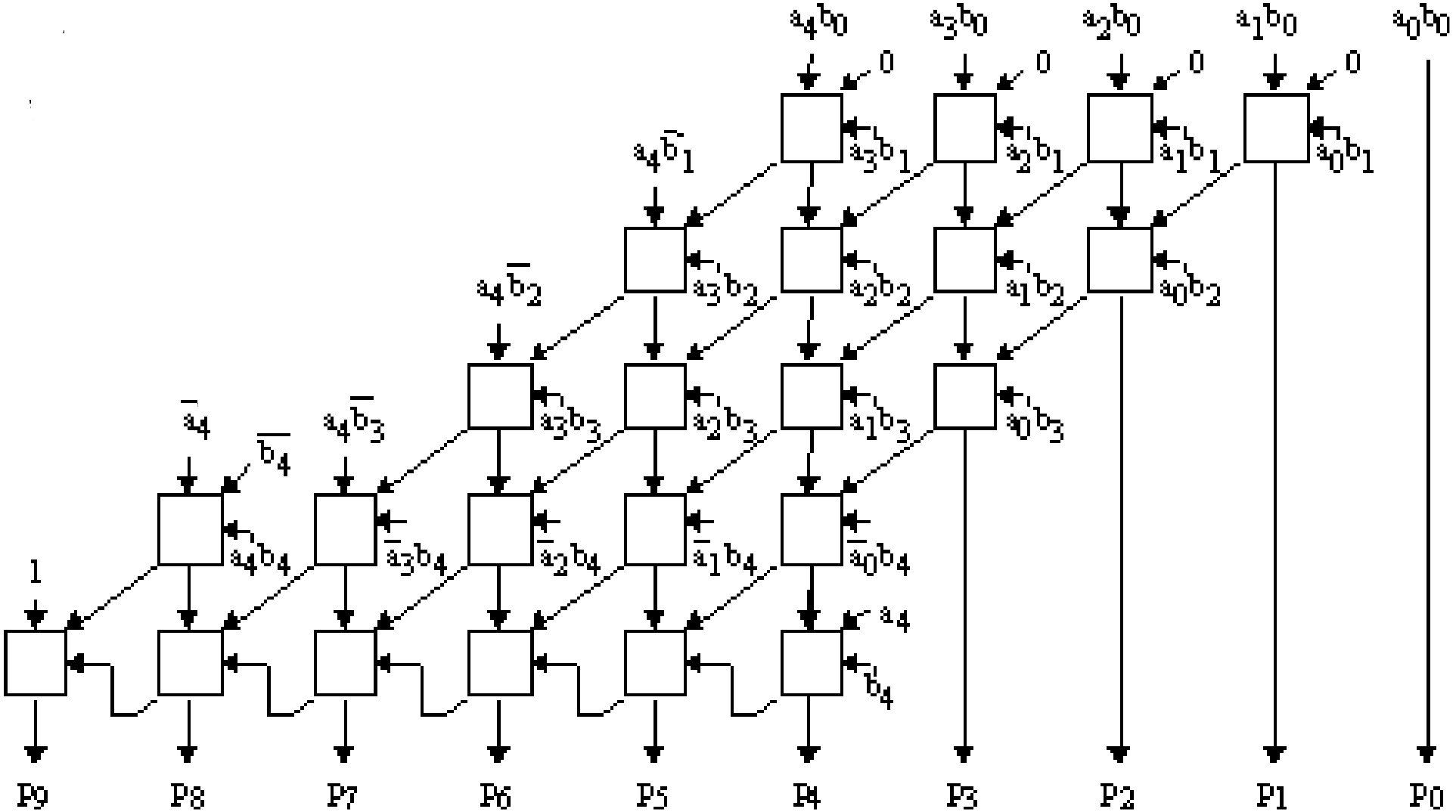
Transformationen im Baugh-Wooley-Multiplizierer

					$-A_4$	A_3	A_2	A_1	A_0
				$\times$	$-B_4$	B_3	B_2	B_1	B_0
					$-A_4B_0$	A_3B_0	A_2B_0	A_1B_0	A_0B_0
				$-A_4B_1$	A_3B_1	A_2B_1	A_1B_1	A_0B_1	
			$-A_4B_2$	A_3B_2	A_2B_2	A_1B_2	A_0B_2		
		$-A_4B_3$	A_3B_3	A_2B_3	A_1B_3	A_0B_3			
	A_4B_4	$-A_3B_4$	$-A_2B_4$	$-A_1B_4$	$-A_0B_4$				
P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

wird transformiert in

					$-A_4$	A_3	A_2	A_1	A_0
				$\times$	$-B_4$	B_3	B_2	B_1	B_0
					$A_4\overline{B}_0$	A_3B_0	A_2B_0	A_1B_0	A_0B_0
				$A_4\overline{B}_1$	A_3B_1	A_2B_1	A_1B_1	A_0B_1	
			$A_4\overline{B}_2$	A_3B_2	A_2B_2	A_1B_2	A_0B_2		
		$A_4\overline{B}_3$	A_3B_3	A_2B_3	A_1B_3	A_0B_3			
	A_4B_4	$\overline{A}_3B_4$	$\overline{A}_2B_4$	$\overline{A}_1B_4$	$\overline{A}_0B_4$				
	$\overline{A}_4$				A_4				
1	$\overline{B}_4$				B_4				
P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

Baugh-Wooley-Multiplizierer



Optimierung der Baugh-Wooley-Multiplikation

In baumartigen Multiplizierern erhöht der Baugh-Wooley-Algorithmus
die Anzahl der Einträge in der kritischen Spalte $l - 1$ und damit evtl. die Latenz.

Mit folgenden alternativen Regeln zur Vereinfachung wird dies vermieden:

- $-S = (1 - S) - 1 = \overline{S} - 1$.
- Alle auftretenden Werte von -1 können zusammengefasst und als Einsen in den Spalten l und $2l - 1$ wiedergegeben werden (in denen sie unschädlich sind).

Transformationen im optimierten Baugh-Wooley-Multiplizierer

					$-A_4$	A_3	A_2	A_1	A_0
				$\times$	$-B_4$	B_3	B_2	B_1	B_0
					$-A_4B_0$	A_3B_0	A_2B_0	A_1B_0	A_0B_0
				$-A_4B_1$	A_3B_1	A_2B_1	A_1B_1	A_0B_1	
			$-A_4B_2$	A_3B_2	A_2B_2	A_1B_2	A_0B_2		
		$-A_4B_3$	A_3B_3	A_2B_3	A_1B_3	A_0B_3			
	A_4B_4	$-A_3B_4$	$-A_2B_4$	$-A_1B_4$	$-A_0B_4$				
P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

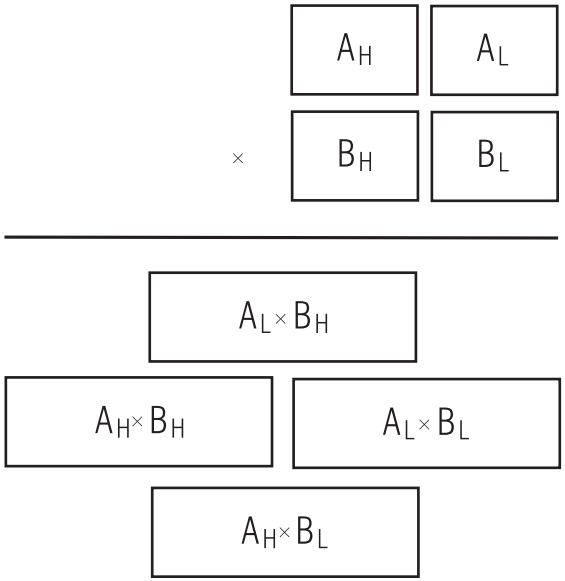
wird transformiert in

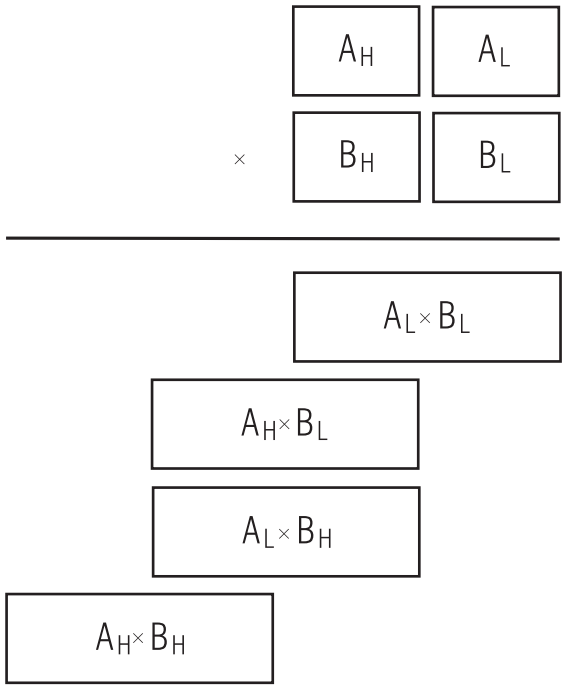
					$-A_4$	A_3	A_2	A_1	A_0
				$\times$	$-B_4$	B_3	B_2	B_1	B_0
1				1	$\overline{A_4B_0}$	A_3B_0	A_2B_0	A_1B_0	A_0B_0
				$\overline{A_4B_1}$	A_3B_1	A_2B_1	A_1B_1	A_0B_1	
			$\overline{A_4B_2}$	A_3B_2	A_2B_2	A_1B_2	A_0B_2		
		$\overline{A_4B_3}$	A_3B_3	A_2B_3	A_1B_3	A_0B_3			
	A_4B_4	$\overline{A_3B_4}$	$\overline{A_2B_4}$	$\overline{A_1B_4}$	$\overline{A_0B_4}$				
P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

Rekursiver Aufbau großer Multiplizierer

Ein $(2n \times 2n)$-Multiplizierer kann aus vier $(n \times n)$-Multiplizierern aufgebaut werden:

$$\begin{aligned} A \times B &= (A_H \times 2^n + A_L) \times (B_H \times 2^n + B_L) \\ &= A_H \times B_H \times 2^{2n} \\ &\quad + (A_H \times B_L + A_L \times B_H) \times 2^n \\ &\quad + A_L \times B_L \end{aligned}$$



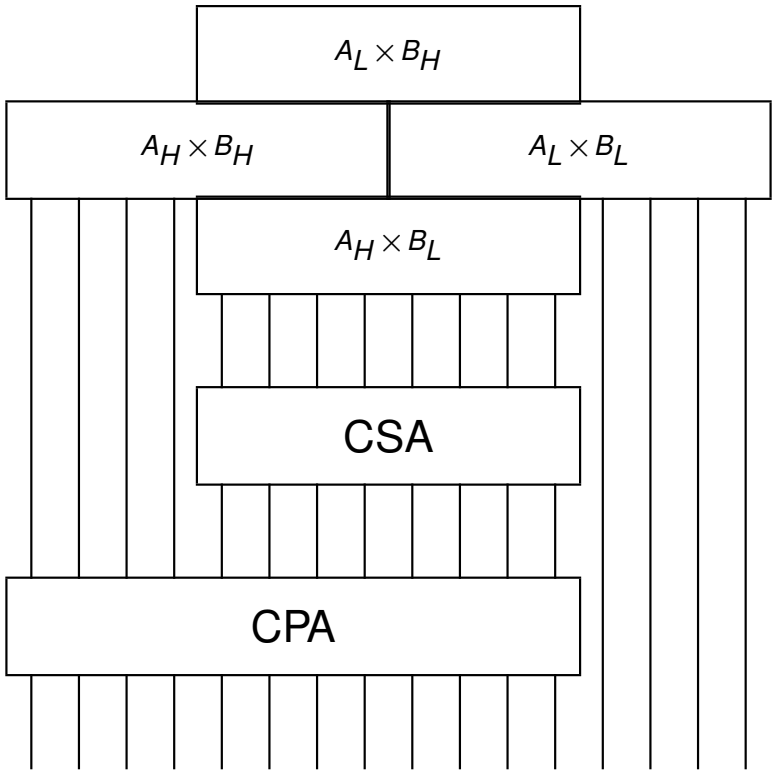


Optimierung durch Umordnung der Teilprodukte:

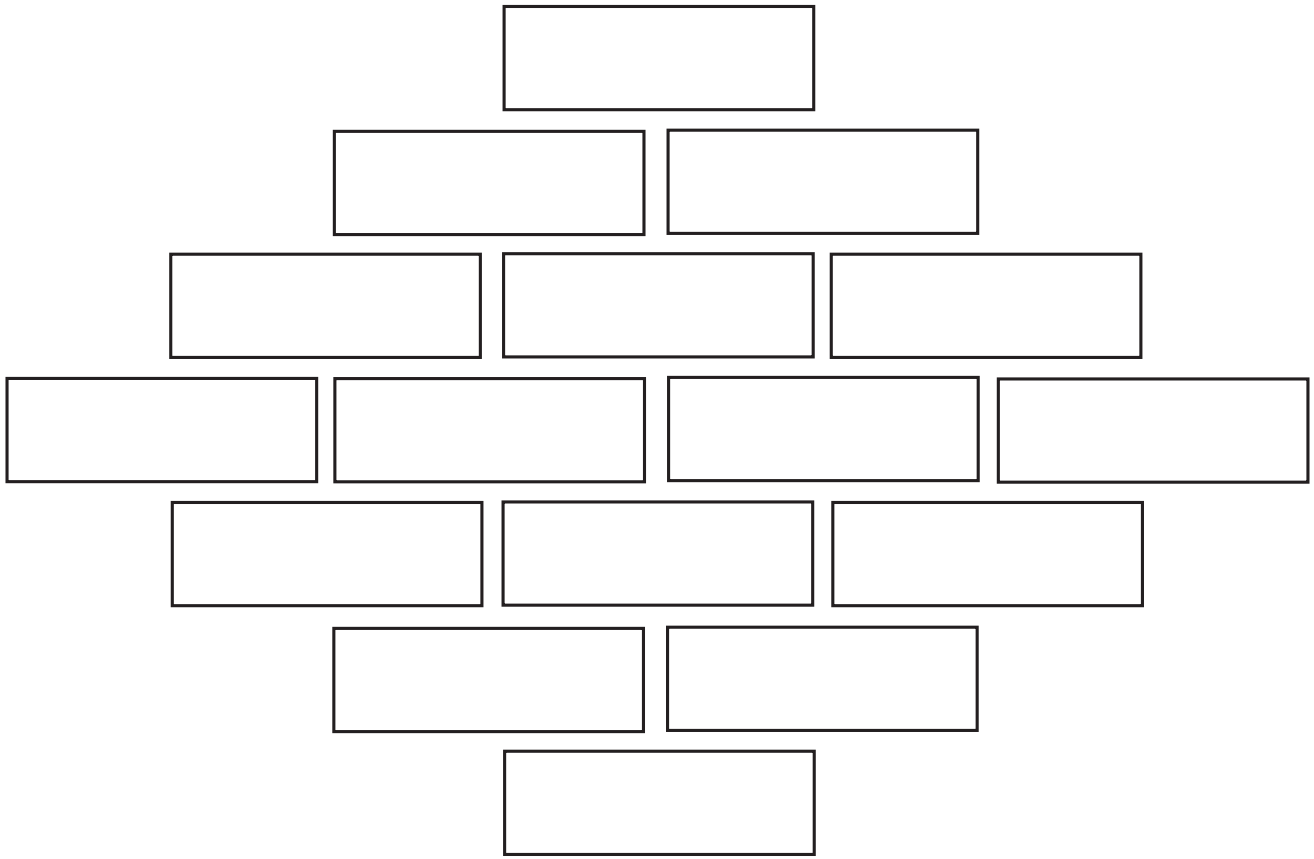
- Minimale Anzahl von Addierern.
- Möglichst kurze Dauer der Summation.

Detailaufbau des $(2n \times 2n)$-Multiplizierers

Benötigt werden vier $(n \times n)$-Multiplizierer, ein $2n$-Bit CSA und ein $3n$-Bit CPA:



Additionsschema für einen $(4n \times 4n)$-Multiplizierer



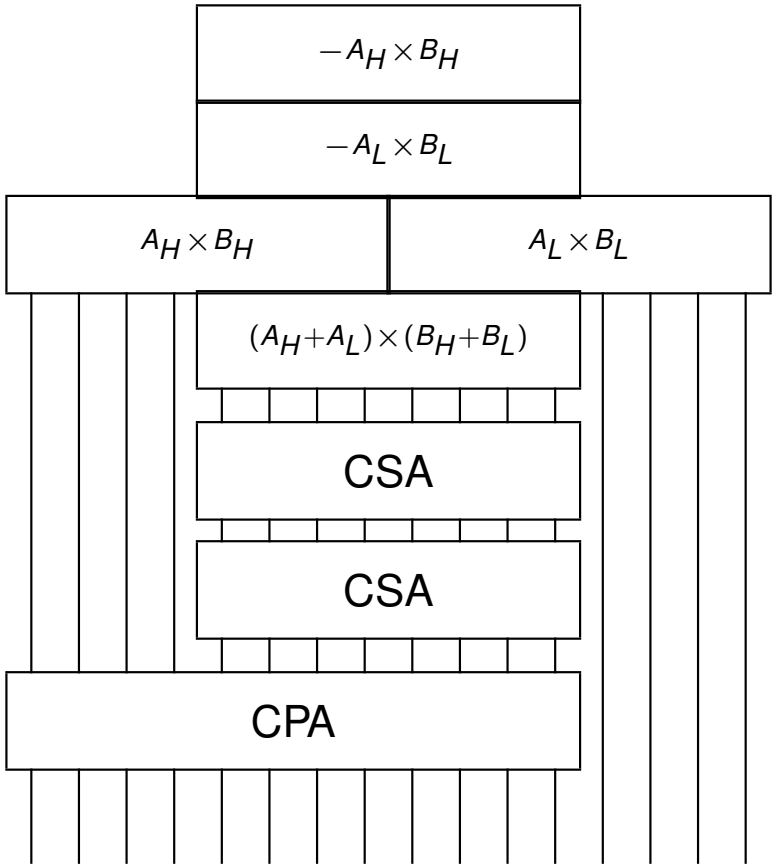
Einsparung von Teilmultiplizierern

Unter Benutzung einer Idee von Karatsuba (1962; beschrieben z. B. in: Aho, Hopcroft, Ullman, The design and analysis of computer algorithms, 1974) kann ein $(2n \times 2n)$-Multiplizierer aus nur drei $(n \times n)$-Multiplizierern aufgebaut werden:

$$\begin{aligned} A \times B &= (A_H \times 2^n + A_L) \times (B_H \times 2^n + B_L) \\ &= A_H \times B_H \times 2^{2n} + (A_H \times B_L + A_L \times B_H) \times 2^n + A_L \times B_L \\ &= A_H \times B_H \times 2^{2n} \\ &\quad + ((A_H + A_L) \times (B_H + B_L) - A_H \times B_H - A_L \times B_L) \times 2^n \\ &\quad + A_L \times B_L \end{aligned}$$

Sparsamer $(2n \times 2n)$-Multiplizierer

Benötigt werden zwei $(n \times n)$-Multiplizierer, ein $((n+1) \times (n+1))$-Multiplizierer, zwei $(2n+2)$-Bit CSA, ein $3n$-Bit CPA und zwei n-Bit CPA:



Quadrierer

Das Quadrieren von A ist im Prinzip durch eine Multiplikation $A \times A$ möglich.
Ein spezieller Quadrierer ist aber schneller (und einfacher) als ein allgemeiner Multiplizierer.

Auf das allgemeine Spaltenschema der Teilprodukte einer Multiplikation können beim Quadrieren folgende Regeln zur Vereinfachung systematisch angewandt werden:

- $A_i A_i = A_i$
- Ein Paar von Termen $A_i A_j$ und $A_j A_i$ in einer Spalte kann ersetzt werden durch den Term $A_i A_j$ (oder $A_j A_i$) in der nächsthöheren Spalte.

Es ist in der Zahlentheorie wohlbekannt, dass $(A^2 \bmod 4) = (A \bmod 2)$. Systematische Reduktion des Spaltenschemas reproduziert unter anderem diesen nützlichen Zusammenhang!

Abhängig von der Länge des Operanden und den spezifischen Optimierungszielen können, obgleich weniger systematisch, weitere Reduktionsschritte erfolgen.

Beispiel: Quadrieren von $A = A_4 A_3 A_2 A_1 A_0 \in \mathrm{UInt}_2(5)$.

Quadrierer (1. Reduktionsschritt, systematisch)

					A_4	A_3	A_2	A_1	A_0
			$\times$		A_4	A_3	A_2	A_1	A_0
					A_4A_0	A_3A_0	A_2A_0	A_1A_0	A_0A_0
				A_4A_1	A_3A_1	A_2A_1	A_1A_1	A_0A_1	
			A_4A_2	A_3A_2	A_2A_2	A_1A_2	A_0A_2		
		A_4A_3	A_3A_3	A_2A_3	A_1A_3	A_0A_3			
	A_4A_4	A_3A_4	A_2A_4	A_1A_4	A_0A_4				
P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

wird reduziert zu

					A_4	A_3	A_2	A_1	A_0
			$\times$		A_4	A_3	A_2	A_1	A_0
	A_4A_3	A_4A_2	A_4A_1	A_4A_0	A_3A_0	A_2A_0	A_1A_0		A_0
	A_4		A_3A_2	A_3A_1	A_2A_1		A_1		
			A_3		A_2				
P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	0	A_0

Quadrierer (2. Reduktionsschritt, sporadisch)

					A_4	A_3	A_2	A_1	A_0
			$\times$		A_4	A_3	A_2	A_1	A_0
	A_4A_3	A_4A_2	A_4A_1	A_4A_0	A_3A_0	A_2A_0	A_1A_0		A_0
	A_4		A_3A_2	A_3A_1	A_2A_1		A_1		
			A_3		A_2				
P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	0	A_0

ergibt mit $A_1A_0 + A_1 = 2A_1A_0 + A_1(1 - A_0) = 2A_1A_0 + A_1\overline{A}_0$

					A_4	A_3	A_2	A_1	A_0
			$\times$		A_4	A_3	A_2	A_1	A_0
	A_4A_3	A_4A_2	A_4A_1	A_4A_0	A_3A_0	A_2A_0	$A_1\overline{A}_0$		A_0
	A_4		A_3A_2	A_3A_1	A_2A_1	A_1A_0			
			A_3		A_2				
P_9	P_8	P_7	P_6	P_5	P_4	P_3	$A_1\overline{A}_0$	0	A_0

Potenzierung

Potenzierung $P = A^k$ mit $k \in \mathbb{N}$ kann als Spezialfall einer Serie von Multiplikationen aufgefasst werden.

Mit $k = \sum_{i=0}^{j-1} k_i \times 2^i$ in kanonischer Binärdarstellung und den Hilfsgrößen $H^{(i)} = A^{2^i}$ für $i = 0, \ldots, j-1$ gilt $A^k = \prod\{H^{(i)} : k_i = 1\}$.

Die $H^{(i)}$ berechnen sich nach den Formeln $H^{(0)} = A, H^{(i+1)} = H^{(i)} \times H^{(i)}$.

Der Iterationsprozess verläuft über $P^{(0)} = 1, P^{(i+1)} = P^{(i)} \times (H^{(i)})^{k_i}, P = P^{(j)}$.

Die $P^{(i)}$ können mit einem Multiplizierer, die $H^{(i)}$ mit einem Quadrierer berechnet werden.

i	k	k_i	$H^{(i)}$	$P^{(i)}$
0	1010111	1	A	1
1	1010111	1	A^2	A
2	1010111	1	A^4	A^3
3	1010111	0	A^8	A^7
4	1010111	1	A^{16}	A^7
5	1010111	0	A^{32}	A^{23}
6	1010111	1	A^{64}	A^{23}
7	1010111		A^{128}	A^{87}

Tabellen-Multiplizierer

Die Schaltfunktion eines Multiplizierers kann auch durch ein ROM realisiert werden.

Dabei wird die Konkatenation der beiden Operanden als Adresse verwendet.

Zur vorzeichenlosen Multiplikation wird ein ROM der Größe $2^{l+m} \times (l + m - 1)$ benötigt (das niederwertigste Bit ergibt sich direkt aus $P_0 = A_0 B_0$).

Für große Operandenlängen scheidet das Verfahren damit als zu aufwendig aus.

Ein rekursiv aufgebauter Multiplizierer kann aber mit kleinen Tabellen-Multiplizierern arbeiten.

Ein ROM zur Quadrierung in $\mathrm{UInt}_2(l)$ hat dagegen nur die Größe $2^l \times (2l - 2)$ und lässt sich damit leichter realisieren (Reduzierung auf $2^l \times (2l - 3)$ Bit möglich).

Realisierung eines Multiplizierers in $\mathrm{UInt}_2(l)$ durch einen Quadrierer

Ausgangspunkt: $4 \times A \times B = (A+B)^2 - (A-B)^2$ nach Vorsortierung $A \geq B$.

Die beiden niederwertigsten Bits der Quadrate sind irrelevant (stimmen überein).

Erforderlich ist ein Quadrierer für Operanden in $\mathrm{UInt}_2(l+1)$.

Bei Auslegung als Tabellenquadrierer sind dazu $2 \times l \times 2^{l+1}$ Bit Speicherplatz nötig.

Bei Parallelberechnung beider Quadrate wird zusätzlich ein Quadrierer in $\mathrm{UInt}_2(l)$ benötigt.

Optimierung:

$$A \times B = \begin{cases} ((A+B)/2)^2 - ((A-B)/2)^2 & \text{für gerades } A+B, \\ ((A-1+B)/2)^2 - ((A-1-B)/2)^2 + B & \text{sonst.} \end{cases}$$

Nun ist nur noch ein Quadrierer in $\mathrm{UInt}_2(l)$ nötig.

Allerdings muss halbiert werden (Rechtsverschiebung),
eine Fallunterscheidung ist nötig (Prüfbit fällt bei der Rechtsverschiebung ab)
und es wird ggf. zusätzlich addiert.

Beispiel (ohne Parallelität): Zur Berechnung von 16-Bit-Produkten aus 8-Bit-Operanden genügt eine Wertetabelle mit 1024 Byte (512 Byte in optimierter Ausführung).

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Division

Division in UInt

Aus dem Dividenden A und dem Divisor $B \neq 0$ werden der Quotient Q und der Rest R berechnet.

Durch die Vorschriften $A = Q \times B + R$ und $0 \leq R < B$ werden R und Q eindeutig festgelegt.

Für $B, Q \in \mathrm{UInt}_2(l)$ hat $A < 2^l \times B$ zu gelten.

Im Falle $B = 0$ liegt eine Division-durch-Null-Ausnahme vor.

Im Falle $A \geq 2^l \times B > 0$ liegt eine Überlauf-Ausnahme vor.

Gilt ebenfalls $A \in \mathrm{UInt}_2(l)$, so kommt dieser Fall generell nicht vor.

Da jedoch bei Multiplikation in $\mathrm{UInt}_2(l)$ ein doppelt langes Produkt anfallen kann,
wird in der Praxis auch $A \in \mathrm{UInt}_2(2l)$ zugelassen.

Basis-Algorithmus für Division in UInt

$$
\begin{aligned}
Q^{(0)} &= 0 \\
R^{(0)} &= A \\
q_i &= \begin{cases} 1 & \text{falls } R^{(i)} \geq 2^{l-1-i} \times B \\ 0 & \text{sonst} \end{cases} \\
Q^{(i+1)} &= 2 \times Q^{(i)} + q_i \\
R^{(i+1)} &= R^{(i)} - 2^{l-1-i} \times q_i \times B, \quad i = 0, 1, \ldots, l-1 \\
Q &= Q^{(l)} \\
R &= R^{(l)}
\end{aligned}
$$

Die Auswahl der Quotientenbits q_i erfolgt hier so, dass unter der Bedingung $A < 2^l \times B$ stets $0 \leq R^{(i)} < 2^{l-i} \times B$ erfüllt ist; dies garantiert dann u. a. auch $0 \leq R < B$.

Der Test $R^{(0)} \geq 2^l \times B$ entdeckt alle Ausnahmen. Dabei genügt die Prüfung $R^{(0)}_{[l:2l-1]} \geq B$.

Eleganterer Algorithmus für Division in UInt

Unter Benutzung einer skalierten Hilfsgröße $B' = 2^l \times B$ ergibt sich:

$$
\begin{aligned}
Q^{(0)} &= 0 \\
R^{(0)} &= A \\
q_i &= \begin{cases} 1 & \text{falls } 2 \times R^{(i)} \geq B' \\ 0 & \text{sonst} \end{cases} \\
Q^{(i+1)} &= 2 \times Q^{(i)} + q_i \\
R^{(i+1)} &= 2 \times R^{(i)} - q_i \times B', \quad i = 0, 1, \ldots, l-1 \\
Q &= Q^{(l)} \\
R &= 2^{-l} \times R^{(l)}
\end{aligned}
$$

Mit $A < B'$ gilt stets $0 \leq R^{(i)} < B'$, also insbesondere $0 \leq R < B$.

Der Vergleich $2 \times R^{(i)} \geq B'$ und die Subtraktion $2 \times R^{(i)} - q_i \times B'$ werden faktisch nur auf den höherwertigen $l+1$ Bits des Registers R durchgeführt: $R^{(i)}_{[l-1:2l-1]} \geq B$ bzw.
$R^{(i)}_{[l-1:2l-1]} + (-q_i \times B)$ (für 2K-Addition kann $-q_i \times B$ vorberechnet werden)

Gemeinsames Register für Quotient und Rest

Q kann effizient in den freigewordenen niederwertigen Bits von R gespeichert werden:

$$
\begin{aligned}
R^{(0)} &= A \\
q_i &= \begin{cases} 1 & \text{falls } 2 \times R^{(i)} \geq B' \\ 0 & \text{sonst} \end{cases} \\
R^{(i+1)} &= (2 \times R^{(i)} - q_i \times B') \mid q_i, \quad i = 0, 1, \ldots, l-1 \\
Q &= R^{(l)}_{[0:l-1]} \\
R &= R^{(l)}_{[l:2l-1]}
\end{aligned}
$$

Restoring-Division

In jedem Schritt wird versuchsweise eine Subtraktion ausgeführt,
die allerdings bei negativem Rest durch Addition wieder rückgängig gemacht wird.

Aufwand: l Links-Verschiebungen, l Subtraktionen, l Tests, durchschnittlich $l/2$ Additionen.

$$
\begin{aligned}
Q^{(0)} &= 0 \\
R^{(0)} &= A \\
R_*^{(i+1)} &= 2 \times R^{(i)} - B' \\
(R^{(i+1)}, q_i) &= \begin{cases} (R_*^{(i+1)}, 1) & \text{falls } R_*^{(i+1)} \geq 0 \\ (R_*^{(i+1)} + B', 0) & \text{sonst} \end{cases} \\
Q^{(i+1)} &= 2 \times Q^{(i)} + q_i \\
Q &= Q^{(l)} \\
R &= 2^{-l} \times R^{(l)}
\end{aligned}
$$

Restoring-Division (Beispiel)

$$A = 436_{10} = 0110110100_2, B = 15_{10} = 01111_2 \Rightarrow Q = 29_{10} = 11101_2, R = 1$$

	0	1	1	0	1	1	0	1	0	0	|	-	-	-	-	-	$R^{(0)}$, $Q^{(0)}$
0	1	1	0	1	1	0	1	0	0	-	|	-	-	-	-	-	linksschieben
	0	1	1	1	1												subtrahieren
0	0	1	1	0	0	0	1	0	0	-	|	-	-	-	-	1	positiver Rest
0	1	1	0	0	0	1	0	0	-	-	|	-	-	-	1	-	linksschieben
	0	1	1	1	1												subtrahieren
0	0	1	0	0	1	1	0	0	-	-	|	-	-	-	1	1	positiver Rest
0	1	0	0	1	1	0	0	-	-	-	|	-	-	1	1	-	linksschieben
	0	1	1	1	1												subtrahieren
0	0	0	1	0	0	0	0	-	-	-	|	-	-	1	1	1	positiver Rest
0	0	1	0	0	0	0	-	-	-	-	|	-	1	1	1	-	linksschieben
	0	1	1	1	1												subtrahieren
1	1	1	0	0	1	0	-	-	-	-	|	-	1	1	1	0	negativer Rest
	0	1	1	1	1												addieren
0	0	1	0	0	0	0	-	-	-	-	|	-	1	1	1	0	
0	1	0	0	0	0	-	-	-	-	-	|	1	1	1	0	-	linksschieben
	0	1	1	1	1												subtrahieren
	0	0	0	0	1	-	-	-	-	-	|	1	1	1	0	1	positiver Rest

Non-performing-Division

Der Partialrest wird nur dann durch die Differenz ersetzt, wenn diese nicht negativ war.

Aufwand: l Links-Verschiebungen, l Tests, durchschnittlich $l/2$ Subtraktionen.

$$
\begin{aligned}
Q^{(0)} &= 0 \\
R^{(0)} &= A \\
(R^{(i+1)}, q_i) &= \begin{cases} (2 \times R^{(i)} - B', 1) & \text{falls } 2 \times R^{(i)} \geq B' \\ (2 \times R^{(i)}, 0) & \text{sonst} \end{cases} \\
Q^{(i+1)} &= 2 \times Q^{(i)} + q_i \\
Q &= Q^{(l)} \\
R &= 2^{-l} \times R^{(l)}
\end{aligned}
$$

Non-performing-Division (Beispiel)

$$A = 436_{10} = 0110110100_2, B = 15_{10} = 01111_2 \Rightarrow Q = 29_{10} = 11101_2, R = 1$$

	0	1	1	0	1	1	0	1	0	0	|	-	-	-	-	-	$R^{(0)}, Q^{(0)}$
0	1	1	0	1	1	0	1	0	0	-	|	-	-	-	-	-	linksschieben
	0	1	1	1	1												subtrahieren
0	0	1	1	0	0	0	1	0	0	-	|	-	-	-	-	1	positiver Rest
0	1	1	0	0	0	1	0	0	-	-	|	-	-	-	1	-	linksschieben
	0	1	1	1	1												subtrahieren
0	0	1	0	0	1	1	0	0	-	-	|	-	-	-	1	1	positiver Rest
0	1	0	0	1	1	0	0	-	-	-	|	-	-	1	1	-	linksschieben
	0	1	1	1	1												subtrahieren
0	0	0	1	0	0	0	0	-	-	-	|	-	-	1	1	1	positiver Rest
0	0	1	0	0	0	0	-	-	-	-	|	-	1	1	1	-	linksschieben
	0	1	1	1	1												subtrahieren
1	1	1	0	0	1	0	-	-	-	-	|	-	1	1	1	0	negativer Rest
0	0	1	0	0	0	0	-	-	-	-	|	-	1	1	1	0	alten Rest kopieren
0	1	0	0	0	0	-	-	-	-	-	|	1	1	1	0	-	linksschieben
	0	1	1	1	1												subtrahieren
	0	0	0	0	1	-	-	-	-	-	|	1	1	1	0	1	positiver Rest

Non-restoring-Division

Bei der Non-restoring-Division wird nach einer Subtraktion mit einem eventuell entstandenen negativen Rest weitergearbeitet.

Statt Subtraktionen werden in diesem Fall anschließend Additionen vorgenommen,
bis der Partialrest dadurch wieder nicht negativ geworden ist.

Endet das Verfahren mit einem negativen Partialrest, muss daraus durch einen Korrekturschritt der entsprechende nicht negative Rest berechnet werden.

$R^{(i)}$ nimmt nur Werte im Bereich $[-B', B' - 1]$ an;
damit genügt eine 2K-, 1K- oder VB-Darstellung mit $2l + 1$ Bits.

Non-restoring-Division (Algorithmus)

$$
\begin{aligned}
Q^{(0)} &= 0 \\
R^{(0)} &= A \\
R^{(i+1)} &= \begin{cases} 2 \times R^{(i)} - B' & \text{falls } R^{(i)} \geq 0 \\ 2 \times R^{(i)} + B' & \text{sonst} \end{cases} \\
q_i &= \begin{cases} 1 & \text{falls } R^{(i+1)} \geq 0 \\ 0 & \text{sonst} \end{cases} \\
Q^{(i+1)} &= 2 \times Q^{(i)} + q_i \\
R^{(l+1)} &= \begin{cases} R^{(l)} & \text{falls } R^{(l)} \geq 0 \\ R^{(l)} + B' & \text{sonst} \end{cases} \\
Q &= Q^{(l)} \\
R &= 2^{-l} \times R^{(l+1)}
\end{aligned}
$$

Non-restoring- vs. Restoring-Division

Durch das alternierende Schema der Non-restoring-Division wird über mehrere Schritte hinweg derselbe Effekt wie mit der Restoring-Division erreicht:

$R_*^{(i)} = 2 \times R^{(i-1)} - B' < 0 \Rightarrow R_*^{(i+1)} = 2 \times R^{(i)} - B' = 4 \times R^{(i-1)} - B'$
bei der Restoring-Division,

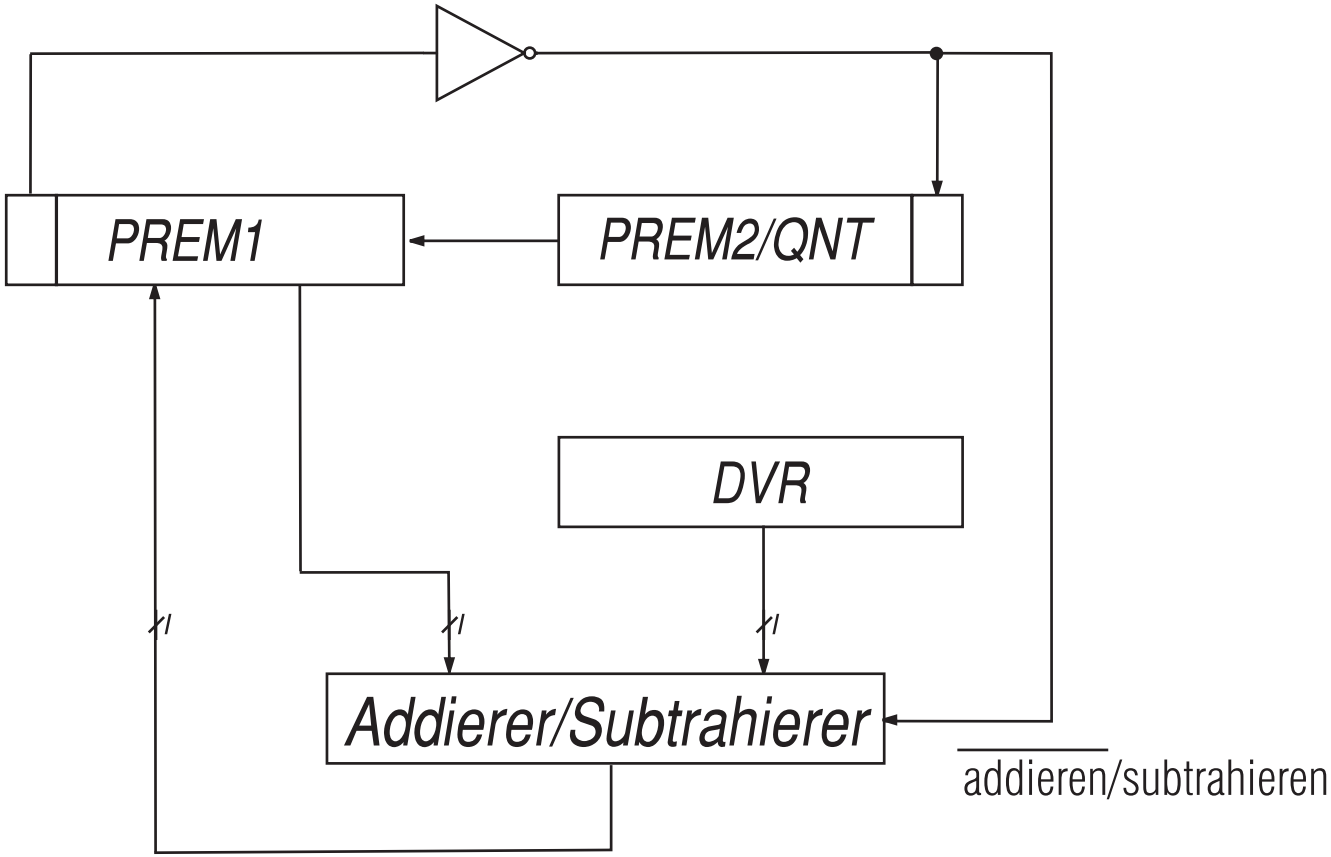
$R^{(i)} = 2 \times R^{(i-1)} - B' < 0 \Rightarrow R^{(i+1)} = 2 \times R^{(i)} + B' = 4 \times R^{(i-1)} - B'$
bei der Non-restoring-Division.

Non-restoring-Division (Beispiel)

$$A = 436_{10} = 0110110100_2, B = 15_{10} = 01111_2 \Rightarrow Q = 29_{10} = 11101_2, R = 1$$

0	0	1	1	0	1	1	0	1	0	0	|	-	-	-	-	-	$R^{(0)}$, $Q^{(0)}$
0	1	1	0	1	1	0	1	0	0	-	|	-	-	-	-	-	linksschieben
0	0	1	1	1	1												subtrahieren
0	0	1	1	0	0	0	1	0	0	-	|	-	-	-	-	1	positiver Rest
0	1	1	0	0	0	1	0	0	-	-	|	-	-	-	1	-	linksschieben
0	0	1	1	1	1												subtrahieren
0	0	1	0	0	1	1	0	0	-	-	|	-	-	-	1	1	positiver Rest
0	1	0	0	1	1	0	0	-	-	-	|	-	-	1	1	-	linksschieben
0	0	1	1	1	1												subtrahieren
0	0	0	1	0	0	0	0	-	-	-	|	-	-	1	1	1	positiver Rest
0	0	1	0	0	0	0	-	-	-	-	|	-	1	1	1	-	linksschieben
0	0	1	1	1	1												subtrahieren
1	1	1	0	0	1	0	-	-	-	-	|	-	1	1	1	0	negativer Rest
1	1	0	0	1	0	-	-	-	-	-	|	1	1	1	0	-	linksschieben
0	0	1	1	1	1												addieren
0	0	0	0	0	1	-	-	-	-	-	|	1	1	1	0	1	positiver Rest

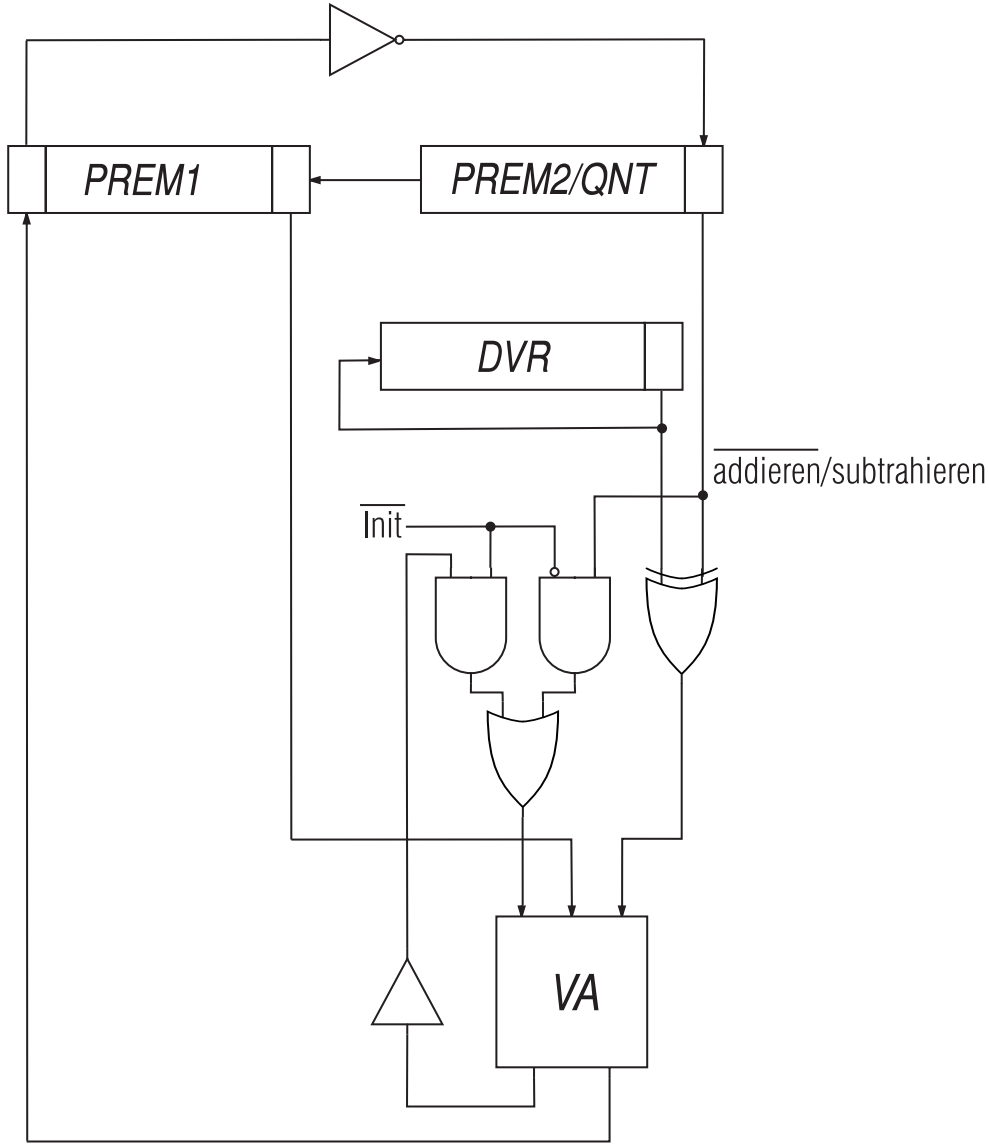
Sequentieller Non-restoring-Dividierer



Vergleich sequentieller Non-restoring-Dividierer

l	Addierer	t	a	$a \times t \times 10^{-4}$
16	RCLA	192	688	11,87
	BCLA	320	428	13,70
	1-Level-Carry-Skip	336	368	12,36
32	RCLA	512	956	49,95
	SRCLA	512	1160	59,39
	2-Level-Carry-Skip	832	828	68,89
	BCLA	896	856	76,69
	1-Level-Carry-Skip	928	736	68,30
64	SRCLA	1024	2544	260,51
	MSBCLA	1280	2080	266,24
	RCLA	1536	1912	293,37
	SBCLA	1792	2060	369,15
	1-Level-Carry-Skip	2240	1472	329,73
	BCLA	2304	1832	422,09

Serieller Non-restoring-Dividierer



Non-restoring-Division mit Signed-Binary-Quotient

Statt des bei der Non-restoring-Division erhaltenen Quotienten
könnte bei einer Addition $\bar{1}$ als Quotientenziffer,
bei einer Subtraktion 1 als Quotientenziffer erzeugt werden:

$$\begin{aligned} q_i &= \begin{cases} 1 & \text{falls } R^{(i)} > 0 \\ \bar{1} & \text{falls } R^{(i)} < 0 \end{cases} \\ R^{(i+1)} &= 2 \times R^{(i)} - q_i \times B' \end{aligned}$$

Im Falle $R^{(i)} = 0$ stoppt das Verfahren. Modifizierter Korrekturschritt:

$$\text{Sei } k = \begin{cases} l & \text{falls } R^{(i)} \neq 0 \quad \forall i < l \\ \min\{i | R^{(i)} = 0\} & \text{sonst} \end{cases}$$

$$\text{Setze } R = \begin{cases} R^{(k)} & \text{falls } R^{(k)} \geq 0 \\ R^{(k)} + B' & \text{sonst} \end{cases}$$

Recoding des Quotienten der Non-restoring-Division

Setze

$$Q = (\tilde{q}_0 \ldots \tilde{q}_{l-1})$$

mit

$$\tilde{q}_i = \begin{cases} 1 & \text{falls } i \leq k-2 \ \wedge \ q_{i+1} = 1 \\ 0 & \text{falls } i \leq k-2 \ \wedge \ q_{i+1} = \bar{1} \\ 1 & \text{falls } i = k-1 \ \wedge \ R^{(k)} \geq 0 \\ 0 & \text{falls } i = k-1 \ \wedge \ R^{(k)} < 0 \\ 0 & \text{falls } i \geq k \end{cases}$$

Die Berechnung der $\tilde{q}_i$ kann „on-the-fly“ erfolgen.

Statt $q_i = \bar{1}$ kann auch direkt $q_i = 0$ codiert werden.

Non-restoring-Division mit vorzeichenbehafteten Operanden

Die Non-restoring-Division kann auf vorzeichenbehaftete Operanden erweitert werden, z. B.:

$$q_i = \begin{cases} 1 & \text{falls} \quad R^{(i)} \times B' \geq 0 \\ \bar{1} & \text{falls} \quad R^{(i)} \times B' < 0 \end{cases}$$

und

$$R^{(i+1)} = 2 \times R^{(i)} - q_i \times B'$$

Die Quotientenziffer für $R^{(i)} = 0$ ist beliebig, es gälte z. B. auch die Regel

$$q_i = \begin{cases} 1 & \text{falls Vorzeichen}(R^{(i)}) = \text{Vorzeichen}(B) \\ \bar{1} & \text{sonst} \end{cases}$$

Non-restoring-Division in Int_2 (Beispiel)

Sei $l = 3$, $A = 010000_2 = 16$, $B = 101_2 = -3$.

$R^{(0)} = A$		0	0	1	0	0	0	0	
$2 \times R^{(0)}$		0	1	0	0				$q_0 = \bar{1}$
$+B'$	+	1	1	0	1				
$R^{(1)}$		0	0	0	1				
$2 \times R^{(1)}$		0	0	1	0				$q_1 = \bar{1}$
$+B'$	+	1	1	0	1				
$R^{(2)}$		1	1	1	1				
$2 \times R^{(2)}$		1	1	1	0				$q_2 = 1$
$-B'$	+	0	0	1	1				
$R^{(3)}$		0	0	0	1				

Ergebnis: $Q = \bar{1}\bar{1}1_2 = \bar{1}0\bar{1}_2 = -5$ und $R = 2^{-3} \times R^{(3)} = 1$.

Schlusskorrektur für Non-restoring-Division mit Vorzeichen

Für vorzeichenbehaftete Operanden wird in der Regel verlangt,
dass der Rest R und der Dividend A dasselbe Vorzeichen besitzen.

Ist dies nach Abschluss der Non-restoring-Division nicht ohnehin der Fall, müssen in einem abschließenden Schritt der Quotient $Q^{(l)}$ und der Rest $R^{(l)}$ geeignet korrigiert werden:

$$A \times R^{(l)} < 0, B' \times R^{(l)} > 0 \Rightarrow R^{(l+1)} = R^{(l)} - B',\ Q = Q^{(l)} + 1$$

$$A \times R^{(l)} < 0, B' \times R^{(l)} < 0 \Rightarrow R^{(l+1)} = R^{(l)} + B',\ Q = Q^{(l)} - 1$$

Schlusskorrektur für Non-restoring-Division in Int_2 (Beispiel)

Sei $l = 3$, $A = 010100_2 = 20$, $B = 011_2 = 3$.

$R^{(0)} = A$		0	0	1	0	1	0	0	
$2 \times R^{(0)}$		0	1	0	1				$q_0 = 1$
$-B'$	+	1	1	0	1				
$R^{(1)}$		0	0	1	0				
$2 \times R^{(1)}$		0	1	0	0				$q_1 = 1$
$-B'$	+	1	1	0	1				
$R^{(2)}$		0	0	0	1				
$2 \times R^{(2)}$		0	0	1	0				$q_2 = 1$
$-B'$	+	1	1	0	1				
$R^{(3)}$		1	1	1	1				

$R^{(4)} = R^{(3)} + B' = 0010000$, d. h. $R = 2^{-3} \times R^{(4)} = 2$, und $Q = Q^{(3)} - 1 = 110_2 = 6$.

Schlusskorrektur für Non-restoring-Division mit Vorzeichen (2)

Ein abschließender Korrekturschritt wird auch nötig,
wenn vor Ende der Non-restoring-Division der Rest Null auftritt.

Statt des korrekten Ergebnisses $R = 0$ wird nämlich wegen der vorzunehmenden Subtraktion sowie eventuell nachfolgender Additionen der Rest $R = -B'$ und ein falscher Quotient berechnet.

Die in diesem Fall durchzuführende Korrektur lautet:

$$R^{(l)} = -B' \Rightarrow R^{(l+1)} = 0,\ Q = Q^{(l)} - 1$$

Schlusskorrektur für Non-restoring-Division in Int_2 (2. Beispiel)

Sei $l = 3$, $A = 110100_2 = -12$, $B = 011_2 = 3$.

$R^{(0)} = A$		1	1	1	0	1	0	0	
$2 \times R^{(0)}$		1	1	0	1				$q_0 = \bar{1}$
$+B'$	+	0	0	1	1				
$R^{(1)}$		0	0	0	0				Rest Null
$2 \times R^{(1)}$		0	0	0	0				$q_1 = 1$
$-B'$	+	1	1	0	1				
$R^{(2)}$		1	1	0	1				
$2 \times R^{(2)}$		1	0	1	0				$q_2 = \bar{1}$
$+B'$	+	0	0	1	1				
$R^{(3)}$		1	1	0	1				

$R^{(4)} = R^{(3)} + B' = 0$, d. h. $R = 0$, und $Q = Q^{(3)} - 1 = \bar{1}00_2 = -4$.

Recoding für Non-restoring-Division in Int_2

Der während einer Non-restoring-Division berechnete Quotient
muss abschließend in 2-Komplement-Darstellung überführt werden.

Diese erhält man als $\tilde{Q} = (1 - p_0)p_1p_2 \ldots p_{l-1}1$ mit $p_i = (q_i + 1)/2$.

Die bereits früher benutzte Kodierung des Wertes $\bar{1}$ durch die Ziffer 0
entspricht exakt dem Übergang von q_i zu p_i.

Die Umwandlung des Quotienten kann bitweise und simultan zur Berechnung der Quotientenbits geschehen; die abschließenden Korrekturschritte können dann bereits auf einer 2-Komplement-Darstellung des Quotienten aufsetzen.

Beispiel:
Mit $A = 110100_2$ und $B = 011_2$ erhält man statt des Quotienten $Q^{(3)} = \bar{1}1\bar{1}_2$
direkt die Darstellung $\tilde{Q} = (1 - 0)101_2 = 1101_2$.

Durch den abschließenden Korrekturschritt erhält man $Q = \tilde{Q} - 1 = 1100_2 = -4$.

Recoding für Non-restoring-Division in Int_2 (Beweis)

$$
\begin{aligned}
\tilde{Q} &= -(1-p_0)\times 2^l + \sum_{i=1}^{l-1} p_i \times 2^{l-i} + 1 \\
&= q_0 \times 2^{l-1} - 2^{l-1} + \sum_{i=1}^{l-1} (q_i + 1) \times 2^{l-i-1} + 1 \\
&= q_0 \times 2^{l-1} - 2^{l-1} + \sum_{i=1}^{l-1} q_i \times 2^{l-i-1} + \sum_{i=1}^{l-1} 2^{l-i-1} + 1 \\
&= \sum_{i=0}^{l-1} q_i \times 2^{l-i-1} \\
&= Q^{(l)}
\end{aligned}
$$

Array-Dividierer

Alle Divisionsalgorithmen können durch einen Array-Dividierer implementiert werden,
in dem jeder Schritt des Algorithmus in einer separaten Zeile des Arrays ausgeführt wird.

Für eine Radix-2-Division benötigt man l Zeilen zu je l Zellen.

Bei Verwendung eines Ripple-Carry-Addierers ist die Ausführungszeit proportional zu l^2,
bei Verwendung eines Carry-Lookahead-Addierers proportional zu $l \times \log l$.

Non-performing-Array-Dividierer:

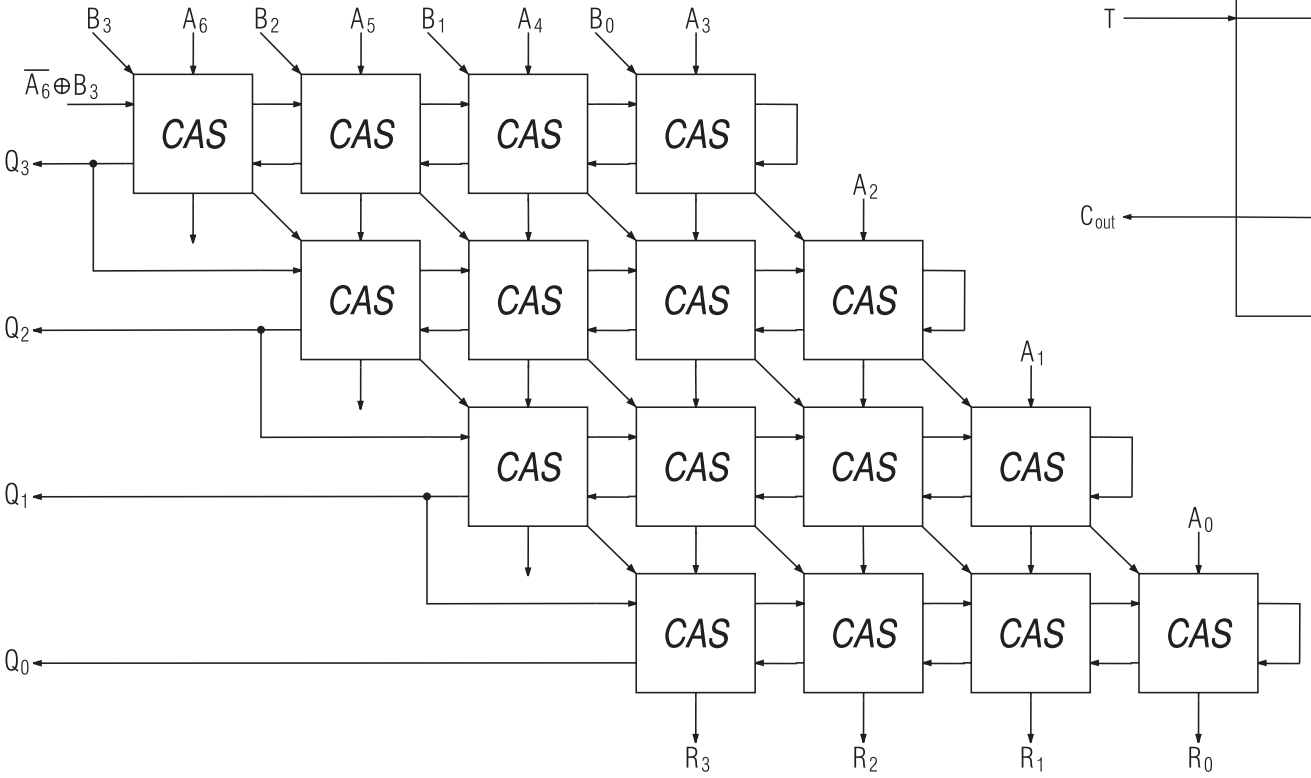
In einer Zeile wird die Differenz zwischen Partialrest und Divisor berechnet
und das Quotientenbit entsprechend des Vorzeichens dieser Differenz bestimmt.

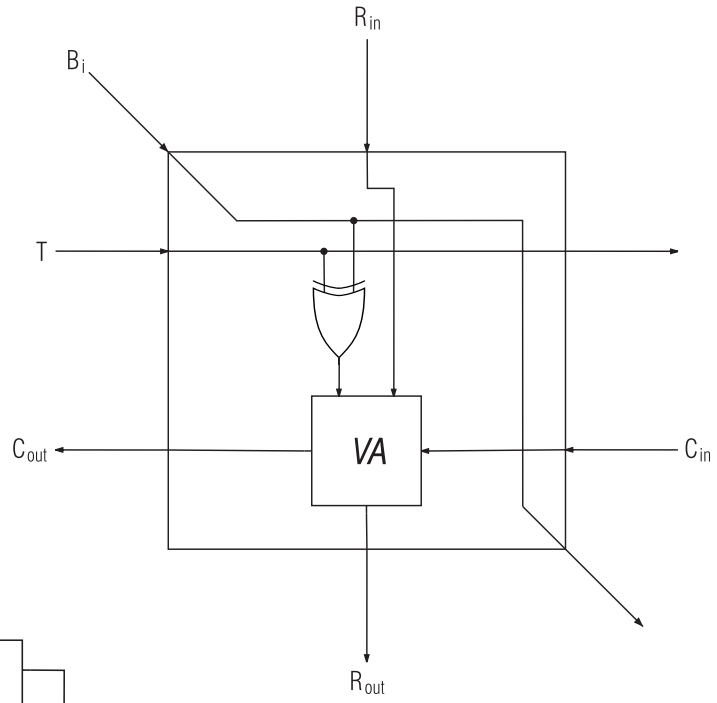
Bei erfolgreicher Subtraktion wird deren Ergebnis an die nächste Zeile von Zellen weitergereicht,
andernfalls der ursprüngliche Partialrest.

Die Auswahl des weiterzureichenden Operanden erfolgt durch Multiplexer,
gesteuert mittels des in derselben Zeile berechneten Quotientenbits.

Non-restoring-Division kann entsprechend durch einen Array-Dividierer implementiert werden
(bringt allerdings fast keinen Zeitgewinn).

Non-restoring-Array-Dividierer





Beschleunigung der Division

- Radix-2^k-Division, mit $k > 1$
- Abbruch der Division, wenn der Partialrest Null geworden ist
- Normalisierung der Operanden
- Überspringen von Folgen von Nullen oder Einsen im Partialrest
- Carry-save-Techniken bei Berechnung des Partialrests
- Unscharfe Vergleiche zur Bestimmung eines Quotientenbits
- Iterative Konvergenzalgorithmen

Radix-2^k-Division

Radix-2^k-Division bestimmt mehrere Quotientenbits gleichzeitig in einem Divisionsschritt.

Es sind die Vielfachen $2 \times B', 3 \times B', \ldots, (2^k - 1) \times B'$ zu bilden.

Bei Non-performing-Division bestimmt der Faktor des größten Vielfachen von B', das kleiner oder gleich dem augenblicklichen Partialrest ist, welche Folge von k Quotientenbits ausgewählt wird.

Entsprechend gibt es auch Varianten der Non-restoring-Division.

Prinzipielle Vorgehensweisen:

- Es werden $2^k - 1$ Vergleichsoperationen (parallel) durchgeführt.
  Dies ist für große k meist zu aufwendig.
- Es werden einige wenige führende Bits von Divisor und Partialrest inspiziert,
  um eine kleine Auswahl in Frage kommender Vielfacher zu bestimmen.

  Für diese Auswahl erfolgt dann ein vollständiger Vergleich.

Normalisierung der Operanden

Bei der Normalisierung werden durch Linksverschieben führende Ziffern aus den Operanden entfernt, die mit dem Vorzeichen übereinstimmen.

Wesentliche Ziele der Normalisierung:

- Divisor und initialer Partialrest sollen etwa auf die gleiche Größenordnung gebracht werden
- Überspringen arithmetischer Operationen

Viele Divisionsverfahren setzen normalisierte Operanden voraus.

Überspringen von Folgen von Nullen im Partialrest

Ist der Divisor vollständig normalisiert und besitzt der Partialrest zwei führende Nullen, braucht bei Non-performing-Division kein Operandenvergleich zu erfolgen, da das nächste Quotientenbit sicher Null ist.

Es wird dann nur die Linksverschiebung von Partialrest und Quotient durchgeführt, wobei als Quotientenbit Null eingesetzt wird.

Besitzt der Partialrest sogar mehr als zwei führende Nullen, können alle bis auf eine gleichzeitig übersprungen werden (Renormalisierung des Partialrests).

Bei der Abschlussbehandlung einer derart modifizierten Division ist zu beachten, dass die Anzahl überlesbarer Nullen im Partialrest größer sein kann als die Anzahl noch zu berechnender Quotientenbits.

Als Verschiebedistanz ist deshalb das Minimum aus der Anzahl überlesbarer Nullen im Partialrest und der Anzahl noch zu berechnender Quotientenbits zu wählen.

Achtung! Besitzt der Partialrest nur eine führende Null, kann daraus nicht auf den Wert des nächsten Quotientenbits geschlossen werden.

Überspringen von Nullen und Einsen

Bei der Non-restoring-Division mit vollständig normalisiertem Divisor können in der Phase der Subtraktion führende Nullen, in der Phase der Addition führende Einsen überlesen werden.

Es sind die drei führenden Bits des Partialrests in 2-Komplement-Darstellung zu prüfen:

- Sind diese Null, würde auf eine Subtraktion sofort eine Addition folgen.
  Die Subtraktion wird aufgeschoben, es ergibt sich ein Quotientenbit Null.
- Sind diese dagegen Eins, würde auf eine Addition sofort eine Subtraktion folgen.
  Die Addition wird aufgeschoben, es ergibt sich ein Quotientenbit Eins.

Dieses Verfahren kann auch auf mehr als drei führende Bits angewandt werden.

Effekt der so optimierten Non-restoring-Division:

Pro arithmetischer Operation werden durchschnittlich 2,66 Bits des Quotienten erzeugt.

Wird die Anzahl der simultan überlesenen Positionen beschränkt, sinkt dieser Wert
(1,5 Bits pro arithmetischer Operation, falls pro Schritt maximal 2 Bits überlesen werden).

Bei Restoring-Division lassen sich nur durchschnittlich 1,74 Bits pro Subtraktion erzielen.

Überspringen von Nullen und Einsen bei Non-restoring-Division in UInt_2

Bei positivem Partialrest:

- Normalisierung des Partialrests (Überlesen von Nullen).
- Für jeden Normalisierungsschritt wird 0 als Quotientenbit eingetragen.
- Linksverschiebung des Partialrests, Subtraktion des Divisors.
- Ist der Partialrest negativ, füge Quotientenbit 0 ein, sonst 1.

Bei negativem Partialrest:

- Normalisierung des Partialrests (Überlesen von Einsen).
- Für jeden Normalisierungsschritt wird 1 als Quotientenbit eingetragen.
- Linksverschiebung des Partialrests, Addition des Divisors.
- Ist der Partialrest negativ, füge Quotientenbit 0 ein, sonst 1.

SRT-Division

- Benannt nach Sweeney, Robertson und Tocher
- Klasse von Verfahren
- „Moderner“ Divisionsalgorithmus
- Variante der Non-restoring-Division
- Theorie systematisiert die Technik des Überspringens von Nullen und Einsen
- Benutzt redundante Zahlendarstellung für Quotienten

Allgemeines Schema der Radix-2-SRT-Division in UInt_2

$$R^{(i+1)} \quad = \quad 2 \times R^{(i)} - q_i \times B',$$

$$\text{erlaubte Werte für } q_i: \quad \begin{cases} 1 & \text{falls } 0 \leq 2 \times R^{(i)} \\ 0 & \text{falls } -B' \leq 2 \times R^{(i)} < B' \\ \bar{1} & \text{falls } 2 \times R^{(i)} < 0 \end{cases}$$

Es gilt stets $-B' \leq R^{(i)} < B'$, in 2K-, 1K- oder VB-Darstellung genügen damit $2l + 1$ Bits.

Der Spielraum in der Auswahl der Quotientenziffern kann genutzt werden,
um die Geschwindigkeit durch eine der folgenden Techniken zu erhöhen:

- Die Wahrscheinlichkeit für eine Quotientenziffer 0 sollte groß werden, um durchschnittlich weniger arithmetische Operationen pro Quotientenziffer durchführen zu müssen.
- Vergleichsoperationen (einschließlich der Berechnung der Vergleichsoperanden) sollten möglichst einfach durchführbar sein.

SRT-Division mit Präferenz für Quotientenziffer 0

$$
\begin{aligned}
R^{(i+1)} &= 2 \times R^{(i)} - q_i \times B', \\
q_i &= \begin{cases} 1 & \text{falls } B' \leq 2 \times R^{(i)} \\ 0 & \text{falls } -B' \leq 2 \times R^{(i)} < B' \\ \bar{1} & \text{falls } 2 \times R^{(i)} < -B' \end{cases}
\end{aligned}
$$

Problem: Durchführung der Vergleichsoperationen im Allgemeinen zu aufwendig!

In synchronen Implementierungen ist eine variierende Anzahl von Operationen überdies irrelevant.

SRT-Division mit vereinfachten Vergleichsoperationen

Für Dividenden $A \in \mathrm{Int}_2(2l)$ und normalisierten Divisor $B \in \mathrm{UInt}_2(l)$ garantiert die Vorschrift

$$R^{(i+1)} = 2 \times R^{(i)} - q_i \times B',$$

$$q_i = \begin{cases} 1 & \text{falls } 2^{2l-2} \leq R^{(i)} \\ 0 & \text{falls } -2^{2l-2} \leq R^{(i)} < 2^{2l-2} \\ \bar{1} & \text{falls } R^{(i)} < -2^{2l-2} \end{cases}$$

die Invariante $R^{(i)} \in \mathrm{Int}_2(2l)$.

Zur Entscheidung genügen in 2-Komplement-Darstellung die beiden Bits $R^{(i)}_{2l-1}$ und $R^{(i)}_{2l-2}$:

$2^{2l-2} \leq R^{(i)} \Leftrightarrow R^{(i)}_{2l-1} = 0 \wedge R^{(i)}_{2l-2} = 1$ bzw. $R^{(i)} < -2^{2l-2} \Leftrightarrow R^{(i)}_{2l-1} = 1 \wedge R^{(i)}_{2l-2} = 0$.

Für einen normalisierten Dividenden $A \in \mathrm{UInt}_2(2l)$ muss ein Denormalisierungsschritt vorgeschaltet oder ein Register $R^{(i)} \in \mathrm{Int}_2(2l+1)$ benutzt werden.

SRT-Division mit normalisiertem Divisor in $\mathrm{Int}_2(l)$

Ausgehend von $A \in \mathrm{Int}_2(2l-1)$ ergibt sich mit der Vorschrift

$$R^{(i+1)} = 2 \times R^{(i)} - q_i \times B',$$

$$q_i = \begin{cases} 1 & \text{falls } 2^{2l-3} \leq R^{(i)}, B > 0 \\ \bar{1} & \text{falls } 2^{2l-3} \leq R^{(i)}, B < 0 \\ 0 & \text{falls } -2^{2l-3} \leq R^{(i)} < 2^{2l-3} \\ \bar{1} & \text{falls } R^{(i)} < -2^{2l-3}, B > 0 \\ 1 & \text{falls } R^{(i)} < -2^{2l-3}, B < 0 \end{cases}$$

die Invariante $R^{(i)} \in \mathrm{Int}_2(2l-1)$.

Sollen die Vorzeichen von R und A übereinstimmen, ist die Voraussetzung $A \in \mathrm{Int}_2(2l-1)$ lediglich für $2^{2l-2} \leq A \leq 2^{2l-2} + 2^{l-1} - 1$, $B = -2^{l-1}$ nicht erfüllt.

In diesen Fällen gilt $Q = B$ und $R = A - 2^{2l-2} = A_{[0:l-1]}$.

Entsprechendes gilt für die Variante zur Minimierung von $|R|$.

SRT-Division mit variabler Verschiebedistanz

Die durchschnittliche Anzahl der Normalisierungsverschiebungen in der SRT-Division mit Vergleichsgröße $K = 2^{2l-2}$ hängt vom Wert des normalisierten Divisors B' ab.

3 Quotientenbits pro arithmetischer Operation wird erreicht für $17/28 \times 2^{2l} \leq B' \leq 3/4 \times 2^{2l}$.

Für $B'/2^{2l}$ außerhalb $[17/28, 3/4]$ kann eine andere Vergleichsgröße K verwendet werden.

Zur Auswahl der Vergleichsgröße K und zur Bestimmung der Quotientenziffern genügen einige führende Bits von Divisor und Partialrest. Beispiel für 5 führende Bits:

$K/2^{2l-1}$	$B'' = B'/2^{2l} = B/2^l$	
	theoretisch bester Bereich	praktischer Bereich
$3/8$	$9/20 \leq B'' \leq 9/16$	$1/2 \leq B'' < 9/16$
$7/16$	$21/40 \leq B'' \leq 21/32$	$9/16 \leq B'' < 5/8$
$1/2$	$3/5 \leq B'' \leq 3/4$	$5/8 \leq B'' < 3/4$
$5/8$	$3/4 \leq B'' \leq 15/16$	$3/4 \leq B'' < 15/16$
$3/4$	$9/10 \leq B'' \leq 9/8$	$15/16 \leq B'' < 1$

SRT-Division mit redundanter Codierung des Partialrests

Moderne SRT-Dividierer verwenden Carry-Save-Techniken zur Beschleunigung der Subtraktion bzw. Addition.

Die Freiheitsgrade in der Quotientenziffernauswahl werden genutzt, um die Anzahl der zur Selektion benötigten führenden Ziffern des Partialrests klein zu halten.

Durch weitere Techniken können auch mehrere Divisionsschritte überlappt werden.

Zur weiteren Beschleunigung der Division kann der Quotient zunächst in einer Signed-Digit-Darstellung mit größerer Zahlenbasis berechnet werden (Radix-2^k-SRT).

Die Quotientenziffern der redundanten Darstellung können on-the-fly in klassische Binärdarstellung überführt werden.

Radix-2-SRT-Division in UInt_2 mit redundant codiertem Partialrest

Die Vorschrift

$$R^{(i+1)} = 2 \times R^{(i)} - q_i \times B', \qquad q_i = \begin{cases} 1 & \text{falls } 0 \leq 2 \times R^{(i)} \\ 0 & \text{falls } -B' \leq 2 \times R^{(i)} < B' \\ \bar{1} & \text{falls } 2 \times R^{(i)} < 0 \end{cases}$$

mit der Eigenschaft $-2^{2l} < -B' \leq R^{(i)} < B' < 2^{2l}$ kann für normalisierten Divisor B verfeinert werden zu

$$R^{(i+1)} = 2 \times R^{(i)} - q_i \times B', \qquad q_i = \begin{cases} 1 & \text{falls } 0 \leq R^{(i)} \\ 0 & \text{falls } -2^{2l-2} \leq R^{(i)} < 2^{2l-2} \\ \bar{1} & \text{falls } R^{(i)} < 0 \end{cases}$$

und dies weiter zu

$$R^{(i+1)} = 2 \times R^{(i)} - q_i \times B', \qquad q_i = \begin{cases} 1 & \text{falls } 0 \leq R^{(i)} \\ 0 & \text{falls } -2^{2l-2} \leq R^{(i)} < 2^{2l-3} \\ \bar{1} & \text{falls } R^{(i)} < -2^{2l-3} \end{cases}$$

Radix-2-SRT-Division in UInt_2 mit redundant codiertem Partialrest

Liegt die Carry-Save-Darstellung $R^{(i)} = S^{(i)} + C^{(i)}$ mit $R^{(i)}, S^{(i)}, C^{(i)} \in \mathrm{Int}_2(2l+1)$ vor, so kann in $\mathrm{Int}_2(4)$ die Testgröße $\overline{R}^{(i)} = S^{(i)}_{[2l-3:2l]} + C^{(i)}_{[2l-3:2l]}$ berechnet werden.

Es gilt dann $2^{2l-3} \times \overline{R}^{(i)} \leq R^{(i)} < 2^{2l-3} \times \overline{R}^{(i)} + 2^{2l-2}$.

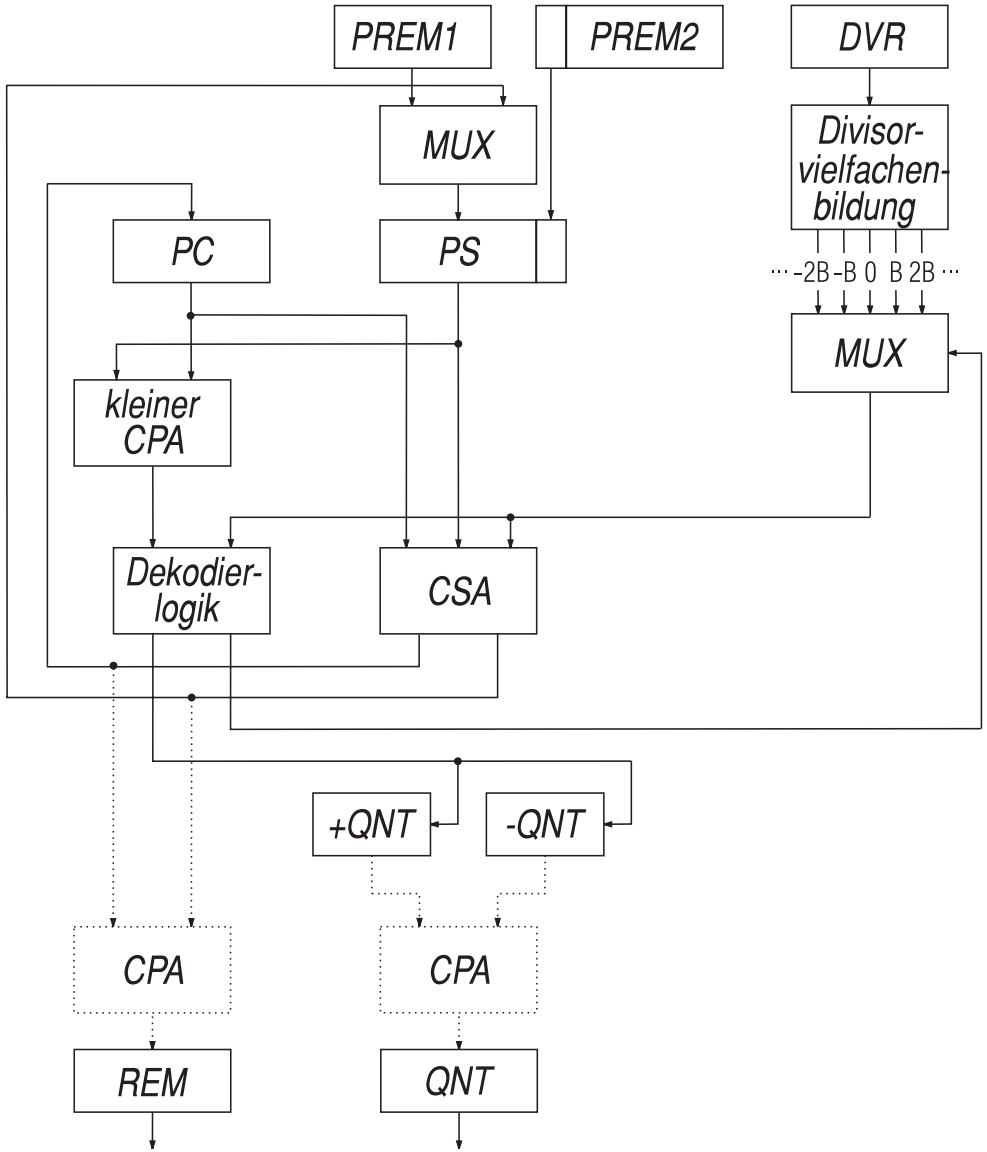
Die Vorschrift

$$R^{(i+1)} = 2 \times R^{(i)} - q_i \times B', \qquad q_i = \begin{cases} 1 & \text{falls } 0 \leq R^{(i)} \\ 0 & \text{falls } -2^{2l-2} \leq R^{(i)} < 2^{2l-3} \\ \bar{1} & \text{falls } R^{(i)} < -2^{2l-3} \end{cases}$$

wird dann konsistent erfüllt durch die Vorschrift

$$R^{(i+1)} = 2 \times R^{(i)} - q_i \times B', \qquad q_i = \begin{cases} 1 & \text{falls } 0 \leq \overline{R}^{(i)} & (\overline{R}^{(i)}_3 = 0) \\ 0 & \text{falls } -2 \leq \overline{R}^{(i)} \leq -1 & (\overline{R}^{(i)}_3 = \overline{R}^{(i)}_2 = \overline{R}^{(i)}_1 = 1) \\ \bar{1} & \text{falls } \overline{R}^{(i)} \leq -3 & \end{cases}$$

Sequentieller SRT-Dividierer



Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Iterative Division, Quadratwurzelberechnung

Verfahren der iterativen Division

Es wird eine Folge von Zahlen berechnet, die schnell gegen den gesuchten Quotienten (oder Größen, aus denen dieser leicht berechnet werden kann) konvergiert.

Zur Durchführung wird meist auf die wiederholte Anwendung der Multiplikation zurückgegriffen.

Gründe für die Verwendung iterativer Divisionsverfahren sind:

- Der Einsatz des schnellsten Multiplikationsverfahrens zusammen mit dem schnellsten Divisionsverfahrens ist evtl. aus Kostengründen nicht möglich.
- Ein schneller Multiplizierer wird kostengünstiger, wenn er auch im Rahmen der Division einsetzbar ist.
- Die Prinzipien der iterativen Division sind auch bei anderen Operationen (z. B. Wurzelberechnung) anwendbar.
- Die Geschwindigkeit iterativer Verfahren kann mit der sequenzieller Divisionsverfahren durchaus konkurrieren.
- Ein eventueller Zeitverlust bei iterativer Division wirkt sich u. U. kaum aus, da die Division recht selten gebraucht wird.

Division durch Konvergenz

In den bisher besprochenen Divisionsverfahren ist die Anzahl der Schritte im schlechtesten Fall proportional zu l, aber jeder Schritt besteht im Prinzip nur aus einer Addition oder Subtraktion.

In Verfahren der Division durch Konvergenz beträgt die Anzahl der Schritte nur noch etwa $\log_2 l$, jeder Schritt enthält aber eine Multiplikation und ist somit langsamer und aufwendiger als in den zuvor studierten Verfahren.

Es kommt deshalb bei der Division durch Konvergenz vor allem darauf an, einen möglichst schnellen Multiplizierer zu benutzen.

Weitere Aspekte sind die Anzahl der benötigten Iterationen und die Komplexität eines Iterationsschritts.

Prinzip der Division durch wiederholte Multiplikation

Werden Divisor und Dividend sukzessive mit einer Folge von Faktoren so multipliziert, dass die Divisorfolge gegen 1 strebt, so konvergiert die Dividendenfolge gegen den Quotienten Q:

$$Q = \frac{A}{B} = \frac{A \times X^{(0)} \times X^{(1)} \times X^{(m-1)}}{B \times X^{(0)} \times X^{(1)} \times X^{(m-1)}} \rightarrow \frac{Q}{1}$$

Falls auch der Rest R benötigt wird, muss er in einem separaten Schritt durch eine Multiplikation und eine Subtraktion aus der Formel $R = A - B \times Q$ berechnet werden.

Zentraler Punkt bei der Durchführung des Verfahrens ist die Auswahl der Faktoren $X^{(i)}$:

- Die Konvergenz der Divisorfolge gegen 1 muss in allen Fällen garantiert sein.
- Die Konvergenz sollte möglichst schnell erfolgen.
- Die Multiplikationen mit den Faktoren $X^{(i)}$ sollten einfach sein.
- Die Faktoren $X^{(i)}$ sollten leicht bestimmbar sein.
- Das Abbruchkriterium sollte einfach sein.

Auswahl der Skalierungsfaktoren

Sei der Divisor eine normalisierte binäre Nachkommazahl, $B = 0{,}1xxxx\ldots$ (0, wird dabei nicht mitgespeichert). Diese Situation kann durch arithmetisches Verschieben von B (und zugleich A) stets hergestellt werden und entspricht einer Uminterpretation der Gewichtsfaktoren.

Es gilt $1/2 \leq B < 1$, also $B = 1 - Z$ mit $0 < Z \leq 1/2$

Wählt man $X^{(0)} = 1 + Z$, so erhält man einen neuen Divisor

$$B^{(1)} = B \times X^{(0)} = (1 - Z) \times (1 + Z) = 1 - Z^2$$

Es gilt also $B < B^{(1)} < 1$, und insbesondere $3/4 \leq B^{(1)}$

Im 2. Schritt wählt man $X^{(1)} = 1 + Z^2$, und erhält

$$B^{(2)} = B^{(1)} \times X^{(1)} = (1 - Z^2) \times (1 + Z^2) = 1 - Z^4$$

mit $B < B^{(1)} < B^{(2)} < 1$ und $15/16 \leq B^{(2)}$

Allgemein hat im $(i+1)$-ten Schritt der Divisor die Form $B^{(i)} = 1 - Z^{2^i}$,
und $B^{(i)}$ hat deswegen mindestens 2^i führende Einsen.

Der nächste Faktor ist $X^{(i)} = 1 + Z^{2^i}$, und man erhält

$$B^{(i+1)} = B^{(i)} \times X^{(i)} = 1 - Z^{2^{i+1}}$$

mit 2^{i+1} führenden Einsen.

Konvergenzverhalten

Würden die Divisoren $B^{(i)}$ mit wachsender Genauigkeit berechnet, gälte $\lim_{i \to \infty} B^{(i)} = 1$.

Wegen der auftretenden Rundungsfehler konvergiert die Divisorfolge aber eventuell nur gegen $1 - \mathit{ulp}$ (*unit in the last place*, d. h. hier $\mathit{ulp} = 2^{-l}$).

Da sich die Anzahl der führenden Einsen von $B^{(i)}$ in jedem Schritt verdoppelt, wächst die Anzahl der Schritte bis zur Konvergenz nur logarithmisch in l (quadratische Konvergenz).

Division durch wiederholte Multiplikation (Goldschmidt-Verfahren)

Die Faktoren $X^{(i)}$ können direkt aus $B^{(i)}$ berechnet werden:
$X^{(i)} = 2 - B^{(i)}$ (d. h. $X^{(i)}$ ist das 2-Komplement von $B^{(i)}$)

Man erhält als Algorithmus (Anderson-Earle-Goldschmidt-Powers)

$$\begin{aligned} A^{(0)} &= A \\ B^{(0)} &= B \\ X^{(i)} &= 2 - B^{(i)} \\ A^{(i+1)} &= A^{(i)} \times X^{(i)} \\ B^{(i+1)} &= B^{(i)} \times X^{(i)} \end{aligned}$$

Beispiel: 15-Bit Dividend $A = 0{,}011010000000000_2 = 0{,}40625_{10}$
und Divisor $B = 0{,}110000000000000_2 = 0{,}75_{10}$

i	$X^{(i)} = 2 - B^{(i)}$	$A^{(i+1)} = A^{(i)} \times X^{(i)}$	$B^{(i+1)} = B^{(i)} \times X^{(i)}$
0	1,010000000000000	0,100000100000000	0,111100000000000
1	1,000100000000000	0,100010100010000	0,111111110000000
2	1,000000010000000	0,100010101010101	0,111111111111111

Konvergenz nach 3 Schritten (SRT-Division: 15 Schritte)

Der berechnete Quotient $Q = A^{(3)} = 0{,}100010101010101_2 \approx 0{,}54165_{10}$ stellt die genaueste Approximation durch eine 15-Bit-Nachkommazahl dar; der exakte Quotient wäre die unendlich periodische Zahl $0{,}100010101010\overline{101} = 0{,}5416\overline{6}$

Beschleunigung der Konvergenz

Die sukzessive Multiplikation von A und B mit einer Folge von Faktoren $X^{(i)}$ wird mit wachsendem i immer effizienter: Der Effekt der Multiplikation mit $X^{(i)}$ ist ebensogroß wie der aller vorhergehenden Faktoren $X^{(0)}, X^{(1)}, \ldots, X^{(i-1)}$ zusammen.

Man fasst deshalb die ersten Schritte besser zu einem einzigen (effizienteren) Schritt zusammen, indem man einen Faktor $X_*^{(0)}$ benutzt, der schon eine relativ große Anzahl führender Einsen in $B^{(1)}$ erzeugt.

Der weitere Ablauf des Verfahrens erfolgt dann nach dem beschriebenen Algorithmus.

Der Faktor $X_*^{(0)}$ kann aus B mittels einer Wertetabelle bestimmt werden.

Um den Speicher für die Wertetabelle möglichst klein zu halten, berücksichtigt man nur so viele führende Ziffern von B bzw. $X_*^{(0)}$, wie zur Erzeugung der gewünschten Anzahl von k führenden Einsen in $B^{(1)}$ unbedingt nötig sind (lt. Parhami sind dies jeweils nur k Bits).

Der Faktor $X_*^{(0)}$ ist eine Approximation von $(1 + Z) \times (1 + Z^2) \times \ldots \times (1 + Z^{2^j})$ mit einem geeigneten j, $k \approx 2^{j+1}$.

Anmerkung: Da Divisor *und* Dividend mit demselben Faktor $X_*^{(0)}$ multipliziert werden, ist die Güte der Approximation eher unkritisch.

Beschleunigung der Konvergenz (Beispiel)

Die Nachkommateile der langen Gleitkommazahlen in der IBM 360/91 sind 56 Bit lang.

Statt 6 Schritten (d. h. 11 Multiplikationen) benötigt man nur 4 Schritte (d. h. 7 Multiplikationen), falls im ersten Schritt bereits 7 führende Einsen in $B^{(1)}$ garantiert werden.

Dazu genügen die ersten 10 Bit von $X_*^{(0)}$, und zu deren Berechnung die ersten 7 Bit von B.

Man benötigt also ein ROM mit 128×10 Bit.

Weitere Beschleunigung

Weitere Beschleunigung möglich durch Multiplikatoren mit geringerer Anzahl von Stellen für die ersten Schritte.

Soll in Schritt $(i+1)$ ein Divisor $B^{(i+1)}$ mit $\alpha \geq 2^{i+1}$ führenden Einsen aus einem Divisor $B^{(i)}$ mit mindestens $\alpha/2$ führenden Einsen erzeugt werden, genügt es, als Faktor $X_*^{(i)}$ das 2-Komplement der ersten α Bits von $B^{(i)}$ zu benutzen.

In jedem solchen verkürzten $X_*^{(i)}$ besitzen mindestens die ersten $\alpha/2$ führenden Bits alle denselben Wert 1, die Anzahl der Teilprodukte des Multiplizierers kann deshalb (notfalls über den Umweg einer ternären Codierung) auf höchstens $\alpha/2$ gesenkt werden.

Die Berechnung von $A^{(i)}$ und $B^{(i)}$ kann parallel oder – bei gepipelinetem Multiplizierer – überlappt erfolgen.

Division durch Kehrwertbildung

Bei Division durch Kehrwertbildung berechnet man den Kehrwert des Divisors und multipliziert ihn anschließend mit dem Dividenden.

Dies ist besonders effizient, wenn mehrere Divisionen mit demselben Divisor auszuführen sind.

Die Genauigkeit des Quotienten bei Division durch Kehrwertbildung ist meist kleiner als bei Division durch sukzessive Addition/Subtraktion und muss gegebenenfalls durch künstliche Verlängerung des Divisors und Dividenden auf den gewünschten Grad gesteigert werden.

Newton-Verfahren

Sei $f(x)$ eine stetig differenzierbare reelle Funktion mit Nullstelle s, für die $f'(s) \neq 0$ gilt.

Dann ist s ein Fixpunkt der Iterationsfunktion $F(x) = x - f(x)/f'(x)$.

Für Startwerte x_0 aus einer geeigneten Umgebung von s konvergiert die Folge der $\{x_i\}$ mit $x_{i+1} = F(x_i)$ gegen den Fixpunkt s. Diese Konvergenz ist quadratisch.

Da wir den Wert $1/B$ bestimmen wollen, wählen wir $f(x) = 1/x - B$.

$K = 1/B$ ist die einzige Nullstelle von f, die Voraussetzungen an f sind erfüllt.

Mit $f'(x) = -1/x^2$ ergibt sich die Iterationsfunktion $F(x) = x \times (2 - B \times x)$.

Division durch Kehrwertbildung (Newton-Raphson-Verfahren)

Der Kehrwert eines normalisierten Divisors kann mit dem Newton-Raphson-Verfahren durch folgende Iterationsvorschrift mit quadratischer Konvergenz berechnet werden:

$$K^{(i+1)} \quad = \quad K^{(i)} \times (2 - B \times K^{(i)})$$

Für den Fehler $\delta_i = 1/B - K^{(i)}$ gilt bei Rechnung ohne Rundungsfehler: $\delta_{i+1} = B \times \delta_i^2$

Zur Konvergenz notwendig und hinreichend ist dann $|\delta_0| < 1/B$.

Dies ist gleichbedeutend mit $0 < K^{(0)} < 2/B$, wegen $B < 1$ genügt $0 < K^{(0)} \leq 2$.

Für $K^{(0)} = 1$ erhalten wir die Abschätzung $0 < \delta_i \leq 2^{1-2^i}$.

Für $K^{(0)} = 3/2$ erhalten wir die bessere Abschätzung $0 < \delta_i \leq 2^{1-2^{i+1}}$.
Empfohlener Wert lt. Parhami: $K^{(0)} = 4(\sqrt{3} - 1) - 2 \times B$ (impliziert $|\delta_0| < 0.072$)

Ein noch besserer Startwert $K_*^{(0)}$ kann zur Beschleunigung des Verfahrens direkt mittels einer Wertetabelle aus B erzeugt werden.

Nützlich für Abschätzungen: $B\delta_i = (B\delta_0)^{2^i}$

Division durch Kehrwertbildung (Beispiel)

Die Division durch Kehrwertbildung kann äußerst schnell durchgeführt werden.

Beispiel: Berechnung des Kehrwerts der Mantisse einer normalisierten doppelt genauen Gleitkommazahl entsprechend dem Standard IEEE-754 (53 Bits Signifikand).

16 führende Ziffern von B (wovon die erste konstant 1 ist) genügen, um aus einer Wertetabelle ($32K \times 16$-bit ROM) eine Approximation $K_*^{(0)}$ mit $|K_*^{(0)} - 1/B| < 1.5 \times 2^{-16}$ zu entnehmen.
Mit zwei Newton-Raphson-Iterationen ergibt sich die gewünschte Genauigkeit von 53 Bit.

Es werden ein Speicherzugriff, 4 Multiplikationen und 2 Komplement-Operationen benötigt.

Einige Multiplikationen können mit reduzierter Genauigkeit durchgeführt werden.
Statt des 2-Komplements kann auch das schnellere 1-Komplement verwendet werden.

Das Verfahren von Ferrari

Das Verfahren von Ferrari (verallgemeinertes Newton-Verfahren der Ordnung 3) benötigt zur Erzielung derselben Genauigkeit in der Regel weniger Operationen als das Newton-Raphson-Verfahren.

Der Startwert $K^{(0)}$ wird einer Wertetabelle entnommen,
die weiteren Folgenglieder berechnen sich aus der Rekursion

$$\begin{aligned} Z^{(i+1)} &= 1 - B \times K^{(i)} \\ K^{(i+1)} &= (1 + Z^{(i+1)} \times (1 + Z^{(i+1)})) \times K^{(i)} \end{aligned}$$

Vergleich der Verfahren

Das Newton-Raphson-Verfahren und das Verfahren von Ferrari besitzen den Nachteil, dass alle Operationen eines Schritts aufeinander aufbauen und deshalb nur sequentiell abgewickelt werden können.

Das Verfahren von Anderson-Earle-Goldschmidt-Powers erlaubt die parallele oder überlappte Ausführung der Multiplikationen eines Schritts.

Außerdem liefert das Verfahren eine Folge, die gegen den Quotienten $Q = A/B$ konvergiert, nicht nur gegen den Kehrwert $K = 1/B$, und spart so Multiplikationen.

Noch schnellere Varianten des Verfahrens von Anderson-Earle-Goldschmidt-Powers sind beschrieben in:
M. D. Ercegovac, et al.: Improving Goldschmidt division, square root, and square root reciprocal. IEEE Transactions on Computers, Vol. 49, No. 7 (Juli 2000) pp. 759–763.

Varianten vorzeichenbehafteter Division

Ausgehend von den Operanden $A, B \in \mathbb{Q}, B \neq 0$ sollen $Q \in \mathbb{Z}$ und/oder $R \in \mathbb{Q}$ berechnet werden mit $A = R + B \times Q$ und den alternativen, jeweils eindeutigen Nebenbedingungen

(a) $|R| < |B|$, $\text{sgn}(R) = \text{sgn}(A)$ (minimiert $|Q|$)

(b) $|R|$ minimal

(c) $0 \leq R$, R minimal

(d) $Q = \lfloor A/B \rfloor$

(e) $Q = \lceil A/B \rceil$

Für $A, B > 0$ fallen (a), (c) und (d) zusammen.

Für $A > 0, B < 0$ fallen (a), (c) und (e) zusammen.

Für $A < 0, B > 0$ fallen (a) und (e) einerseits, (c) und (d) andererseits zusammen.

Für $A, B < 0$ fallen (a) und (d) einerseits, (c) und (e) andererseits zusammen.

Implementierung der Varianten

Verfahren, die mit vorzeichenbehafteten Operanden arbeiten, werden meist mit einem abschließenden Korrekturschritt versehen, der Situation (a) herstellt.

Abhängig von den Vorzeichen der Operanden ist damit gleichzeitig Situation (d) oder alternativ Situation (e) erledigt. Der jeweils andere Fall muss separat berechnet werden, etwa durch die Beziehungen $\lceil A/B \rceil = \lfloor A/B \rfloor$, falls $R = 0$, und $\lceil A/B \rceil = \lfloor A/B \rfloor + 1$ sonst.

Alternativ zu (a) können auch (b) oder (c) durch eine Schlusskorrektur erzwungen werden.

Bei vorzeichenloser Berechnung erzeugen Restoring-Division, Non-performing-Division und Goldschmidt-Verfahren implizit die Eigenschaft (a) (und damit auch (c) und (d)).

Sind nicht alle Vorzeichenkombinationen direkt implementiert, kann z. B. die Beziehungen $\lfloor -Z \rfloor = -\lceil Z \rceil$ benutzt werden.

Die Eigenschaft (b) kann auch direkt berechnet werden, indem statt $\lfloor A/B \rfloor$ die Division $\lfloor (A + K)/B \rfloor$ durchgeführt wird und der Rest R anschließend in $R - K$ abgeändert wird.

Dabei ist $K = (B - 1)/2$ für diejenigen B, deren Darstellung mit einer Eins endet, und $K = B/2$ oder $K = B/2 - 1$ in den übrigen Fällen.

Ausschließliche Berechnung nicht negativer Reste

Soll nur ein nicht negativer Rest ohne zugehörigen Quotienten berechnet werden
($A \in \mathrm{UInt}_2(m)$, normalisiertes $B \in \mathrm{UInt}_2(l)$, $m \geq l$, $0 \leq R < B$),
braucht der Quotient nicht beschränkt zu werden.

Berechnung mit dem Non-Restoring-Verfahren:

$$
\begin{aligned}
B' &= 2^{m-l} \times B \\
R^{(0)} &= A \\
R^{(i+1)} &= \begin{cases} 2 \times R^{(i)} - B' & \text{falls } R^{(i)} \geq 0 \\ 2 \times R^{(i)} + B' & \text{sonst} \end{cases} \\
R^{(m-l+2)} &= \begin{cases} R^{(m-l+1)} & \text{falls } R^{(m-l+1)} \geq 0 \\ R^{(m-l+1)} + B' & \text{sonst} \end{cases}
\end{aligned}
$$

Entsprechend kann auch für gebrochene Operanden vorgegangen werden.

Quadratwurzelberechnung

Die Berechnung der Quadratwurzel gilt oft als die „fünfte Grundrechnungsart“ (z. B. vorgeschrieben im Standard IEEE-754).

Verfahren für die Berechnung von Quadratwurzeln können durch Modifikation praktisch aller Divisionsverfahren gewonnen werden.

Wird ein Dividierer zusammen mit einem Radizierer implementiert, ist der zusätzliche Aufwand für den Radizierer sehr gering.

Aus dem Radikanden $A \in \mathrm{UInt}_2(2l)$ errechnen wir eine Wurzel $Q \in \mathrm{UInt}_2(l)$ und einen minimalen Rest $R \geq 0$ mit $A = Q^2 + R$.

Aus der Minimalität des Rests R folgt die Maximalität von Q sowie die Beziehung $R \leq 2 \times Q$, also $R \in \mathrm{UInt}_2(l+1)$ ausreichend.

Durch Skalierung von A und R mit 2^{-2l} sowie von Q mit 2^{-l} ergibt sich der Radikand als binäre Nachkommazahl $A = 0{,}a_1\,a_2 \ldots a_{2l}$, die Quadratwurzel als $Q = 0{,}q_1\,q_2 \ldots q_l$.

Non-performing-Quadratwurzelberechnung

Es sei $Q^{(i)} = \sum_{k=1}^{i} q_k \times 2^{-k}$, also insbesondere $Q^{(0)} = 0$ und $Q^{(l)} = Q$.

$R^{(i)}$ bezeichnet den Partialrest zu $Q^{(i)}$, wobei $R^{(i)} = A - (Q^{(i)})^2$ und $R^{(0)} = A$ gilt.

Das Quotientenbit q_i besitzt genau dann den Wert 1, wenn damit $R^{(i)} \geq 0$ wird,
also $R^{(i-1)} \geq 2^{1-i} Q^{(i-1)} + 2^{-2i} = 2^{1-i}(Q^{(i-1)} + 2^{-i-1})$.

Man beachte, dass $Q^{(i-1)}$ eine Nullziffer zum Gewicht 2^{-i-1} besitzt.

Durch Skalierung von $R^{(i)}$ mit 2^i ergibt sich der Algorithmus

$$R_*^{(i)} = 2 \times R^{(i-1)} - 2 \times Q^{(i-1)} - 2^{-i}$$

$$(q_i, R^{(i)}) = \begin{cases} (1, R_*^{(i)}) & \text{falls } R_*^{(i)} \geq 0 \\ (0, 2 \times R^{(i-1)}) & \text{falls } R_*^{(i)} < 0 \end{cases}$$

Es gilt die Beziehung $R^{(i)} = 2 \times R^{(i-1)} - q_i \times (2 \times Q^{(i-1)} + 2^{-i})$,
aber auch $R^{(i)} = 2 \times R^{(i-1)} - q_i \times (2 \times Q^{(i-1)} + q_i \times 2^{-i})$.

Zur Speicherung von R und Q genügt ein Register der Länge $2l + 1$ Bit.

Non-performing-Quadratwurzelberechnung (Beweis)

$$
\begin{aligned}
R^{(l)} &= 2 \times R^{(l-1)} - q_l \times (2 \times Q^{(l-1)} + q_l \times 2^{-l}) \\
&= 4 \times R^{(l-2)} - 2 \times q_{l-1} \times (2 \times Q^{(l-2)} + q_{l-1} \times 2^{1-l}) \\
&\quad - q_l \times (2 \times Q^{(l-1)} + q_l \times 2^{-l}) \quad = \quad \ldots \\
&= 2^l \times (R^{(0)} - [(q_1 \times 2^{-1})^2 + (q_2 \times 2^{-2})^2 + \cdots + (q_l \times 2^{-l})^2] \\
&\quad - [2 \times q_2 \times 2^{-2} \times q_1 \times 2^{-1} + \cdots + 2 \times q_l \times 2^{-l} \times \sum_{i=1}^{l-1} q_i \times 2^{-i}]) \\
&= 2^l \times (A - (\sum_{i=1}^{l} q_i \times 2^{-i})^2) \quad = \quad 2^l \times (A - Q^2)
\end{aligned}
$$

Damit ist $R = R^{(l)} \times 2^{-l}$ der endgültige Rest.

Non-performing-Quadratwurzelberechnung (Beispiel)

$R^{(0)} = A$		0	0	,	1	0	1	1	$A = 11/16 = 176/256$
$2 \times R^{(0)}$		0	1	,	0	1	1	0	
$-(0 + 2^{-1})$	$-$	0	0	,	1	0	0	0	
$R^{(1)}$		0	0	,	1	1	1	0	$q_1 = 1$, $Q^{(1)} = 0{,}1$
$2 \times R^{(1)}$		0	1	,	1	1	0	0	
$-(2 \times Q^{(1)} + 2^{-2})$	$-$	0	1	,	0	1	0	0	
$R^{(2)}$		0	0	,	1	0	0	0	$q_2 = 1$, $Q^{(2)} = 0{,}11$
$2 \times R^{(2)}$		0	1	,	0	0	0	0	kleiner als $2 \times Q^{(2)} + 2^{-3} = 1{,}101$
$R^{(3)} = 2 \times R^{(2)}$		0	1	,	0	0	0	0	$q_3 = 0$, $Q^{(3)} = 0{,}110$
$2 \times R^{(3)}$		1	0	,	0	0	0	0	kein negatives Vorzeichen!
$-(2 \times Q^{(3)} + 2^{-4})$	$-$	0	1	,	1	0	0	1	
$R^{(4)}$		0	0	,	0	1	1	1	$q_4 = 1$, $Q^{(4)} = 0{,}1101$

Es ergibt sich $Q = 0{,}1101_2 = 13/16$ und $R = R^{(4)} \times 2^{-4} = 7/256 = (176 - 169)/256 = A - Q^2$.

Non-restoring-Quadratwurzelberechnung

$$
\begin{aligned}
q_i &= \begin{cases} 1 & \text{falls } R^{(i-1)} \geq 0 \\ \bar{1} & \text{falls } R^{(i-1)} < 0 \end{cases} \\
R^{(i)} &= 2 \times R^{(i-1)} - q_i \times (2 \times Q^{(i-1)} + q_i \times 2^{-i})
\end{aligned}
$$

Es folgt $-2 < R^{(i)} < 2$.

Die Ziffern von Q müssen abschließend wieder in eine Binärdarstellung umgewandelt werden, genau wie für den Quotienten einer Non-restoring-Division.

Falls der Rest negativ wurde, muss noch eine Schlusskorrektur erfolgen.

Non-restoring-Quadratwurzelberechnung (Beispiel)

$R^{(0)} = A$		0	0	,	0	1	1	0	0	1	$A = 25/64$
$2 \times R^{(0)}$		0	0	,	1	1	0	0	1	0	$q_1 = 1$, $Q^{(1)} = 0{,}1$
$-(0 + 2^{-1})$	$-$	0	0	,	1	0	0	0	0	0	
$R^{(1)}$		0	0	,	0	1	0	0	1	0	
$2 \times R^{(1)}$		0	0	,	1	0	0	1	0	0	$q_2 = 1$, $Q^{(2)} = 0{,}11$
$-(2 \times Q^{(1)} + 2^{-2})$	$-$	0	1	,	0	1	0	0	0	0	
$R^{(2)}$		1	1	,	0	1	0	1	0	0	
$2 \times R^{(2)}$		1	0	,	1	0	1	0	0	0	$q_3 = \bar{1}$, $Q^{(3)} = 0{,}11\bar{1}$
$+(2 \times Q^{(2)} - 2^{-3})$	$+$	0	1	,	1	0	$\bar{1}$	0	0	0	
$R^{(3)}$		0	0	,	0	0	0	0	0	0	

Es ergibt sich $Q = 0{,}11\bar{1}_2 = 0{,}101_2 = 5/8$ und $R = 0$.

SRT-Quadratwurzelberechnung

Eine ähnlich einfache Bedingung für q_i wie bei der SRT-Division erhält man durch folgenden Algorithmus für $1 \geq A \geq 1/4$:

$$q_i = \begin{cases} 1 & \text{falls} \quad 1/2 \leq R^{(i-1)} \\ 0 & \text{falls} \quad -1/2 \leq R^{(i-1)} < 1/2 \\ \bar{1} & \text{falls} \quad R^{(i-1)} < -1/2 \end{cases}$$

$$R^{(i)} = 2 \times R^{(i-1)} - q_i \times (2 \times Q^{(i-1)} + q_i \times 2^{-i})$$

Abweichend davon wird im ersten Schritt immer $q_1 = 1$ gesetzt.

SRT-Quadratwurzelberechnung (Beispiel)

$R^{(0)} = A$		0	0	,	0	1	1	1	1	0	1	$A = 61/128$
$2 \times R^{(0)}$		0	0	,	1	1	1	1	0	1	0	$q_1 = 1$, $Q^{(1)} = 0{,}1$
$-(0 + 2^{-1})$	$-$	0	0	,	1	0	0	0	0	0	0	
$R^{(1)}$		0	0	,	0	1	1	1	0	1	0	
$2 \times R^{(1)}$		0	0	,	1	1	1	0	1	0	0	$q_2 = 0$, $Q^{(2)} = 0{,}10$
$R^{(2)}$		0	0	,	1	1	1	0	1	0	0	
$2 \times R^{(2)}$		0	1	,	1	1	0	1	0	0	0	$q_3 = 1$, $Q^{(3)} = 0{,}101$
$-(2 \times Q^{(2)} + 2^{-3})$	$-$	0	1	,	0	0	1	0	0	0	0	
$R^{(3)}$		0	0	,	1	0	1	1	0	0	0	
$2 \times R^{(3)}$		0	1	,	0	1	1	0	0	0	0	$q_4 = 1$, $Q^{(4)} = 0{,}1011$
$-(2 \times Q^{(3)} + 2^{-4})$	$-$	0	1	,	0	1	0	1	0	0	0	
$R^{(4)}$		0	0	,	0	0	0	1	0	0	0	

Es ergibt sich $Q = 0{,}1101_2 = 11/16$ und $R = R^{(4)} \times 2^{-4} = 1/256 = (122 - 121)/256 = A - Q^2$.

Iterative Berechnung der Quadratwurzel

Die Quadratwurzel kann mittels des Newton-Verfahrens nach der Iterationsvorschrift

$$Q^{(i+1)} = \frac{1}{2} \times \left(Q^{(i)} + \frac{A}{Q^{(i)}} \right) \qquad \text{(abgeleitet aus der Funktion } f(Q) = Q^2 - A\text{)}$$

berechnet werden.

Für Argumente $1 > A \geq 1/4$ und den Startwert $Q^{(0)} = 1$ konvergiert die Folge quadratisch von oben gegen $\sqrt{A}$.

Der Fehler $\delta_i = 1 - \sqrt{A}$ zeigt das Verhalten $\delta_{i+1} = \delta_i^2/(2Q^{(i)})$.

$Q^{(1)} = (1 + A)/2$ kann auch ohne die Verwendung arithmetischer Operationen berechnet oder durch einen noch genaueren Tabellenwert ersetzt werden.

Nachteilig ist die vorkommende Divisionsoperation.

Weitere Iterationsvorschriften

Der benötigte Kehrwert $1/Q^{(i)}$ kann selbst wieder iterativ durch eine Folge $K^{(i)}$ approximiert werden, es ergeben sich zwei verschränkte Rekurrenzgleichungen:

$$\begin{aligned} Q^{(i+1)} &= \frac{1}{2} \times \left(Q^{(i)} + A \times K^{(i)} \right) \\ K^{(i+1)} &= K^{(i)} \times (2 - Q^{(i)} \times K^{(i)}) \end{aligned}$$

Als Startwerte für $1 > A \geq 1/4$ eignen sich $Q^{(0)} = (1 + A)/2$ und $K^{(0)} = 1$.

Die Konvergenzgeschwindigkeit ist besser als linear, aber schlechter als quadratisch.

Alternativ kann die Iterationsvorschrift äquivalent in

$$Q^{(i+1)} = Q^{(i)} + \frac{1}{2 \times Q^{(i)}} \times \left(A - (Q^{(i)})^2 \right)$$

umgeschrieben und $1/(2 \times Q^{(i)})$ mittels einer Wertetabelle approximativ bestimmt werden.

Da aber in jedem Schritt ein ungenauer Wert aus der Tabelle benutzt wird, konvergiert das Verfahren ebenfalls nicht mehr quadratisch.

Divisionsfreie Iterationsverfahren

Bessere Ergebnisse erhält man mit einer der Formeln

(A) $Z^{(i+1)} = \frac{Z^{(i)}}{2} \times \left(3 - A \times (Z^{(i)})^2\right)$ (bzw. $Z^{(i+1)} = Z^{(i)} + Z^{(i)}(1 - A \times (Z^{(i)})^2)/2$)

(B) $Z^{(i+1)} = \frac{1}{2} \times \left[Z^{(i)} + 2 \times Z^{(i)} - (A \times Z^{(i)}) \times (Z^{(i)})^2\right]$

In beiden Fällen ergibt sich quadratische Konvergenz gegen den Wert $1/\sqrt{A}$.

Den gesuchten Wert $\sqrt{A}$ erhält man durch Multiplikation des Fixpunktes mit A.

Die Verfahren unterscheiden sich in Aufwand und Parallelitätsgrad:

(A) 3 Multiplikationen, 1 Addition pro Schritt, keine Parallelausführung möglich.

(B) 3 Multiplikationen, 2 Additionen pro Schritt, bei Parallelausführung schneller als (A).

Verfahren (A) wird im Cray-2-Supercomputer benutzt.

Als Startwert $Z^{(0)}$ wird eine Schätzung von $1/\sqrt{A}$ einer Wertetabelle entnommen.

Zwei weitere Wertetabellen liefern $3 \times Z^{(0)}/2$ und $(Z^{(0)})^3/2$, aus denen durch eine Multiplikation und eine Subtraktion dann $Z^{(1)}$ folgt.

Nach einer weiteren, vollen Iteration liegt $Z^{(2)}$ vor.

Eine abschließende Multiplikation ergibt nun $\sqrt{A}$.

Quadratwurzelberechnung nach dem Goldschmidt-Verfahren

$$
\begin{aligned}
A^{(0)} &= A \\
Q^{(0)} &= A \\
\\
Z^{(i)} &= (3 - A^{(i)})/2 \\
A^{(i+1)} &= A^{(i)} \times Z^{(i)} \times Z^{(i)} \\
Q^{(i+1)} &= Q^{(i)} \times Z^{(i)}
\end{aligned}
$$

$Q^{(i)}$ strebt hier direkt gegen den gewünschten Wert $\sqrt{A}$:

Es gilt die Invariante $(Q^{(i)})^2 = A \times A^{(i)}$ und die $A^{(i)}$ konvergieren quadratisch gegen 1.

Mit $\varepsilon_i = 1 - A^{(i)}$ gilt $\varepsilon_{i+1} = \varepsilon_i^2(3 - \varepsilon_i)/4$.

Für $1 > A \geq 1/4$ folgt $0 < \varepsilon_i \leq 3/4$ und $9/16 \leq (3 - \varepsilon_i)/4 < 3/4$.

Eine Beschleunigung des Verfahrens ergibt sich durch Entnahme eines genaueren Korrekturfaktors $Z^{(0)} \approx 1/\sqrt{A}$ aus einer Wertetabelle.

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Auswertung von Standardfunktionen

Auswertung von Standardfunktionen

Neben den vier Grundoperationen $+$, $-$, $\times$, $/$ wird häufig auch die Implementierung von Standardfunktionen wie e^x, $\ln x$, $\sin x$, $\cos x$ ... gefordert.

Grundlegende Techniken zur Auswertung von Standardfunktionen sind:

- Entnahme aus Wertetabellen oder Zusammensetzung aus tabellierten Werten.
- Polynomapproximation, insbesondere Taylor-Reihen oder Tschebyschew-Approximation.
- Rationale Approximation.
- Gekoppelte Rekurrenzgleichungen, insbesondere additive oder multiplikative Normalisierung, vor allem das CORDIC-Verfahren.

Auswertung von Standardfunktionen: Wertetabellen

- Konzeptionell einfachste Methode zur Auswertung von Standardfunktionen.
- Interessant wegen hoher Packungsdichte von Speicherstrukturen.
- Tabelleninhalte leichter verifizierbar als Logikstrukturen.
- Vorteile hinsichtlich Robustheit, Flexibilität und Time-to-Market.

Wertetabellen: Direkter versus indirekter Zugriff

Für *direkten Tabellenzugriff* werden Werte einer Funktion $f(x_1, x_2, \ldots, x_k)$ für alle interessierenden Argumente (u. a. bestimmt durch die Darstellungsgenauigkeit) in einem Speicher abgelegt.

Sind die Argumente mit $m_1, m_2, \ldots, m_k$ Bits dargestellt, so ergibt sich ein l-Bit-Funktionswert durch Anlegen der konkatenierten Argumente als m-Bit-Adresse an ein $(2^m \times l)$-Bit-ROM,

$$m = \sum_{i=1}^{k} m_i.$$

Wegen des Aufwands kommt diese Methode nur für Zahlen geringer Genauigkeit in Frage.

Bei *indirektem Tabellenzugriff* werden die Argumente zunächst aufbereitet,
dann ein oder mehrere Tabellenzugriffe durchgeführt,
deren Ergebnisse schließlich zum gesuchten Funktionswert verknüpft werden.

Beispiel:

Berechnung von Produkten aus einer Tabelle von Quadraten: $x \times y = \dfrac{(x+y)^2 - (x-y)^2}{4}$

Interpolation von Tabellenwerten

Lineare Interpolation: $f(x) \approx f(y) + \dfrac{(x-y) \times [f(z) - f(y)]}{z - y}$ für alle $x \in [y, z]$

Zur Steigerung der Genauigkeit auch $f(x) \approx g(y) + \dfrac{(x-y) \times [g(z) - g(y)]}{z - y}$

Für eine effiziente Implementierung werden als Stützstellen all diejenigen Zahlen aus dem interessierenden Bereich gewählt, deren r niederwertigste Ziffern Null sind.

Außer der linearen Interpolation wird gelegentlich auch quadratische Interpolation angewandt.

Auswertung von Standardfunktionen mit kleinen Multiplizierern

Die Verwendung kleiner Multiplizierer wird möglich, indem Segmente der Repräsentation eines Arguments als (kurze) eigenständige Zahlen behandelt werden.

Beispiel:

Berechnung von e^X, wobei $X = 1 + A_1\lambda + A_2\lambda^2 + A_3\lambda^3 + A_4\lambda^4$ mit $\lambda = 2^{-k}$ und $A_i \in \mathrm{UInt}_2(k)$.

Ercegovac et al. schlagen in IEEE Computer, Vol. 49, No. 7, folgenden Algorithmus vor:

Approximiere e^A mit $A = A_2\lambda^2 + A_3\lambda^3 + A_4\lambda^4$ durch

$$1 + A + \frac{1}{2}A_2^2\lambda^4 + A_2A_3\lambda^5 + \frac{1}{6}A_2^3\lambda^6$$

Multipliziere das Ergebnis mit dem Wert von $e^{1+A_1\lambda}$ aus einer Wertetabelle.

Auswertung von Standardfunktionen: Taylor-Reihen

Genügend oft differenzierbare Funktionen können prinzipiell über eine Taylor-Reihe ausgewertet werden.

Taylorscher Satz: Es sei a aus dem offenen Intervall I. Besitzt die reelle Funktion $f\colon I \to \mathbb{R}$ auf I die Ableitungen $f', f'', \ldots, f^m$, so gibt es zu jedem $x \in I$ ein $\mu \in (0,1)$ mit

$$f(x) = \sum_{i=0}^{m-1} f^{(i)}(a) \times \frac{(x-a)^i}{i!} + f^m(a + \mu \times (x-a)) \times \frac{(x-a)^m}{m!}$$

Beispiel: $f(x) = e^x = \sum_{i=0}^{m-1} \frac{x^i}{i!} + R_m(x)$ mit $R_m(x) = e^{\mu \times x} \times \frac{x^m}{m!}$

- Ohne Rundungsfehler konvergiert die Reihenentwicklung für alle x.
- Die Konvergenzgeschwindigkeit ist dabei monoton in $|x|^{-1}$.
- Ob in beschränkter Genauigkeit Konvergenz eintritt, hängt von der Art der Rundung ab.
- Wegen Rundungsfehlern ist nicht klar, ob ein stationäres Ergebnis exakt gerundet ist.
- Tschebyschew-Approximation liefert im Allgemeinen bessere Resultate.

Auswertung von Standardfunktionen: Rationale Approximation

Rationale Approximation festen Grades ist häufig effizienter als Polynomapproximation.

Beispiel:

Die Funktion $f(x) = 2^x$ mit $|x| < 1$ wird gut approximiert durch den Quotienten von Polynomen des Grades 5 mit konstanten Koeffizienten a_i und b_i,

$$2^x \approx \frac{((((a_5 \times x + a_4) \times x + a_3) \times x + a_2) \times x + a_1) \times x + a_0}{((((b_5 \times x + b_4) \times x + b_3) \times x + b_2) \times x + b_1) \times x + b_0}$$

Der Aufwand im Hornerschema beträgt 10 Additionen, 10 Multiplikationen und eine Division.

Daraus lässt sich zum Beispiel leicht weiter berechnen:

$e^x = 2^{x \times \log_2 e} = 2^m \times 2^y$, wobei $x \times \log_2 e = m + y$ mit $m \in \mathbb{Z}$ und $|y| < 1$.

Gekoppelte Rekurrenzgleichungen

Ein wichtiger Ansatz zur Auswertung von Standardfunktionen basiert auf einem System gekoppelter Rekurrenzgleichungen:

- Jede einzelne Rekurrenzgleichung beschreibt einen iterativ in jedem Schritt des Verfahrens neu zu berechnenden Wert.
- Jeder Rekurrenzgleichung entspricht damit eine Folge von Zahlen.
- In den Rekurrenzgleichungen kommen gemeinsame Variablen (*Koppelterme*) vor.
- Wird durch spezielle Wahl der Koppelterme eine der Zahlenfolgen gegen einen vorgegebenen Grenzwert getrieben, streben die übrigen Zahlenfolgen entweder direkt gegen einen der gesuchten Funktionswerte oder gegen Zahlen, aus denen der Funktionswert leicht berechnet werden kann.
- Wichtige Formen von Algorithmen mit gekoppelten Rekurrenzgleichungen sind die additive Normalisierung (von speziellem Interesse ist hier das CORDIC-Verfahren) und die multiplikative Normalisierung.

Auswertung der Exponentialfunktion durch additive Normalisierung

Zur Auswertung der Exponentialfunktion in Festkomma-Darstellung wird ein System von Rekurrenzgleichungen mit Koppeltermen B_i benutzt:

$$\begin{aligned} X_{i+1} &= X_i - \ln B_i \\ Y_{i+1} &= Y_i \times B_i \end{aligned}$$

Die B_i werden so gewählt, dass die Folge der X_i gegen 0 konvergiert:

$$X_{i+1} = X_0 - \sum_{k=0}^{i} \ln B_k \to 0$$

In der Praxis wird die Folge wegen beschränkter Genauigkeit stationär, $X_i = 0$ für alle $i \geq m$:

$$X_m = 0 \Rightarrow X_0 = \sum_{i=0}^{m-1} \ln B_i = \ln \prod_{i=0}^{m-1} B_i \Rightarrow Y_m = Y_0 \times \prod_{i=0}^{m-1} B_i = Y_0 \times e^{X_0}$$

Additive Normalisierung

Anforderungen an das Verfahren:

- Die Folge der B_i soll einfach zu bestimmen sein.
- Die Werte $\ln B_i$ sollen leicht zu berechnen sein.
- Die Multiplikationen $Y_i \times B_i$ sollen einfach durchzuführen sein.
- Die Anzahl m der Iterationen soll höchstens linear in der Länge l von X_0 wachsen.

Zur Vereinfachung der Multiplikation wird $B_i = 1 + s_i \times 2^{-i}$ mit $s_i \in \{-1, 0, 1\}$ gewählt:
Eine Multiplikation mit B_i besteht nur aus einer Verschiebung und einer Addition/Subtraktion.

$\ln B_i = \ln(1 + s_i \times 2^{-i})$ ist abhängig von s_i positiv, Null oder negativ, was zur Konvergenz von positiven bzw. negativen X_i gegen 0 gebraucht wird.

Die Werte von $\ln(1 \pm 2^{-i})$ müssen aus einer Wertetabelle bestimmt werden, da ihre wiederholte Berechnung zu viel Zeit kosten würde; hierfür wird ein $(2m \times l)$-bit ROM benötigt.

Die Werte sind normalerweise einem Modus *Round-to-nearest* entsprechend gerundet.

Additive Normalisierung: Konvergenzbereich

Zur Erzielung von $X_0 = \sum_{i=0}^{m-1} \ln(1 + s_i \times 2^{-i})$ mit $s_i \in \{-1, 0, 1\}$ muss gelten

$$\sum_{i=1}^{m-1} \ln(1 - 2^{-i}) \leq X_0 \leq \sum_{i=0}^{m-1} \ln(1 + 2^{-i})$$

Für entsprechend großes m genügt $-1{,}24 \leq X_0 \leq 1{,}56$; für jedes X_0 in diesem Intervall gibt es eine Folge $s_0, s_1, \ldots, s_{m-1}$, die Konvergenz von X_i gegen $X_m = 0$ garantiert.

Liegt X nicht im vorgeschriebenen Bereich, transformiert man
$e^X = 2^n \times e^{f \times \ln 2}$ mit $X \times \log_2 e = n + f$, $n \in \mathbb{Z}$, $0 \leq f < 1$ und setzt $X_0 = f \times \ln 2$.
Wegen $0 \leq f < 1$ gilt $0 \leq X_0 < \ln 2 \approx 0{,}7$, also X_0 im zulässigen Bereich.

Allgemeine Potenzen berechnen sich über $Z^X = e^{X \times \ln Z}$.

Additive Normalisierung: Implementierungsdetails

Für positives X_0 genügt $s_i \in \{0, 1\}$:

$$\begin{aligned} D_i &= X_i - \ln(1 + 2^{-i}) \\ (X_{i+1}, Y_{i+1}) &= \begin{cases} (X_i, Y_i) & \text{falls } D_i < 0 \\ (D_i, Y_i + Y_i \times 2^{-i}) & \text{falls } D_i \geq 0 \end{cases} \end{aligned}$$

Wegen $\ln(1 + s_i \times 2^{-i}) \approx s_i \times 2^{-i} - s_i^2 \times 2^{-1-2\times i}$ wird in Schritt i in der Regel das i-te Bit von X_{i+1} zu Null gemacht; daher funktioniert auch die vereinfachte Regel $s_i =$ Bit i von X_i.

Mit $h = l/2$ gilt $\ln(1 \pm 2^{-i}) \approx \pm 2^{-i}$ für $i \geq h$.

Die abschließenden h Iterationen können daher auch simultan durchgeführt werden können:

$$Y_m = Y_{h+1} = Y_h \times (1 + X_h)$$

Berechnung von $e^{0,25}$ mit 10-Bit-Arithmetik (Round-to-nearest)

i	$1+2^{-i}$	$\ln(1+2^{-i})$	$1-2^{-i}$	$\ln(1-2^{-i})$	X_i	Y_i	s_i
0	10,0000000000	0,1011000110	0	—	0,0100000000	1,0000000000	0
1	1,1000000000	0,0110011111	0,1000000000	−0,1011000110	0,0100000000	1,0000000000	0
2	1,0100000000	0,0011100100	0,1100000000	−0,0100100111	0,0100000000	1,0000000000	1
3	1,0010000000	0,0001111001	0,1110000000	−0,0010001001	0,0000011100	1,0100000000	0
4	1,0001000000	0,0000111110	0,1111000000	−0,0001000010	0,0000011100	1,0100000000	0
5	1,0000100000	0,0000100000	0,1111100000	−0,0000100001	0,0000011100	1,0100000000	0
6	1,0000010000	0,0000010000	0,1111110000	−0,0000010000	0,0000011100	1,0100000000	1
7	1,0000001000	0,0000001000	0,1111111000	−0,0000001000	0,0000001100	1,0100010100	1
8	1,0000000100	0,0000000100	0,1111111100	−0,0000000100	0,0000000100	1,0100011110	1
9	1,0000000010	0,0000000010	0,1111111110	−0,0000000010	0,0000000000	1,0100100011	0
10	1,0000000001	0,0000000001	0,1111111111	−0,0000000001	0,0000000000	1,0100100011	0
11					0,0000000000	1,0100100011	

Das Ergebnis $Y_{11} = 1{,}0100100011_2 \approx 1{,}28418$ ist maximal genau bezüglich 10-Bit-Mantisse.

Berechnung von Logarithmen durch multiplikative Normalisierung

Zur Berechnung von Logarithmen in Festkomma-Darstellung wird ebenfalls ein System von Rekurrenzgleichungen mit Koppeltermen B_i benutzt:

$$\begin{aligned} X_{i+1} &= X_i \times B_i \\ Y_{i+1} &= Y_i - \ln B_i \end{aligned}$$

Die B_i werden so gewählt, dass die Folge der X_i gegen 1 konvergiert:

$$\prod_{i=0}^{m-1} B_i = \frac{1}{X_0} \Rightarrow Y_m = Y_0 - \sum_{i=0}^{m-1} \ln B_i = Y_0 - \ln \prod_{i=0}^{m-1} B_i = Y_0 + \ln X_0$$

Multiplikative Normalisierung: Implementierungsdetails

Für $B_i = 1 + s_i \times 2^{-i}$ mit $s_i \in \{-1, 0, 1\}$ ergibt sich

$$\prod_{i=1}^{m-1} (1 - 2^{-i}) \leq \prod_{i=0}^{m-1} (1 + s_i \times 2^{-i}) \leq \prod_{i=0}^{m-1} (1 + 2^{-i})$$

Für genügend großes m reicht $0{,}21 \leq X_0 \leq 3{,}45$ wegen $0{,}29 \leq \prod_{i=0}^{m-1} (1 + s_i \times 2^{-i}) \leq 4{,}77$.

Es kann dieselbe Wertetabelle wie für die Exponentialfunktion benutzt werden.

Wie im Falle der Exponentialfunktion ist ein vereinfachtes Verfahren durch Einschränkung auf $s_i \in \{0, 1\}$ möglich: $s_i = 1 -$ Bit $(i + 1)$ von X_i.

Auch hier kann die Basis e in der Berechnung von Logarithmen problemlos durch eine andere Basis, z. B. 10, ersetzt werden.

Berechnung von ln 0,25 in Arithmetik mit 10-Bit-Mantisse

i	X_i	Y_i	s_i
0	0,0100000000	0,0000000000	1
1	0,1000000000	-0,1011000110	1
2	0,1100000000	-1,0001100101	1
3	0,1111000000	-1,0101001001	0
4	0,1111000000	-1,0101001001	1
5	0,1111111100	-1,0110000111	0
6	0,1111111100	-1,0110000111	0
7	0,1111111100	-1,0110000111	0
8	0,1111111100	-1,0110000111	1
9	1,0000000000	-1,0110001011	0
10	1,0000000000	-1,0110001011	0
11	1,0000000000	-1,0110001011	

Das Ergebnis $Y_{11} = -1{,}0110001011_2 \approx -1{,}38574$ ist in Rundung zur Null maximal genau bezüglich 10-Bit-Mantisse.

CORDIC-Algorithmen

Im CORDIC-Verfahren (*COordinate Rotate DIgital Computer*) werden trigonometrische (und andere) Funktionen durch eine Folge von Koordinatentransformationen im $\mathbb{R}^3$ berechnet.

Das Verfahren beruht auf additiver Normalisierung und besitzt drei Varianten.

In allen wird ein dreidimensionaler Vektor schrittweise solange transformiert, bis eine bestimmte der Komponenten Null geworden ist; die beiden anderen Komponenten (oder eine davon) geben dann den gesuchten Funktionswert an.

Verfahren T_y (Vektormodus):

$$(x_0, y_0, z_0) \xrightarrow{T_y} (x_1, y_1, z_1) \xrightarrow{T_y} \ldots \xrightarrow{T_y} (x_{m-1}, y_{m-1}, z_{m-1}) \xrightarrow{T_y} (x_m, 0, z_m)$$

Verfahren T_z (Rotationsmodus):

$$(x_0, y_0, z_0) \xrightarrow{T_z} (x_1, y_1, z_1) \xrightarrow{T_z} \ldots \xrightarrow{T_z} (x_{m-1}, y_{m-1}, z_{m-1}) \xrightarrow{T_z} (x_m, y_m, 0)$$

Das zirkuläre CORDIC-Verfahren

Das zirkuläre CORDIC-Verfahren arbeitet mit folgenden gekoppelten Rekurrenzgleichungen:

$$\begin{aligned} x_{i+1} &= x_i + y_i \times \delta_i \\ y_{i+1} &= y_i - x_i \times \delta_i \quad \text{mit} \quad \delta_i \in \mathbb{R},\ \alpha_i = \arctan \delta_i \\ z_{i+1} &= z_i + \alpha_i \end{aligned}$$

Die δ_i werden dabei so gewählt, dass y_i bzw. z_i für $i \to \infty$ gegen Null strebt.

Bei exakter Ausführung ergäbe sich

$$\begin{aligned} x_m &= K \times (x_0 \times \cos\alpha + y_0 \times \sin\alpha) \\ y_m &= K \times (y_0 \times \cos\alpha - x_0 \times \sin\alpha) \quad \text{mit} \quad \alpha = \sum_{i=0}^{m-1} \alpha_i,\ K = \prod_{i=0}^{m-1} \sqrt{1+\delta_i^2} \\ z_m &= z_0 + \alpha \end{aligned}$$

Zirkuläres CORDIC-Verfahren

Für die praktische Anwendung müssen die Multiplikationen mit δ_i schnell durchführbar sein.

Für $\delta_i = \pm 2^{-n_i}$ mit $n_i \in \mathbb{N}$ geht dies einfach durch Rechtsverschiebung.

Die Werte von $\alpha_i = \arctan \delta_i = \arctan(\pm 2^{-n_i})$ werden einer Wertetabelle entnommen.

Das Vorzeichen von δ_i wird so gewählt, dass sich y_i bzw. z_i in Richtung auf die Null zu verändert (eventuell aber auch darüber hinausschießt):

$$T_y: \qquad \delta_i = \operatorname{sgn}(x_i) \times \operatorname{sgn}(y_i) \times 2^{-i}$$

$$T_z: \qquad \delta_i = -\operatorname{sgn}(z_i) \times 2^{-i}$$

mit

$$\operatorname{sgn}(u) = \begin{cases} +1 & \text{falls } u \geq 0 \\ -1 & \text{falls } u < 0 \end{cases}$$

Konvergenz des zirkulären CORDIC-Algorithmus im Vektormodus

Die CORDIC-Algorithmen konvergieren nicht für beliebige Startwerte x_0, y_0 und z_0.
Es lässt sich aber zeigen, dass T_y zur Konvergenz führt, wenn gilt:

$$|\lambda_0| \leq |\arctan(2^{1-m})| + \sum_{i=0}^{m-1} |\arctan(2^{-i})|$$

mit

$$\lambda_0 = \begin{cases} \arctan(y_0/x_0) & \text{falls } x_0 \geq 0 \\ \arctan(y_0/x_0) + \pi & \text{falls } x_0 < 0, y_0 \geq 0 \\ \arctan(y_0/x_0) - \pi & \text{falls } x_0 < 0, y_0 < 0 \end{cases}$$

Funktionen des zirkulären CORDIC-Algorithmus im Vektormodus

Im Rahmen der Rechengenauigkeit gilt dann

$$\begin{aligned} x_m &= K \times \sqrt{x_0^2 + y_0^2} \times \operatorname{sgn}(x_0) \\ y_m &= 0 \\ z_m &= z_0 + \arctan(y_0/x_0) \end{aligned}$$

mit

$$K = \prod_{i=0}^{m-1} \sqrt{1 + \delta_i^2} = \prod_{i=0}^{m-1} \sqrt{1 + 2^{-2i}}$$

Für genügend großes m wird $K \approx 1{,}64676$ und es hat $|\lambda_0| \leq 1{,}74$ zu gelten.

$x_0 = 1$ und $z_0 = 0$ ergeben $x_m = K \times \sqrt{1 + y_0^2}$ und $z_m = \arctan y_0$ für beliebiges y_0.

Wegen $\arctan(1/y) = \pi/2 - \arctan y$ genügt eine Implementierung für $|y_0| \leq 1$.

Mit $x_0 = \sqrt{1 - u^2}$, $y_0 = u$, $z_0 = 0$ ergibt sich $z_m = \arcsin u$.

Mit $x_0 = u$, $y_0 = \sqrt{1 - u^2}$, $z_0 = 0$ ergibt sich $z_m = \arccos u$.

Konvergenz des zirkulären CORDIC-Algorithmus im Rotationsmodus

Entsprechend führt T_z zur Konvergenz, wenn

$$|z_0| \leq |\arctan(2^{1-m})| + \sum_{i=0}^{m-1} |\arctan(2^{-i})|$$

und im Rahmen der Rechengenauigkeit gilt mit demselben Wert von K

$$\begin{aligned} x_m &= K \times (x_0 \times \cos z_0 - y_0 \times \sin z_0) \\ y_m &= K \times (y_0 \times \cos z_0 + x_0 \times \sin z_0) \\ z_m &= 0 \end{aligned}$$

$x_0 = 1/K$ und $y_0 = 0$ ergeben $x_m = \cos z_0$ und $y_m = \sin z_0$ für $|z_0| \leq \pi/2$.

In allen anderen Fällen helfen die Regeln $\cos(z \pm 2j\pi) = \cos z$, $\cos(z - \pi) = -\cos z$, $\sin(z \pm 2j\pi) = \sin z$, $\sin(z - \pi) = -\sin z$.

Mit einer ähnlichen Technik lässt sich $z_m = \arccos c$ (bzw. $z_m = \arcsin c$) direkt berechnen, indem man $x_0 = 0$, $y_0 = 1/K$, $z_0 = 0$ setzt und x_i (bzw. y_i) gegen den Wert c treibt.

Hyperbolische CORDIC-Transformationen

Das hyperbolische CORDIC-Verfahren benutzt die modifizierte CORDIC-Rekursion

$$\begin{aligned} x_{i+1} &= x_i - y_i \times \delta_i \\ y_{i+1} &= y_i - x_i \times \delta_i \quad \text{mit} \quad \delta_i \in (-1, 1),\ \alpha_i = \operatorname{artanh} \delta_i \\ z_{i+1} &= z_i + \alpha_i \end{aligned}$$

Bei exakter Rechnung ergibt sich

$$\begin{aligned} x_m &= K' \times [x_0 \times \cos(i \times \alpha) + i \times y_0 \times \sin(i \times \alpha)] \\ y_m &= K' \times [y_0 \times \cos(i \times \alpha) - i \times x_0 \times \sin(i \times \alpha)] \\ z_m &= z_0 + \alpha \end{aligned}$$

mit

$$\alpha = \sum_{i=0}^{m-1} \alpha_i, \quad K' = \prod_{i=0}^{m-1} \sqrt{1 + \delta_i^2}$$

Hyperbolisches CORDIC-Verfahren

Die Folge der δ_i ist hier aber verschieden von der des zirkulären CORDIC-Verfahrens:
Um K' argument-unabhängig zu machen, durchlaufen die n_i in $\delta_i = \pm 2^{-n_i}$ z. B. die Folge

$$(1, 2, 3, 4, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 13, 14, 15, \ldots),$$

d. h. die natürlichen Zahlen, wobei die Zahlen

$$4, 13, 40, 121, \ldots, k, 3 \times k + 1, \ldots$$

doppelt auftreten.

Mit dieser Wahl der δ_i ergibt sich $K' \approx 0{,}82816$.

Vektormodus des hyperbolischen CORDIC-Verfahrens

Treibt man y_i gegen 0, so erhält man für $|y_0| < 0{,}81$

$$\begin{aligned} x_m &= K' \times \sqrt{x_0^2 - y_0^2} \times \operatorname{sgn}(x_0) \\ y_m &= 0 \\ z_m &= z_0 + \operatorname{artanh}(y_0/x_0) \end{aligned}$$

Damit berechnet man:

$z_m = \operatorname{artanh} y_0$ durch $x_0 = 1$, $z_0 = 0$

$x_m = K' \times \sqrt{1 - y_0^2}$ durch $x_0 = 1$

$x_m = K' \times \sqrt{u}$ durch $x_0 = u + \frac{1}{4}$, $y_0 = u - \frac{1}{4}$

$z_m = \frac{1}{2} \times \ln u$ durch $x_0 = u + 1$, $y_0 = u - 1$, $z_0 = 0$

Argumente außerhalb des zulässigen Bereichs transformiert man mit folgender Formel:

$$\operatorname{artanh}(1 - 2^{-e} \times u) = \operatorname{artanh}\left(\frac{2 - u - 2^{-e} \times u}{2 + u - 2^{-e} \times u}\right) + \frac{e \times \ln 2}{2}$$

Rotationsmodus des hyperbolischen CORDIC-Verfahrens

Treibt man z_i gegen 0, so erhält man für $|z_0| < 1{,}13$

$$\begin{aligned} x_m &= K' \times (x_0 \times \cosh z_0 + y_0 \times \sinh z_0) \\ y_m &= K' \times (y_0 \times \cosh z_0 + x_0 \times \sinh z_0) \\ z_m &= 0 \end{aligned}$$

und daraus mit $x_0 = 1/K'$ und $y_0 = 0$

$$\begin{aligned} x_m &= \cosh z_0 \\ y_m &= \sinh z_0 \end{aligned}$$

Für Argumente außerhalb des Konvergenzbereichs können die folgenden Formeln verwendet werden, die für $|u| < \ln 2 \approx 0{,}69$ und ganzzahliges q gelten:

$\cosh(q \times \ln 2 + u) = 2^{q-1} \times [\cosh u + \sinh u + 2^{-2\times q} \times (\cosh u - \sinh u)]$

$\sinh(q \times \ln 2 + u) = 2^{q-1} \times [\cosh u + \sinh u - 2^{-2\times q} \times (\cosh u - \sinh u)]$

Indirekt mit dem CORDIC-Verfahren berechenbare Funktionen

$\tan u = \sin u / \cos u$

$\cot u = \cos u / \sin u$

$\tanh u = \sinh u / \cosh u$

$\coth u = \cosh u / \sinh u$

$e^u = \sinh u + \cosh u$

$u^t = e^{t \times \ln u}$

$\operatorname{arcosh} u = \ln(u + \sqrt{1 - u^2})$

$\operatorname{arsinh} u = \ln(u + \sqrt{1 + u^2})$

Lineare CORDIC-Transformationen

Eine weitere Variante des CORDIC-Verfahrens benutzt

$$\begin{aligned} x_{i+1} &= x_i \\ y_{i+1} &= y_i - x_i \times \delta_i \quad \text{mit} \quad \alpha_i = \delta_i \in \mathbb{R} \\ z_{i+1} &= z_i + \alpha_i \end{aligned}$$

Bei exakter Ausführung ergäbe sich

$$\begin{aligned} x_m &= x_0 \\ y_m &= y_0 - x_0 \times \alpha \quad \text{mit} \quad \alpha = \sum_{i=0}^{m-1} \alpha_i \\ z_m &= z_0 + \alpha \end{aligned}$$

Treibt man y_i gegen 0, erhält man $z_m = z_0 + y_0/x_0$.

Treibt man z_i gegen 0, erhält man $y_m = y_0 + x_0 \times z_0$.

Geschlossene Darstellung der CORDIC-Verfahren

$$
\begin{aligned}
x_{i+1} &= x_i + h \times y_i \times \delta_i \\
y_{i+1} &= y_i - x_i \times \delta_i \\
z_{i+1} &= z_i + \alpha_i
\end{aligned}
$$

mit

$$h \in \{-1, 0, 1\}, \quad \delta_i \in \mathbb{R}$$

und

$$
\alpha_i = \begin{cases} \arctan \delta_i & h = 1 \\ \delta_i & \text{für} \quad h = 0 \\ \operatorname{artanh} \delta_i & h = -1 \end{cases}
$$

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Gepackte Arithmetik

Verarbeitungsbreite von Mikroprozessoren und Datenformate der Praxis

Gleitkomma-Arithmetik hoher Genauigkeit (64 oder 80 Bit) sowie lange Ganzzahlarithmetik (64, künftig auch 128 Bit) sind in modernen General-Purpose-Prozessoren verfügbar.

Viele rechenintensive Anwendungen (Kompression/Dekompression von Daten, Kommunikation, Musik, Bildern und Video; Visualisierung in 2D- oder 3D-Grafik, Animation, virtuelle Realität; Bildverarbeitung; Sprachanalyse und -synthese; Musikwiedergabe; interaktives Video und Videokonferenzen) benötigen aber häufig nur eine geringe Genauigkeit:

- Schwarz-Weiß-Grafik: Pro Pixel 8 Bit für einfache Grauwerte; 12 Bit für medizinische Bilder; 16 Bit als Zwischenformat.
- Farbgrafik: Jeweils 8 Bit für Rot, Grün, Blau, Transparenz im RGB-Pixel.
- Audio-Samples: 8 oder 16 Bit Mono, 2 x 8 oder 2 x 16 Bit Stereo im WAV-Format.
- Video-Samples: 8 oder 16 Bit für MPEG.
- Text: 8 Bit pro Zeichen.
- 16/32 Bit als Zwischenformat höherer Genauigkeit bei Verarbeitung von 8/16 Bit.
- 32 Bit reicht häufig aus für kaufmännische oder einfache numerische Berechnungen.

Einsatz gepackter Arithmetik: Übersicht

- Hardware-Ansätze (Multimedia-Erweiterungen von Standard-Mikroprozessorarchitekturen)
  - Multimedia Acceleration eXtensions (MAX)
    - MAX-1 im HP PA-7100LC (32-Bit PA-RISC architecture 1.1), ab 1994
    - MAX-2 im HP PA-8000 (64-Bit PA-RISC architecture 2.0), ab 1996
  - Matrix Math eXtensions (MMX) in Intel Pentium MMX und Pentium-II, ab 1996
    - erweitert um Streaming SIMD Extensions (SSE) im Intel Pentium-III
    - zusätzlich erweitert um Streaming SIMD Extensions 2 (SSE2) im Intel Pentium 4
  - 3DNow! zusätzlich zu MMX in AMD K6-2 und K6-III, erweitert im AMD Athlon
  - Visual Instruction Set (VIS) im SUN UltraSparc, ab 1995
  - AltiVec im PowerPC
  - MIPS Digital Media eXtensions (MDMX) im MIPS V
- Reine Software-Ansätze
  - Doubly Enhanced Arithmetic (DE-Arithmetik): Zucker, Lee, Flynn 1994
  - Eckard 1995 (4 parallele 8-Bit-Operationen auf der 32-Bit-ALU eines Pentium)
- Anwendung: SP-Intervallarithmetik durch DP-Skalararithmetik (Kolla 1998)

Effizienz gepackter Arithmetik

Mittels gepackter Arithmetik lassen sich in den einschlägigen Anwendungsbereichen hohe Leistungszuwächse erreichen. Einige Beispiele:

- 90-110% für MPEG-1 und H.261 mit MAX-1 im HP PA-7100LC.
- 50-100% beim Intel media benchmark mit MMX.
- 20-60% mit 3DNow! für CD quality audio und high-quality image.
- VIS erreicht etwa den Faktor 5 für separable Konvolution auf großen Bildern.
- Bis zu 10% alleine durch PAVG-Befehl (Mittelwert-Operation) in SSE für HDTV auf DVD.

Gepackte Arithmetik

Idee: Parallele Ausführung mehrerer arithmetischer Operationen geringer Genauigkeit durch Packen der Operanden und Anwendung einer arithmetischen Operation hoher Genauigkeit.

Auch die effiziente Ausführung von 32-Bit-Code auf einem 64-Bit-Prozessor ist denkbar.

Voraussetzungen für die erfolgreiche Ausnutzung von Teilwort-Parallelität sind:

- Verfügbarkeit gepackter Datentypen.
- Implementierung paralleler Teilwort-Operationen.
- Effiziente Durchführung von Pack- und Unpack-Operationen.
- Genügend Parallelität im Maschinenprogramm.
- Geeignet skalierte Operanden und Ergebnisse.

Verfügbarkeit gepackter Datentypen

- Mehrere Zahlen kürzerer Wortlänge werden in einem Maschinenwort größerer Wortlänge untergebracht.
- Die Teilworte haben in Hardware-Techniken meist die gleiche Länge.
- Jede Teilwortlänge ist normalerweise die Maschinenwortlänge dividiert durch eine kleine Zweierpotenz.
- In Software-Techniken kann dagegen u. U. sehr flexibel codiert werden.

- Jedes Teilwort codiert eine Gleitkommazahl oder eine Festkommazahl.
- Jedes als Festkommazahl aufgefasste Teilwort wird i. d. R. entweder als Ganzzahl oder als Nachkommateil gespeichert.
- Die Skalierungsfaktoren der Festkommazahlen werden implizit durch das Programm festgelegt und können dabei für jedes Teilwort individuell gewählt werden.

Hardware-Implementierung paralleler Teilwort-Operationen

- Grundsätzlich soll die vorhandene Standard-Hardware ohne größere Änderungen verwendbar sein.
- Es werden vorhandene Regularitäten in den üblichen Funktionseinheiten genutzt.
- Es ist nicht sinnvoll, alle Operationen als Teilwort-Operationen zu implementieren.
- Gute Kandidaten sind Befehle, die besonders häufig vorkommen, typischerweise in Schleifen.
- Die effiziente Implementierung eines einzigen neuen Befehls kann bestimmte Anwendungen um bis zu einer Größenordnung beschleunigen.

- Parallele Operationen auf den Zahlen kürzerer Wortlänge werden durch eine einzige Operation auf der vollen Maschinenwortlänge durchgeführt (SIMD-Prinzip).
- Alle parallel durchgeführten Operationen sind normalerweise vom selben Typ.
- Eine gegenseitige Störung der Teiloperationen wird durch Unterbrechung der Carry-Chain zuverlässig verhindert (falls sauber implementiert wurde).

Beispiele paralleler Teilwort-Operationen

- Einstellige Operatoren

  Verschiebung: $(x_1 << h| \cdots |x_k << h)$
  Skalierung: $(x_1 \times z| \cdots |x_k \times z)$
  Wurzel: $(\sqrt{x_1}\,| \cdots |\sqrt{x_k})$

- Zweistellige Operatoren

  Addition, Subtraktion, Multiplikation, Division, Minimum, Maximum, Vergleiche:
  $(x_1 \circ y_1| \cdots |x_k \circ y_k)$ mit $\circ = +, -, \times, \ldots$
  Mittelwert (überlauffrei): $((x_1 + y_1)/2| \cdots |(x_k + x_k)/2)$
  Verschiebung und Akkumulierung: $(x_1 << h + y_1| \cdots |x_k << h + y_k)$

- Reduktionsoperationen

  Summe absoluter Differenzen (Pixel-Abstand in der 1-Norm): $\sum_{i=1}^{k} |x_i - y_i|$

  Reduktion mit Addition oder Subtraktion: $(x_1 \pm x_2|y_1 \pm y_2)$
  Skalarprodukt: $(x_1 * y_1 + x_2 * y_2|x_3 * y_3 + x_4 * y_4)$

Effiziente Durchführung von Pack- und Unpack-Operationen

Fallen die eigentlichen (Teilwort-) Operanden als Ergebnis ungepackter Operationen an, werden Konvertierungsoperationen zwischen den alten und den neuen Typen benötigt:

- Pack fügt mehrere Teilworte zu einem Maschinenwort zusammen.
- Unpack extrahiert die Teilworte eines Maschinenworts.

Neben dem reinen Packen und Entpacken können auch Formatumwandlungen nötig werden, z. B. zwischen Gleitkomma- und Festkommatypen oder zwischen gleichartigen Datentypen verschiedener Genauigkeit.

Die Praxis zeigt, dass Packen und Entpacken häufig mit Hilfe kleiner Codesequenzen realisiert werden, weil die Prozessoren und Hardware-Erweiterungen keine entsprechend mächtigen Befehle bereitstellen.

Werden Operanden bereits gepackt aus dem Speicher geholt und dann direkt verarbeitet bzw. nach der Verarbeitung gepackt in den Speicher zurückgeschrieben, reduziert sich auch die Anzahl der Speicherzugriffe.

Überlaufvermeidung und Rundung

Beim Arbeiten mit Ganzzahlen werden Überläufe meist durch einen der folgenden Hardware-Mechanismen automatisch abgefangen:

- Zirkuläre Arithmetik.
- Vorzeichenbehaftete Sättigungsarithmetik.
- Vorzeichenlose Sättigungsarithmetik.

Naive Nutzung dieser Mechanismen kann allerdings große Fehler in Berechnungen eintragen.
Software-Techniken haben Schwierigkeiten, diese Mechanismen überhaupt nachzubilden;
Überläufe müssen dort durch geeignete Wahl der Operanden vermieden werden.

Beim Arbeiten mit Nachkommateilen können sich entweder prinzipiell keine Bereichsüberschreitungen ergeben, oder sie sind eher geringfügig.
Dagegen müssen hier die Ergebnisse oft gerundet werden.

- Im Multimediabereich wurde bisher häufig Rundung durch Abschneiden bevorzugt.
- Mittlerweile ist aber ein Trend zum genauen Runden (Half-adjust-Rundung) festzustellen.

Äußere Voraussetzungen der Anwendung gepackter Arithmetik

- Grundsätzlich muss zunächst genügend Parallelität im Problem aufgespürt werden, ehe diese dann durch Einsatz geeigneter Befehle der gepackten Arithmetik in einen spürbaren Geschwindigkeitsvorteil umgesetzt werden kann.
- Wegen des mit dem Packen und Entpacken verbundenen Aufwands müssen zwischen diesen Hilfsoperationen entsprechend viele eigentliche Rechenoperationen durchgeführt werden.
- Auch bei Anwendung von Hardware-Implementierungen gepackter Arithmetik müssen alle auftretenden Größen a priori in ihrer Genauigkeit und ihrer Magnitude einschätzbar sein, damit Überläufe und Rundungsfehler beherrschbar bleiben.
  Zur sinnvollen Anwendung von Shift-Add-Befehlen in MAX müssen die Konstanten für Skalierungen sogar schon zur Übersetzungszeit bekannt sein.

Übersetzer

Zur Programmierung mit gepackter Arithmetik gibt es verschiedene Möglichkeiten:

- Durch Einbinden von Bibliotheken kann gepackte Arithmetik transparent genutzt werden.
- Handgeschriebene Assemblerroutinen können durch einen erweiterten Assembler genutzt werden (ist aber mühsam und fehleranfällig).
- Durch Inlining und Makros können Assemblerbefehle direkt in den Code einer höheren Programmiersprache eingefügt werden.
- Bei Erweiterung einer höheren Programmiersprache um geeignete Sprachkonstrukte verbirgt der Compiler weitere Details der Programmierung.
  Es gibt z. B. derartige C-Compiler für MAX-2, MMX und VIS.
- Wünschenswert wäre eine Generierung des Codes durch automatische Vektorisierung.
  Erfahrungen (Choe 1998) für die AMD 3DNow!-Technologie zeigen, dass der dabei erzeugte Code mindestens so gut wie per Hand geschrieben ist.
  Weitere Arbeitsgruppen: T. Conte, North Carolina State Univ.; A. Krall, TU Wien.

Multimedia Acceleration eXtensions (MAX)

Die Idee bei MAX-1 war, einen kleinen Satz nützlicher Primitive, aus denen die wichtigsten Anwendungsroutinen effizient synthetisiert werden können, mit geringstmöglichem Aufwand zur Verfügung zu stellen. Beispiel:
Statt gepackter Multiplikations- oder Divisionsbefehle wurde nur ein gepackter Shift-Add-Befehl als Primitiv für deren Realisierung implementiert.
Multiplikation mit einer Konstanten (ganzzahlig oder Nachkommateil) erfolgt dabei durch gestreckte Multiplikation.
Division kann mit Hilfe von Shift-Add-Befehlen und Sättigungsarithmetik realisiert werden.

MAX-2 erweitert MAX-1 im wesentlichen um einige Transportbefehle und verdoppelt über den breiteren Datenpfad der PA-RISC-Architektur 2.0 die Teilwortparallelität.

Minimaler Zusatzaufwand an Chipfläche:
Etwa 0.2% für MAX-1 im HP PA-7100LC, weniger als 0.1% für MAX-2 im HP PA-8000.
Erreichte Beschleunigung mit MAX-1 im HP PA-7100LC: 1.9–2.1 für MPEG-1 und H.261

Kompatibilität von Gleitkomma-Arithmetik und MMX

MMX benutzt die niederwertigen 64 Bit der insgesamt 80 Bit breiten Gleitkommaregister; in der Maschinensprache werden für die MMX-Register allerdings neue Namen vergeben.
Schreibt ein MMX-Befehl in ein MMX-Register, werden Vorzeichen und Exponentenbits auf 1 gesetzt; der Wert erscheint so für die FP-Arithmetik als NAN oder $-\infty$.

Die FP-Register werden als Stack betrieben; beim Betreten und Verlassen von Gleitkommaroutinen sollte der Stack leer sein. Zur Unterstützung des Stack-Prinzips besitzt jedes FP-Register einen Tag, der einen der Werte *empty*, *valid* oder *NAN* enthält.
Die MMX-Register besitzen wahlfreien Zugriff. Beim ersten Zugriff auf ein MMX-Register werden alle Tags aller FP-Register auf *valid* gesetzt; zum Rücksetzen dient der EMMS-Befehl.

Die Tags der FP-Register werden auch zur Unterstützung effizienten Task-Wechsels benutzt.
Bei der Technik des *Lazy Task-Switch* wird die Benutzung der FP-Register durch Setzen eines Bits im Prozessorstatuswort angezeigt; der nächste Task-Wechsel löst eine Unterbrechung aus, deren Aufgabe das Sichern des FP-State ist.
MMX nutzt diese Techniken in transparenter Weise.

Übersicht gepackte Arithmetik

	MAX-1	MAX-2	MMX	SSE	SSE2		3DNow!	3DNow! +	VIS	DE
Datentypen	INT	INT	INT	FP	FP	INT	FP	FP	INT	INT
Anzahl der Teilworte	2	4	2,4,8	4	2	(2),4,8,16	2	2	2,4,8	beliebig
Register	INT-32	INT-64	FP-64	XMM-128	XMM-128		FP-64	FP-64	FP-64	FP-64
Aufteilung	fest	fest	fest	fest	fest		fest	fest	fest	frei
Datenpfad	INT	INT	INT	FP	FP	INT	FP	FP	FP	FP
Zirkuläre Arithmetik	●	●	●			●				
Saturation (signed/unsigned)	●	●	●			●	●/–	●/–		
add/sub	●	●	●	●	●	2,4,8,16	●		2,4	●
compare			●	●	●	●	●		2,4	
average	●	●		4,8 INT		8,16	8 INT	4 INT		
min/max				●,INT	●	●	●	4,8 INT		
sum of absolute differences				8 INT		16		8 INT	8	
shift		●	2,4			2,4,8				●
shift-add	●	●								evtl.
scale									4	●
mult			4	●	●	8	●	4 INT	4	○
div/sqrt				●	●					
reduce(add)/reduce(sub)							●/–	●		
mult-reduce(add)			4/2			8/4				
expand/reduce precision			●/2,4		●/2,4,8		●	4		
pack			●	●	●	●			2,4	2,evtl.
mix		●	●	●	●	●			8	evtl.
permute		●		4 INT		4		●		

Doubly Enhanced Arithmetic (DE-Arithmetik)

DE-Arithmetik realisiert jeweils zwei Festkomma-Operationen durch eine Gleitkomma-Operation.
Es sind dazu keine Änderungen an der gewöhnlichen Gleitkomma-Hardware nötig.

Literatur

Daniel F. Zucker, Ruby B. Lee: Reuse of high precision arithmetic hardware to perform multiple concurrent low precision calculations. Technical Report CSL-TR-94-616, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, April 1994.
Daniel F. Zucker, Ruby B. Lee, Michael J. Flynn: Achieving subword parallelism by software reuse of the floating point data path. In Proc. Multimedia Hardware Architectures, San Jose, CA, February 1997. SPIE, Vol. 3021, pp. 51–64.
Daniel F. Zucker: Architecture and arithmetic for multimedia enhanced processors.
Ph.D. Dissertation, Department of Electrical Engineering, Stanford University, June 1997.

DE-Arithmetik: Voraussetzungen und Resultate

Benötigt werden nur die üblichen Gleitkomma-Operationen (Addition, Subtraktion, Multiplikation), die allerdings schnell sein sollten.
Packen und Entpacken wird ebenfalls auf diese Operationen zurückgeführt.

Nachteil der Methode: Der Anwender muss selbst darauf achten, dass zwischen den Operationen keine Abhängigkeiten durch Überlauf entstehen (Überläufe zwischendurch schaden nicht, wenn das Endergebnis korrekt entpackt werden kann).

Leistungszuwachs durch DE-Arithmetik: Durchschnittlich 14% für MPEG-Decoder.
Die Methode ist prinzipiell erweiterbar auf höheren Parallelitätsgrad; im Falle von MPEG bringt S=4 etwa doppelten Speedup; S=8 bringt fast nichts mehr und passt nicht mehr in FP-64.

DE-Arithmetik: Packen

Packen zweier (nicht notwendig gleich langer) Festkommazahlen X und Y in eine ausreichend genaue normalisierte Gleitkommazahl Z, die in Vorzeichen-Betrag-Darstellung zur Basis 2 abgespeichert wird:

$Z = Y \times 2^c + X$ mit geeigneter ganzzahliger Konstante c

Durchführung z. B. als FP-Multiply-Add oder durch FP-Multiply, gefolgt von FP-Add.

Speicherlayout des Signifikanten von Z (x steht für eine Ziffer von X, y für eine Ziffer von Y):

$yyy \ldots yyy000 \ldots 000xxx \ldots xxx$ bei gleichem Vorzeichen von X und Y

$yyy \ldots yyy111 \ldots 111xxx \ldots xxx$ bei unterschiedlichem Vorzeichen von X und Y

Nullen bzw. Einsen bilden einen Puffer zwischen den Teilworten.

Das Vorzeichen von Z gibt das Vorzeichen von Y wieder.
Die (identischen) Ziffern der Pufferzone in Z geben indirekt das Vorzeichen von X an:
Null: X und Y haben gleiches Vorzeichen; Eins: X und Y haben verschiedenes Vorzeichen.

DE-Arithmetik: Entpacken

Zum Entpacken wird der Signifikant von Z mittels einer Rundung vom Typ $\square_1$ auf eine feste Anzahl h von Stellen gerundet.

$$Y = \mathrm{round}(Z) \times 2^{-c} \text{ und } X = Z - \mathrm{round}(Z)$$

Praktische Durchführung der benötigten Rundung z. B. durch Addition einer geeigneten großen Konstanten (durch Mantissenangleichung werden die Ziffern von X weggerundet) und anschließende Subtraktion derselben Konstanten (Auslöschung durch Subtraktion).

Ausreichend für korrektes Auspacken sind die folgenden Bedingungen:

- Y überlappt sich im Endergebnis (Zwischenergebnisse dürfen davon abweichen) nicht mit dem Vorzeichenbit von X.
- Die Darstellungslänge von Y (d. h. die Anzahl der signifikanten Ziffern) beträgt höchstens h, ansonsten kann X das falsche Vorzeichen erhalten. Beispiel: $y \ldots y|0y \ldots y||1 \ldots 1x \ldots x$
- Die relative Magnitude von Y zu X (d. h. der Abstand des führenden Bits von Y zum führenden Bit von X) ist größer als h. Beispiel: $y \ldots y||1 \ldots 1x \ldots x|0x \ldots x$

Anmerkung: Wächst die relative Magnitude, so sinkt die Genauigkeit von X.

DE-Arithmetik: Parallele arithmetische Operationen

Parallele Skalierung mit einem Faktor k aus dem FP-Zahlenbereich erfolgt durch FP-Multiplikation mit k:

$$Z \times k = (Y \times k) \times 2^c + (X \times k)$$

Skalierung ändert die relative Magnitude um höchstens ± 1, kann aber die Darstellungslänge von Y entscheidend vergrößern; Einfluss darauf hat man durch die Anzahl der Skalierungen und die Darstellungslänge der Faktoren k:
Ist p die Darstellungslänge von k, so wächst die Darstellungslänge von Y um p oder $p - 1$.

Parallele Addition oder Subtraktion erfolgt durch eine FP-Addition bzw. FP-Subtraktion:

$$Z = Y \times 2^c + X \text{ und } W = V \times 2^c + U \Rightarrow Z \pm W = (Y \pm V) \times 2^c + (X \pm U)$$

Additionen oder Subtraktionen können durch Auslöschung die relative Magnitude entscheidend verändern, ebenso die Darstellungslänge von Y!
Abhilfe: Nur Zahlen etwa gleicher Größe addieren oder subtrahieren.

DE-Arithmetik: Schlechte Übertragbarkeit auf ähnliche Situationen

- Das Packen einfach genauer Gleitkommazahlen (FP-32) für doppelt genaue Gleitkomma-Arithmetik (FP-64) ist weniger geeignet:
  - Beispiel: 24 Bit + 1 Bit + 24 Bit in 53 Bit gepackt $\Rightarrow$ nur 4 Bit Puffer.
  - Während bei Festkommadarstellungen die relative Magnitude grundsätzlich eng beschränkt ist, sind bei Gleitkommazahlen große Variationsbreiten zu erwarten.
- Das Packen kleiner Festkommazahlen (INT-16) für lange Festkomma-Arithmetik (INT-64) ist weniger geeignet:
  - Kein automatisches Skalieren.
  - Manche Prozessoren mit FP-64 verfügen nicht über INT-64.

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Asynchrone Systeme

Asynchrone Systeme

Asynchrone digitale Schaltungen (*self-timed systems*) kommen ohne Taktsignale aus.
Vorteile gegenüber entsprechenden synchronen Systemen sind:

- Höherer Durchsatz:
  - Ausnutzung der von den Daten abhängigen Latenzen.
  - Unmittelbar geeignet für Pipelining, auch ohne Zergliederung in Stufen.
  - Keine Beschränkungen in der Geschwindigkeit durch Verzerrung von Taktsignalen.
- Große Systeme mit hoher Datenrate sind elektrisch leichter zu realisieren:
  - Verteilte (lokalisierte) und damit einfach zu implementierende Steuerung.
  - Kein Netzwerk mit langen Leitungen zur Verteilung von Taktsignalen nötig.
- Geringerer Energieverbrauch:
  - Keine Taktgenerierung nötig.
  - Kleinere Treiber.
  - Anzahl der Pegelwechsel wird minimiert.

Handshakeverfahren

Die Subsysteme eines asynchronen Systems benötigen Mechanismen, sich den Beginn (*start*) bzw. das Ende (*completion*) eines Verarbeitungsvorgangs mitzuteilen.

Derartige zeitliche Information wird in die Daten eingebettet oder läuft mit diesen mit.

Die dazu verwendeten Protokolle besitzen die Form eines *Handshakeverfahrens*:

- Sobald die Eingangssignale stabil an einer Verarbeitungseinheit anliegen, wird diese durch ein Signal (*request*, *data ready*) von der Bereitstellungseinheit davon informiert und kann dann mit der Arbeit beginnen.
- Die Eingangssignale werden stabil gehalten, bis die Verarbeitungseinheit durch ein anderes Signal (*acknowledge*) der Bereitstellungseinheit mitteilt, dass die Eingangssignale nun geändert werden dürfen (die Daten sind abgenommen oder werden aus anderen Gründen nicht mehr benötigt).

Bereitstellungs- und Quittungssignale

Einem Bündel von Datenleitungen (Bus) können zwei Steuerleitungen hinzugefügt werden, auf denen die Bereitstellungs- und Quittungssignale transportiert werden (*bundled data protocol*).

Jedes der benötigten Signale wird durch eine der folgenden Methoden auf einer der Steuerleitungen codiert:

- Pegelgesteuertes Signal (*level-sensitive signaling*): Erreichen eines bestimmten Signalpegels, *low* oder (häufiger) *high*.
- Flankengesteuertes Signal (*edge-triggered signaling*): Ansteigender Signalverlauf; alternativ abfallender Signalverlauf.
- Übergangsgesteuertes Signal (*transition signaling*): Änderung des Signalverlaufs, d. h. sowohl ansteigender als auch abfallender Signalverlauf codieren das Signal.

Doppelsignalbetrieb

Doppelsignalbetrieb (*dual-rail data encoding*) bettet das Bereitstellungssignal direkt in jedes einzelne Datenbit ein:

- Jedes Datenbit wird durch die Signale auf zwei Leitungen realisiert; eine davon steht für den Wert 0, die andere für den Wert 1.
- Die Bereitstellung eines Werts wird durch ein Signal auf der entsprechenden Leitung angezeigt; der Wert selbst ist damit implizit.
- Die Codierung des Signals kann nach jeder der drei oben genannten Methoden erfolgen.

Mit Hilfe des Doppelsignalbetriebs können alle booleschen Gleichungen durch negationsfreie Schaltnetze bzw. -werke aus Und- und Oder-Gattern realisiert werden.

Die Und- und die Oder-Operation sind monoton in ihren Argumenten; dies gilt damit auch für alle Schaltnetze, die ausschließlich aus Und- bzw. Oder-Gattern aufgebaut sind.

Da die Steuersignale direkt in jedes Datenbit integriert sind, kann feingranular und bezüglich der Korrektheit der Schaltung vollständig unabhängig von den Latenzen der Gatter und Verbindungen (*delay-insensitive*, DI) gearbeitet werden.

Dieser Vorteil muss mit einer verdoppelten Anzahl von Leitungen bezahlt werden.

Carry-Completion-Sensing-Addierer (CCSA)

Durch Berechnung der gesamten Übertragsinformation im Doppelsignalbetrieb entsteht aus dem Ripple-Carry-Addierer (RCA) der *Carry-Completion-Sensing-Addierer*.

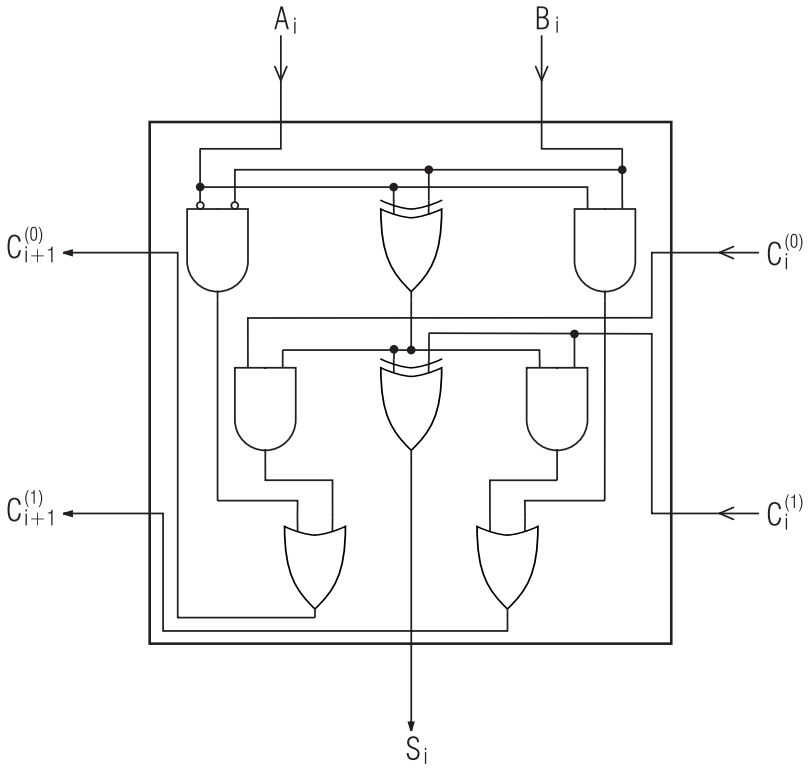
Er benutzt statt Volladdierern eine Modifikation (*Carry-Complete-Volladdierer*), in der das Übertragsbit durch ein Paar von Signalen ersetzt ist, mit den Zuständen $(C_i^{(0)}, C_i^{(1)})$ aus

$(0,1)$	es liegt definitiv Übertrag vor
$(1,0)$	es liegt definitiv kein Übertrag vor
$(0,0)$	die Übertragsinformation ist noch nicht bekannt
$(1,1)$	(tritt im regulären Betrieb nicht auf, Fehler)

$C_{i+1}^{(0)} + C_{i+1}^{(1)} = 1$ zeigt an, dass die Übertragsberechnung abgeschlossen ist.

Der endgültige Zustand von $C_i^{(1)}$ beschreibt das ursprüngliche Übertragsbit C_i.

Carry-Complete-Volladdierer (CVA)



$$
\begin{aligned}
C_{i+1}^{(1)} &= A_i B_i + (\overline{A_i} B_i + A_i \overline{B_i})\, C_i^{(1)} \\
&= A_i B_i + (A_i \oplus B_i)\, C_i^{(1)} \\
(&= A_i B_i + (A_i + B_i)\, C_i^{(1)} \quad)
\end{aligned}
$$

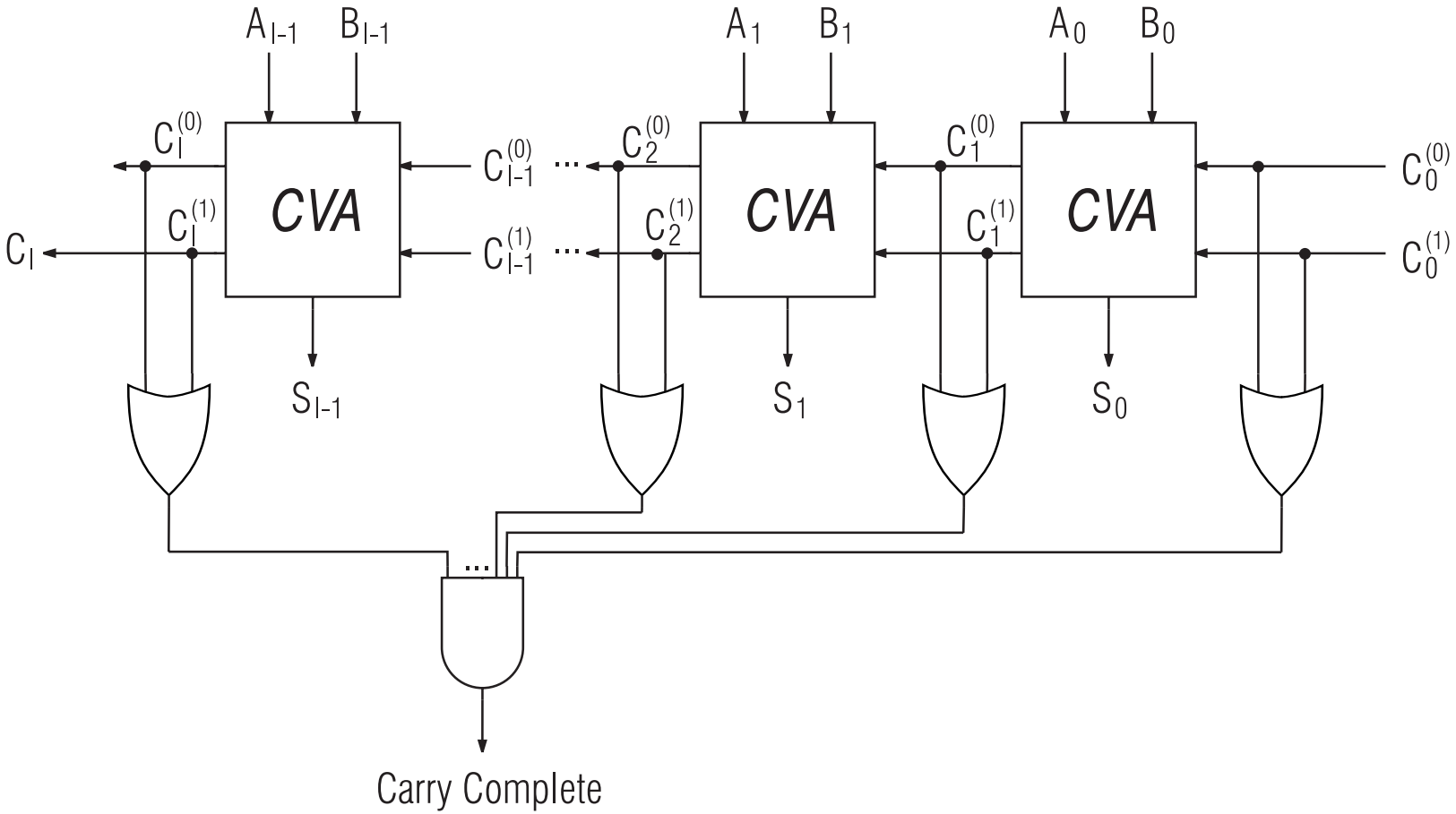
$$
\begin{aligned}
C_{i+1}^{(0)} &= \overline{A_i}\ \overline{B_i} + (\overline{A_i} B_i + A_i \overline{B_i})\, C_i^{(0)} \\
&= \overline{A_i}\ \overline{B_i} + (A_i \oplus B_i)\, C_i^{(0)} \\
(&= \overline{A_i}\ \overline{B_i} + (\overline{A_i} + \overline{B_i})\, C_i^{(0)} \quad)
\end{aligned}
$$

$$
S_i = (A_i \oplus B_i) \oplus C_i^{(1)}
$$

Die Stabilisierung des Summenbits S_i kann am Eintreten der Bedingung $C_i^{(0)} + C_i^{(1)} = 1$ (NB: nicht $C_{i+1}^{(0)} + C_{i+1}^{(1)} = 1$) erkannt werden, da diese sich im Folgenden nicht mehr ändert.

Eine bei der Berechnung von S_i aus $A_i, B_i, C_i^{(1)}$ zwangsläufig auftretende Verzögerung muss allerdings abgeschätzt und berücksichtigt werden.

Carry-Completion-Sensing-Addierer



Ein Carry-Completion-Sensing-Addierer wird als Einheit asynchron betrieben, d. h. die Stabilisierung aller S_i sowie von C_l wird abgewartet, bevor eines der Bits benutzt wird.

Formale Bedingung: $C = \bigwedge_{i=0}^{l} (C_i^{(0)} + C_i^{(1)}) = 1$

Latenz des Carry-Completion-Sensing-Addierer

Die Latenz eines Carry-Completion-Sensing-Addierers für eine bestimmte Eingabe ist im wesentlichen linear im Maximum aus der Länge der längsten 1-Carry-Chain (entspricht der Carry-Chain des Ripple-Carry-Addierers) und der Länge der längsten 0-Carry-Chain.

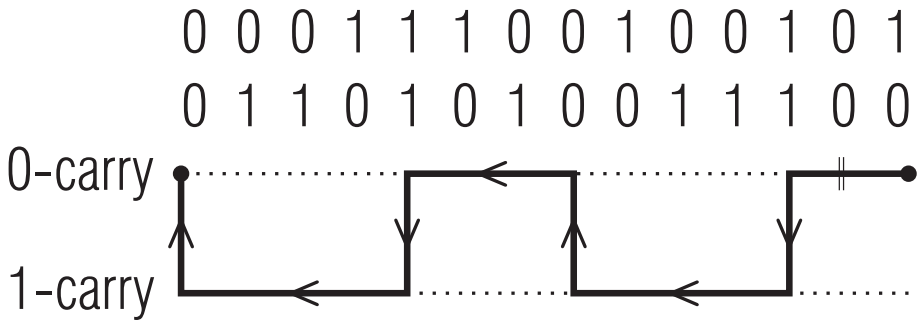
Es ergibt sich dadurch eine durchschnittliche Latenz von nur $\Theta(\log l)$.

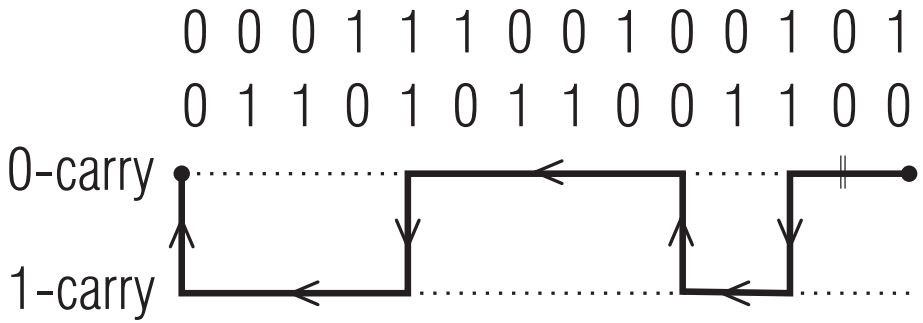
G_i bedeutet das Ende einer Carry-Chain, Beginn einer 1-Carry-Chain.

K_i bedeutet das Ende einer Carry-Chain, Beginn einer 0-Carry-Chain.

P_i bedeutet eine Verlängerung der jeweiligen Carry-Chain.

Für $C_0 = 0$ muss fiktiv K_{-1} vorausgesetzt werden, und G_{-1} für $C_0 = 1$.





Startsignal für den Carry-Completion-Sensing-Addierer

Um sicherzustellen, dass die Verarbeitung erst beginnt, wenn alle Daten (bis auf den Eingangsübertrag, der ja bereits im Doppelsignalbetrieb geschaltet ist) stabil anliegen, können die Übertragsausgänge der Carry-Complete-Volladdierer bis zum Eintreffen eines Startsignals E auf Null gehalten werden:

$$\begin{aligned} C_{i+1}^{(1)} &= E\,(A_i B_i + (A_i + B_i)\,C_i^{(1)}) \\ C_{i+1}^{(0)} &= E\,(\overline{A_i}\;\overline{B_i} + (\overline{A_i} + \overline{B_i})\,C_i^{(0)}) \end{aligned}$$

Auch das globale Fertigstellungssignal C muss zunächst durch E abgeschaltet werden, damit die nachfolgenden Verarbeitungseinheiten nicht versehentlich zu früh aktiviert werden:

$$C = E \;\wedge\; \bigwedge_{i=0}^{l} (C_i^{(0)} + C_i^{(1)})$$

Verzögerungsunabhängiger Ripple-Carry-Addierer (DIRCA)

Für den zuverlässigen Betrieb des CCSA wird die sogenannte Bündelungsbedingung (*bundling constraint*, BC) vorausgesetzt: Das Startsignal darf erst aktiviert werden, wenn alle Eingänge, die nicht im Doppelsignalbetrieb vorliegen, stabil geworden sind; das Fertigstellungssignal darf erst aktiviert werden, wenn alle Ausgänge, die nicht im Doppelsignalbetrieb vorliegen, stabil geworden sind.

Streuen die Latenzen der Gatter und Verbindungen zu stark bzw. liegen für ihre Werte keine gesicherten Abschätzungen vor, kann die Bündelungsbedingung nicht bzw. nicht effizient garantiert werden.

Ein verzögerungsunabhängiger Ripple-Carry-Addierer ergibt sich, wenn *alle* Berechnungen im Doppelsignalbetrieb durchgeführt werden.

Auf ein Startsignal bzw. ein Fertigstellungssignal kann dann sogar völlig verzichtet werden.

Die Summenbits S_i und der Ausgangsübertrag C_l können asynchron und einzeln sofort nach ihrer Stabilisierung verwendet werden.

Arbeitsweise des DIRCA

Die Arbeitsweise des DIRCA wird durch folgende Gleichungen beschrieben:

$$
\begin{aligned}
C_{i+1}^{(1)} &= A_i^{(1)} B_i^{(1)} + (A_i^{(1)} + B_i^{(1)})\, C_i^{(1)} \\
C_{i+1}^{(0)} &= A_i^{(0)} B_i^{(0)} + (A_i^{(0)} + B_i^{(0)})\, C_i^{(0)} \\
S_i^{(1)} &= A_i^{(1)} B_i^{(1)} C_i^{(1)} + A_i^{(1)} B_i^{(0)} C_i^{(0)} + A_i^{(0)} B_i^{(1)} C_i^{(0)} + A_i^{(0)} B_i^{(0)} C_i^{(1)} \\
S_i^{(0)} &= A_i^{(0)} B_i^{(0)} C_i^{(0)} + A_i^{(0)} B_i^{(1)} C_i^{(1)} + A_i^{(1)} B_i^{(0)} C_i^{(1)} + A_i^{(1)} B_i^{(1)} C_i^{(0)}
\end{aligned}
$$

Wird ein Fertigstellungssignal gewünscht, berechnet sich dieses als

$$
C = (C_l^{(0)} + C_l^{(1)}) \wedge \bigwedge_{i=0}^{l-1} (S_i^{(0)} + S_i^{(1)})
$$

Vor Beginn einer neuen Berechnung werden alle Ein- und Ausgänge auf Null zurückgesetzt.

Vor oder hinter dem DIRCA platzierte Register müssen ebenfalls für asynchronen Doppelsignalbetrieb ausgelegt sein.

Verzögerungsunabhängiger Carry-Lookahead-Addierer (DICLA)

Auch ein Carry-Lookahead-Addierer (CLA) kann verzögerungsunabhängig aufgebaut werden, indem alle Ein- und Ausgänge im Doppelsignalbetrieb geschaltet und interne Signale dem Doppelsignalbetrieb angepasst werden (*one-hot coding*).

Ein baumartiger Aufbau ergibt sich durch folgende Beziehungen:

$$
\begin{aligned}
K_{i,i} &= A_i^{(0)} B_i^{(0)} \\
G_{i,i} &= A_i^{(1)} B_i^{(1)} \\
P_{i,i} &= A_i^{(0)} B_i^{(1)} + A_i^{(1)} B_i^{(0)} \\
S_i^{(1)} &= A_i^{(1)} B_i^{(1)} C_i^{(1)} + A_i^{(1)} B_i^{(0)} C_i^{(0)} + A_i^{(0)} B_i^{(1)} C_i^{(0)} + A_i^{(0)} B_i^{(0)} C_i^{(1)} \\
S_i^{(0)} &= A_i^{(0)} B_i^{(0)} C_i^{(0)} + A_i^{(0)} B_i^{(1)} C_i^{(1)} + A_i^{(1)} B_i^{(0)} C_i^{(1)} + A_i^{(1)} B_i^{(1)} C_i^{(0)} \\
\\
P_{i,k} &= P_{i,j}\, P_{j-1,k} \\
K_{i,k} &= K_{i,j} + P_{i,j}\, K_{j-1,k} \\
G_{i,k} &= G_{i,j} + P_{i,j}\, G_{j-1,k} \\
C_j^{(1)} &= G_{j-1,k} + P_{j-1,k}\, C_k^{(1)} \\
C_j^{(0)} &= K_{j-1,k} + P_{j-1,k}\, C_k^{(0)}
\end{aligned}
$$

Beschleunigter verzögerungsunabhängiger Carry-Lookahead-Addierer

Simulationen zeigen, dass der erhoffte Erfolg ausbleibt: Der DICLA ist sogar etwas langsamer als der entsprechende DIRCA!

Eine Verbesserung ergibt sich erst, wenn in die Struktur des DICLA noch etwas mehr Lookahead-Funktionalität eingebaut wird.

Der resultierende verzögerungsunabhängige Carry-Lookahead-Addierer mit Beschleunigungslogik (DICLASP) besitzt eine durchschnittliche Latenz von $\Theta(\log \log l)$ und ist der Addierer mit dem derzeit besten durchschnittlichen Aufwands-Zeit-Produkt $\Theta(l \times \log \log l)$.

DIRCA und DICLASP sind mit einem gegenüber einem RCA bzw. einem CLA nur wenig erhöhten Aufwand in CMOS realisierbar.

Literatur

F.-C. Cheng, S. H. Unger, M. Theobald: Self-timed carry-lookahead adders.
IEEE Transactions on Computers, Vol. 49, No. 7 (July 2000) 659–672.

Durchschnittliche Leistung einiger wichtiger Addierer

Typ des Addierers	Implementierung	Aufwand	Transistoren	Latenz	$A \times T$
RCA	synchron	$O(l)$	$40 \times l$	$O(l)$	$O(l^2)$
CLA	synchron	$O(l)$	$48 \times l - 22$	$O(\log l)$	$O(l \times \log l)$
CCSA	asynchron (BC)	$O(l)$		$O(\log l)$	$O(l \times \log l)$
DIRCA	asynchron (DI)	$O(l)$	$42 \times l$	$O(\log l)$	$O(l \times \log l)$
DICLA	asynchron (DI)	$O(l)$		$O(\log l)$	$O(l \times \log l)$
DICLASP	asynchron (DI)	$O(l)$	$66 \times l - 4$	$O(\log \log l)$	$O(l \times \log \log l)$

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Thema: Komparatoren

Komparatoren für $\mathrm{UInt}_2(l)$

Erzeugung aller Vergleichsprädikate aus $\leq$ hardwaremäßig umständlich und zeitaufwendig.

Kompromiss: Es werden z. B. die Prädikate $<$ und $=$ direkt und simultan berechnet.

Definition der Prädikate $<$ und $=$

$$(A_{l-1}, \ldots, A_0) < (B_{l-1}, \ldots, B_0) \;\Leftrightarrow\; \exists j\colon A_j < B_j \wedge \forall i > j\colon A_i = B_i$$
$$(A_{l-1}, \ldots, A_0) = (B_{l-1}, \ldots, B_0) \;\Leftrightarrow\; \forall i\colon A_i = B_i$$

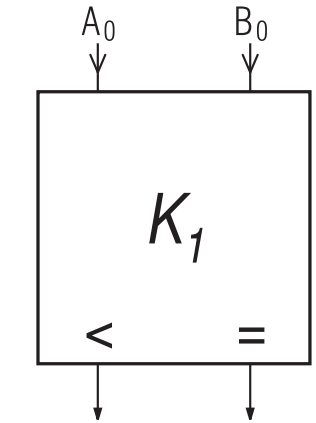
Grundlegende Rekursion für das Prädikat $<$

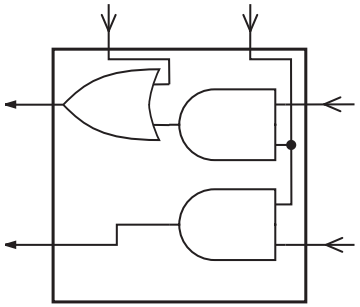
$$\forall k, 1 \leq k \leq l-1\colon \quad (A_{l-1}, \ldots, A_0) < (B_{l-1}, \ldots, B_0) \quad \Leftrightarrow$$
$$(A_{l-1}, \ldots, A_k) < (B_{l-1}, \ldots, B_k) \;\vee$$
$$(A_{l-1}, \ldots, A_k) = (B_{l-1}, \ldots, B_k) \;\wedge\; (A_{k-1}, \ldots, A_0) < (B_{k-1}, \ldots, B_0)$$

Simultane Rekursion für das Prädikat $=$

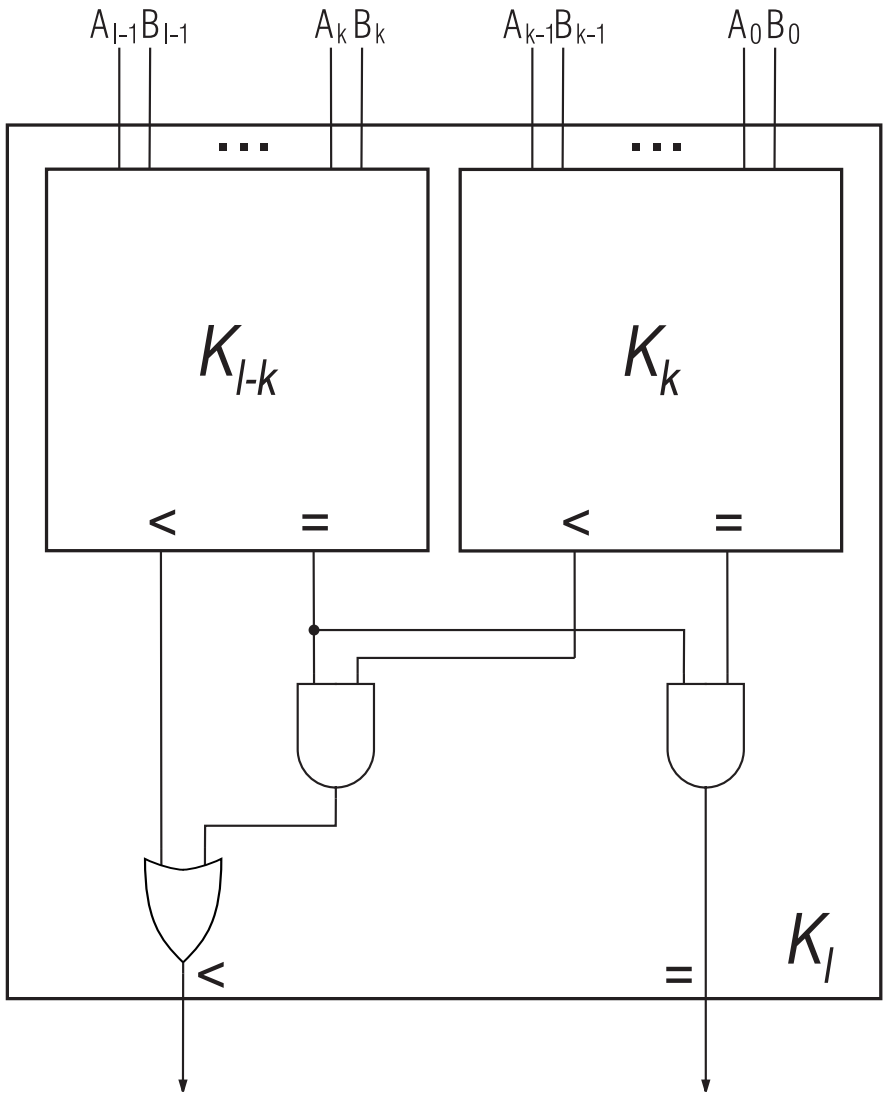
$$\forall k, 1 \leq k \leq l-1\colon \quad (A_{l-1}, \ldots, A_0) = (B_{l-1}, \ldots, B_0) \quad \Leftrightarrow$$
$$(A_{l-1}, \ldots, A_k) = (B_{l-1}, \ldots, B_k) \;\wedge\; (A_{k-1}, \ldots, A_0) = (B_{k-1}, \ldots, B_0)$$

Ein rekursiv aufgebauter Komparator für $\mathrm{UInt}_2(l)$

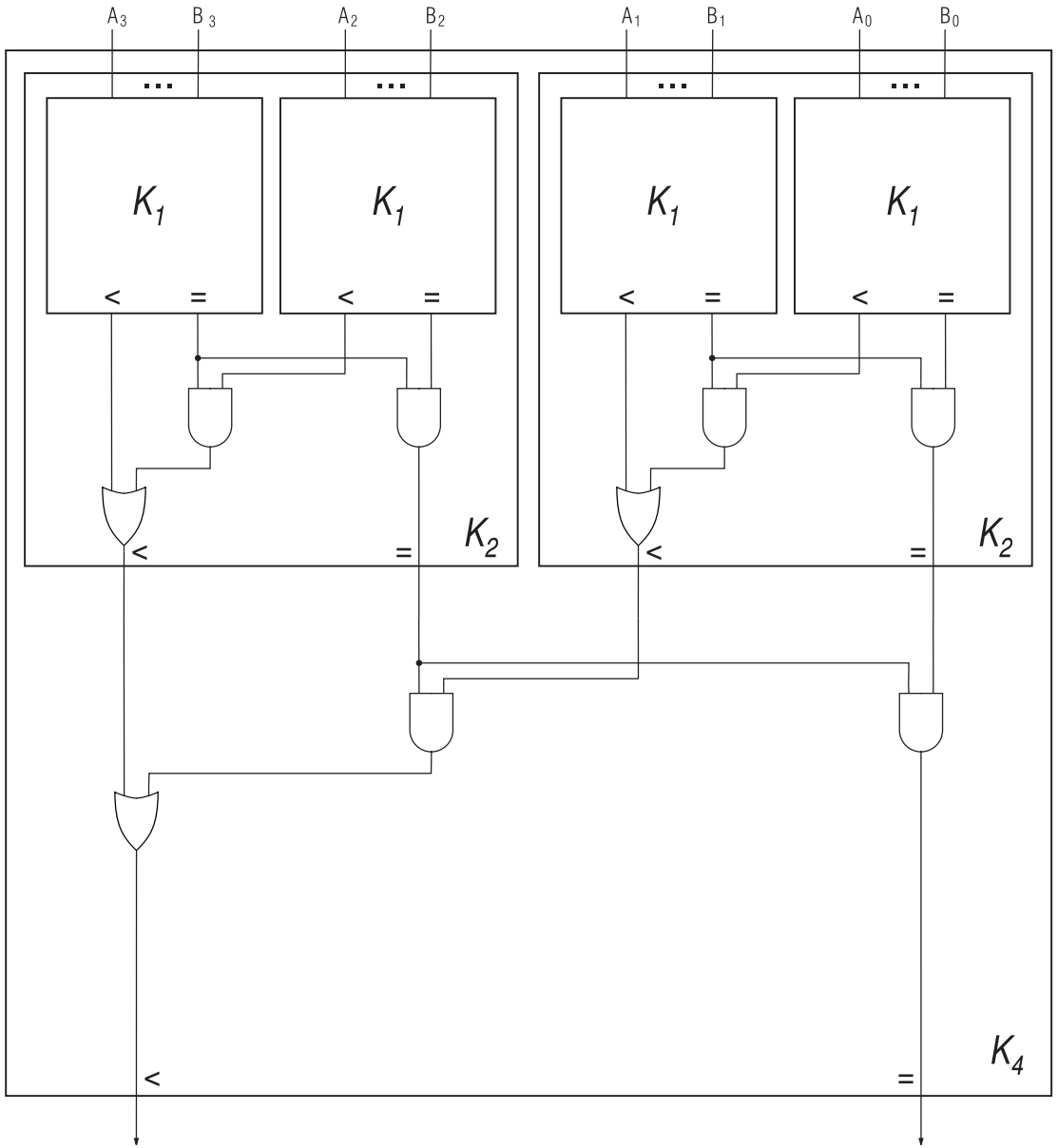




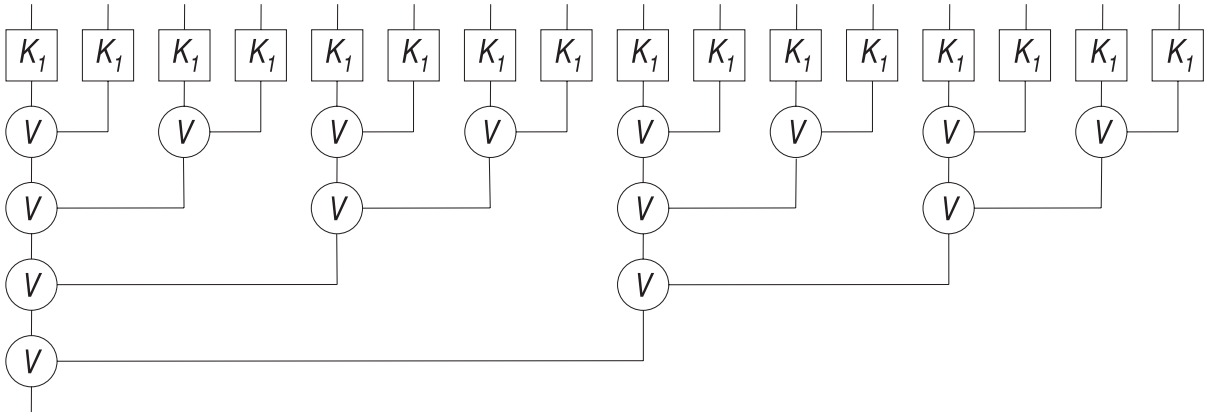
Verknüpfungsmodul V



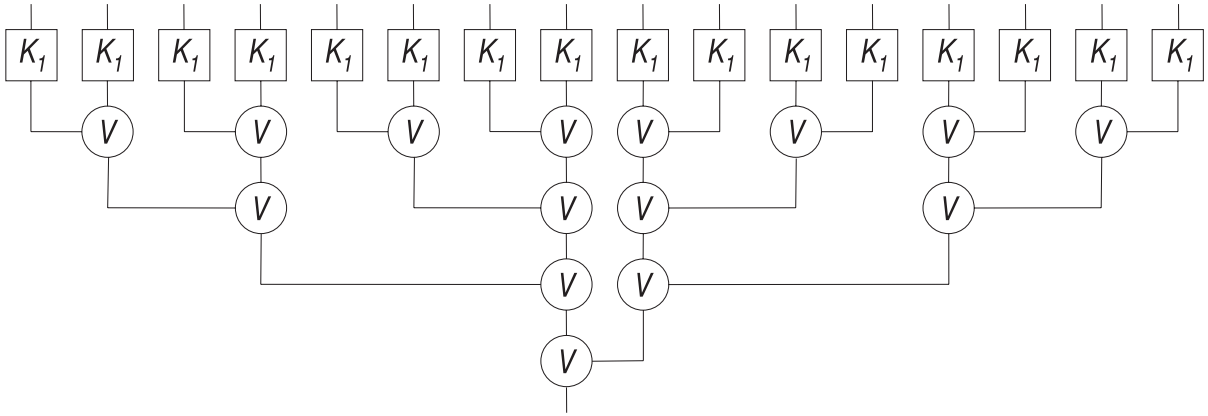
Beispielabwicklung der Hardware-Rekursion



Layout und Leitungslängen baumartiger Komparatoren ($k \approx l/2$)

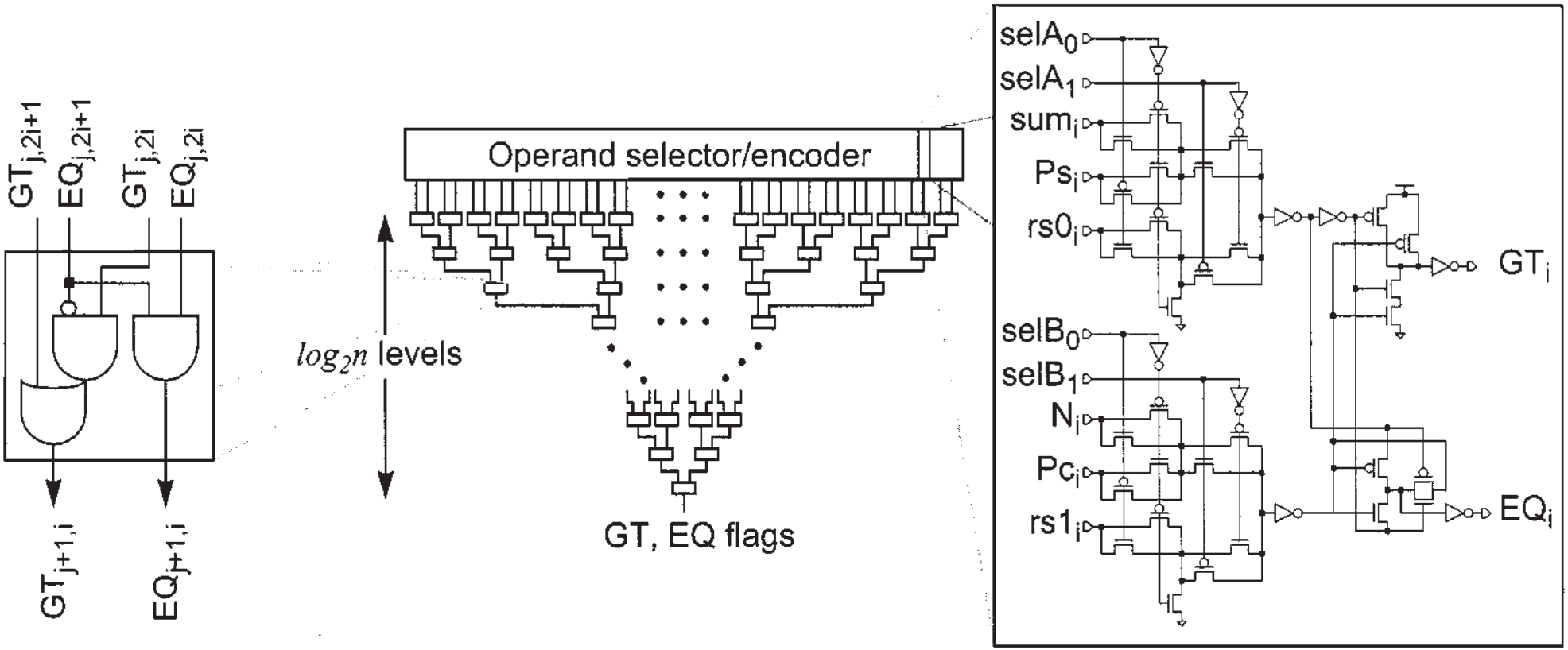


Geschicktes Layout reduziert die Leitungslänge um fast die Hälfte:



Komparator (Anwendungsbeispiel)

Komparator im Domain-Specific Reconfigurable Cryptographic Processor:



(aus J. Goodman, A. P. Chandrakasan: An energy-efficient reconfigurable public-key cryptography processor. IEEE Journal of Solid-State Circuits, Vol. 36 (2001), No. 11, pp. 1808–1820)

NB: Das abgebildete Verknüpfungsmodul (nebst Formel in der Arbeit) ist fehlerhaft!

Baumartige Komparatoren: Aufwandsbetrachtung für $l = 2^m$

$$a_{2k} = 2 \times a_k + 2 \times a_{and} + a_{or} \quad \Rightarrow \quad \boxed{a_{2^m} = 2^m \times a_1 + (2^m - 1) \times (2 \times a_{and} + a_{or})}$$

$$t^{=}_{2k} = t^{=}_{k} + t_{and} \quad \Rightarrow \quad \boxed{t^{=}_{2^m} = t^{=}_{1} + m \times t_{and}}$$

$$t^{<}_{2k} = \max\{t^{=}_{k}, t^{<}_{k}\} + t_{and} + t_{or} \quad \Rightarrow \quad \boxed{t^{<}_{2^m} = \max\{t^{=}_{1}, t^{<}_{1}\} + m \times (t_{and} + t_{or})}$$

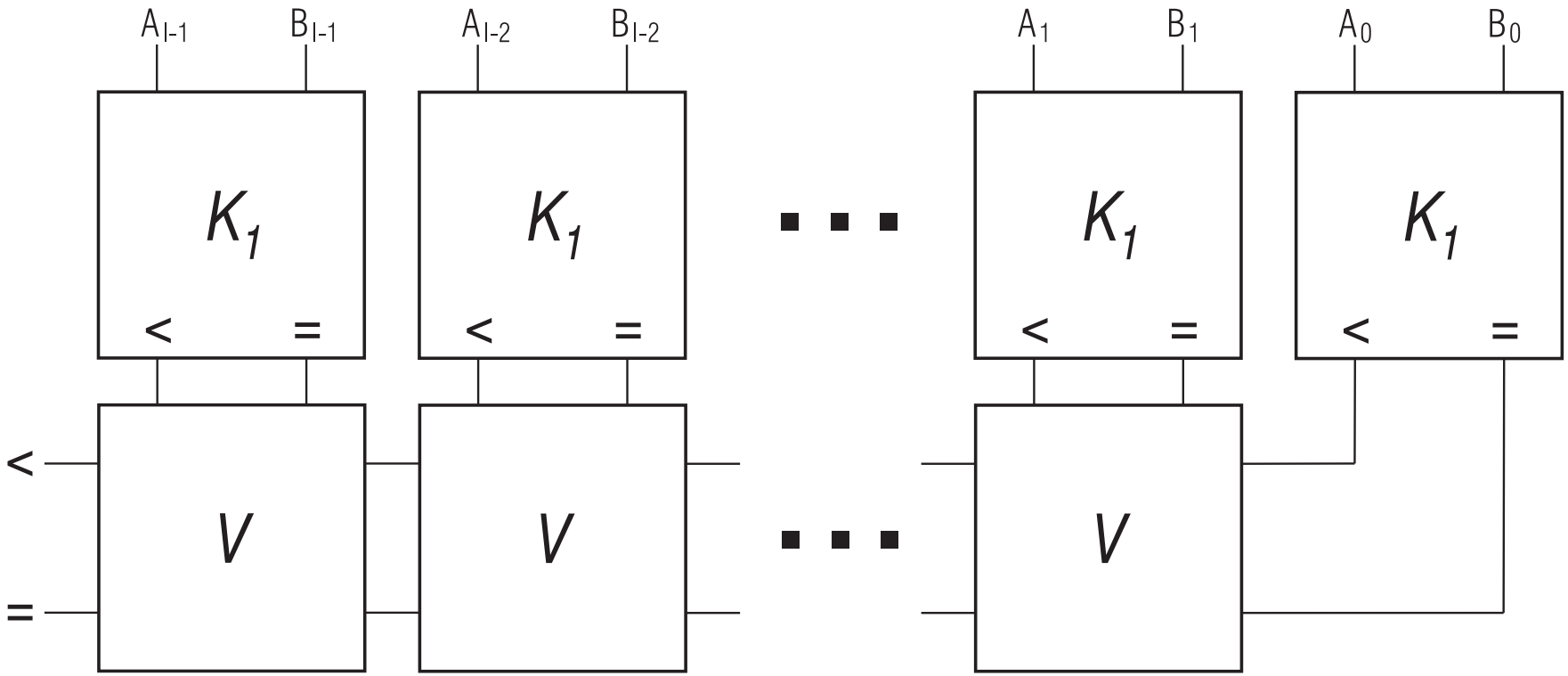
Berücksichtigung von Layout und Leitungslängen führt auf

$$a_{2^m} \approx 2^m \times (a_1 + m \times (2 \times a_{and} + a_{or}))$$

$$t^{=}_{2^m} \approx t^{=}_{1} + m \times t_{and} + m \times \tau + 2^m \times \tau'$$

$$t^{<}_{2^m} \approx \max\{t^{=}_{1}, t^{<}_{1}\} + m \times (t_{and} + t_{or}) + m \times \tau + 2^m \times \tau'$$

Linear kaskadierter Komparator ($k = l - 1$)



Für $k = 1$ ergibt sich ein linear kaskadierter Komparator mit Propagierung nach rechts.

Aufwandsbetrachtung für lineare Komparatoren

$a_l = l \times a_1 + (l-1) \times (2 \times a_{and} + a_{or})$

Der Flächenbedarf entspricht genau dem eines (idealisierten) baumartigen Komparators!

Erklärung: Die Bauteile sind nur anders organisiert. Beispielsweise sind äquivalent:

$((A_3 = B_3) \wedge (A_2 = B_2)) \wedge ((A_1 = B_1) \wedge (A_0 = B_0))$ (baumartiges Schema)
$(A_3 = B_3) \wedge ((A_2 = B_2) \wedge ((A_1 = B_1) \wedge (A_0 = B_0)))$ (lineares Schema)

$t_l^= = t_1^= + (l-1) \times t_{and}$

$t_l^< = \max\{t_1^<, t_1^= + t_{and}\} + (l-2) \times t_{and} + (l-1) \times t_{or}$

Berücksichtigung von Layout und Leitungslängen führt auf

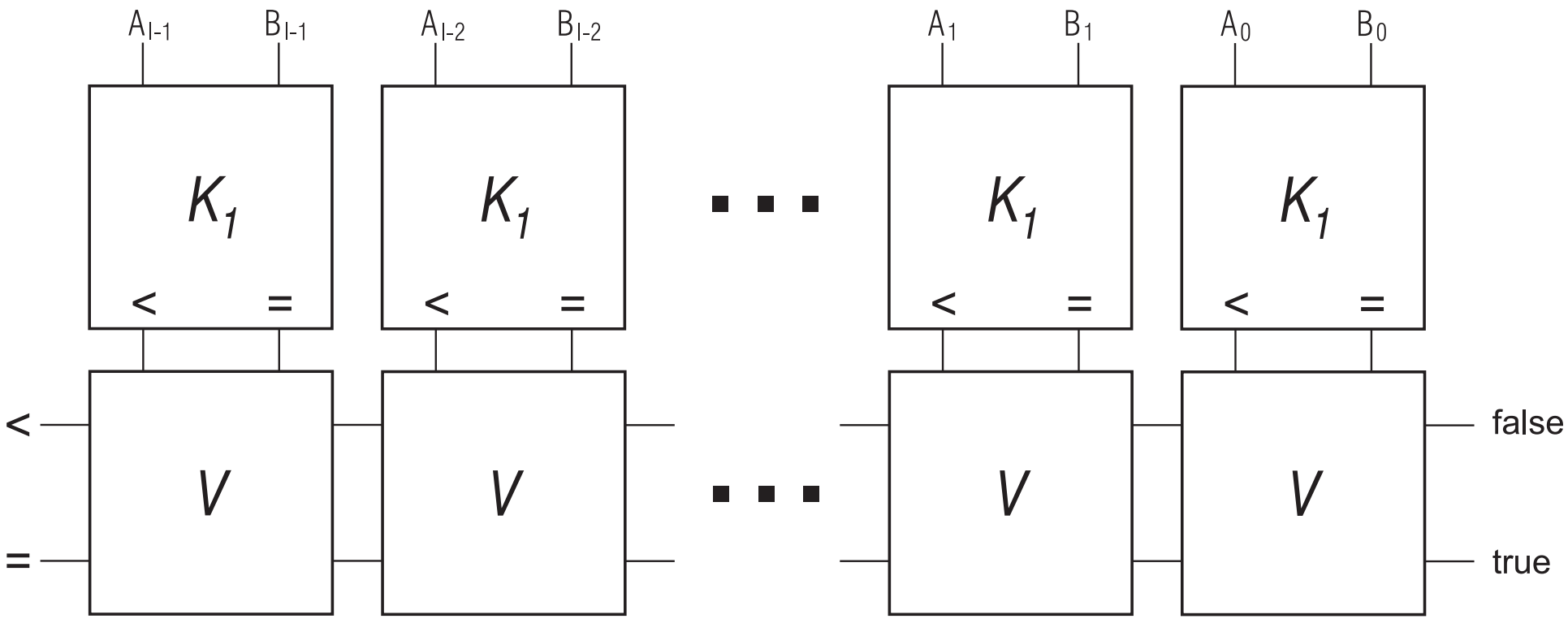
$a_l \approx l \times (a_1 + 2 \times a_{and} + a_{or})$

$t_l^= \approx t_1^= + (l-1) \times t_{and} + l \times \tau'$

$t_l^< \approx \max\{t_1^<, t_1^= + t_{and}\} + (l-2) \times t_{and} + (l-1) \times t_{or} + l \times \tau'$

Linear kaskadierter Komparator aus identischen Komponenten

Aufbau aus identischen Komponenten durch Ergänzung eines Verknüpfungsmoduls:

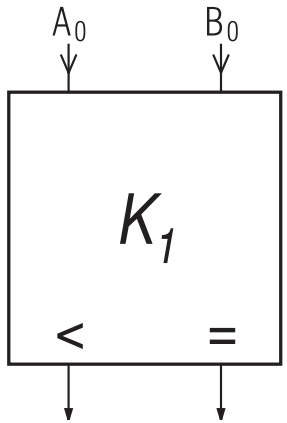


Nachteil: Der kritische Pfad wird (geringfügig) verlängert.

Primitive Komparatorzelle

Der Aufbau der primitiven Komparatorzelle K_1 hängt von der Codierung der Ziffern ab.

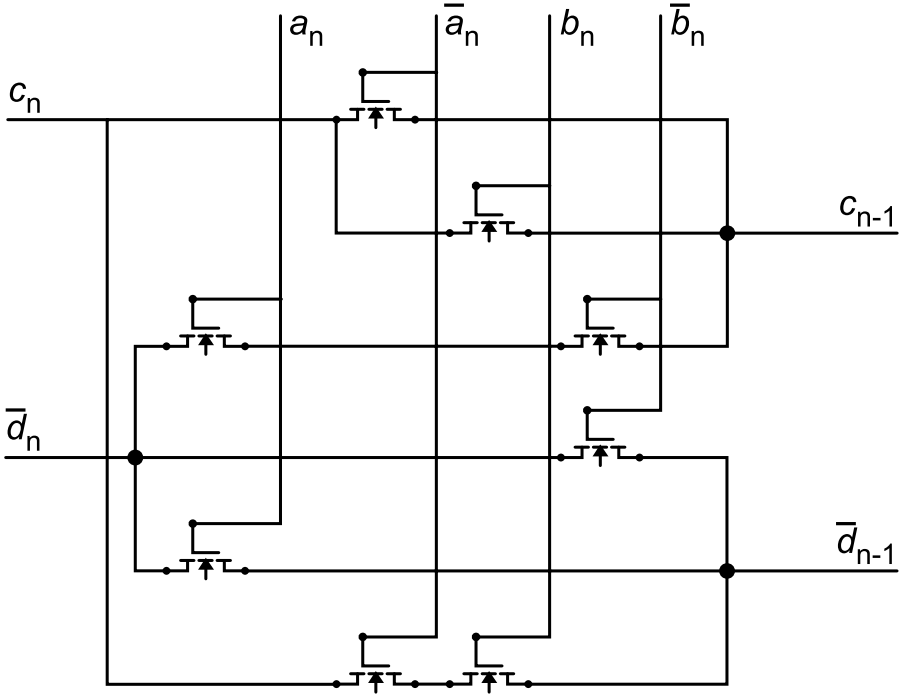
Beispiel
Bei Wahl von *false* für 0 und *true* für 1:



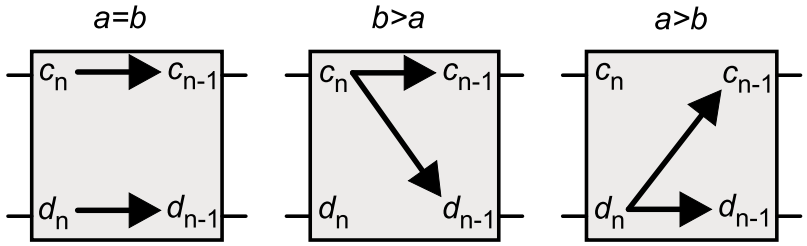
$$
\begin{aligned}
A < B &\Leftrightarrow A = 0 \wedge B = 1 \\
&\Leftrightarrow A = \mathit{false} \wedge B = \mathit{true} \\
&\Leftrightarrow \neg A \wedge B
\end{aligned}
$$

$$
\begin{aligned}
A = B &\Leftrightarrow A = 0 \wedge B = 0 \vee A = 1 \wedge B = 1 \\
&\Leftrightarrow A = \mathit{false} \wedge B = \mathit{false} \vee A = \mathit{true} \wedge B = \mathit{true} \\
&\Leftrightarrow A \wedge B \vee \neg A \wedge \neg B
\end{aligned}
$$

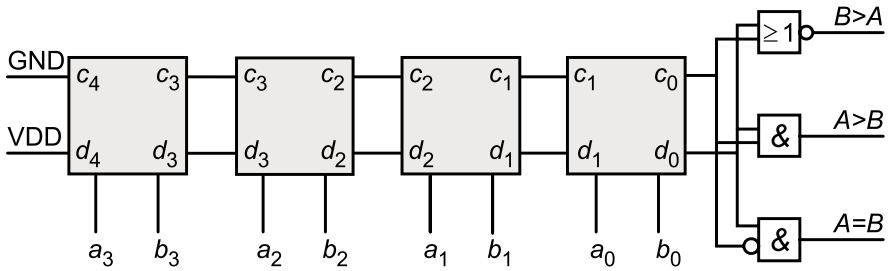
Komparator: Sparsame Variante mit Pass-Transistoren



Komparatorzelle mit Verknüpfungsmodul



Weiterleitung der Eingangspotenziale



4-Bit-Komparator

(aus N. Reifschneider: CAE-gestützte IC-Entwurfsmethoden, 1998, pp. 132f)

Komparatoren für $\mathrm{UInt}_2(l)$: Optimierungen

Tiefe des Netzwerkes durch äquivalente Umformung verringern, Voraussetzung in der Regel Gatter mit mehr als zwei Eingängen.

Extremfall: Gesamtschaltung als zweistufiges Schaltnetz. Benötigt werden:

- Und-Gatter mit Fan-in bis zu $2 \times l$,
- Oder-Gatter mit Fan-in bis zu 2^l.

Für großes l also schlecht realisierbar.

In VLSI können die Leitungslängen dominieren, baumartiger Komparator dann evtl. nicht schneller als linearer Komparator.

Beschleunigung durch synchrone Implementierung als systolisches Feld mit zeitversetzter Eingabe der Daten:

- Pro Takt werden etwa $2 \times \sqrt{l}$ Bits eingegeben.
- Latenz etwa $2 \times \sqrt{l}$ Takte.

Komparatoren für $\mathrm{UInt}_2(l)$: Weitere Techniken

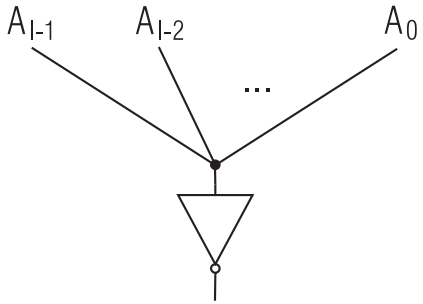
Hardware-Ersparnis durch Mitbenutzung eines Subtrahierers:

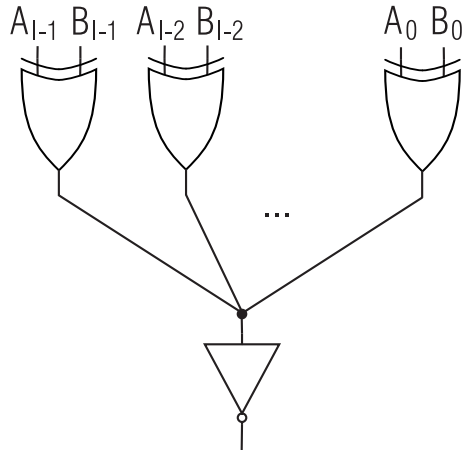
$$A < B \Leftrightarrow (B - A) >_0$$
$$A = B \Leftrightarrow (B - A) =_0$$

Prädikate mit Operand Null sind einfach zu realisieren, Gesamtlösung aber zeitaufwendiger.

Falls $B \ominus A$ undefiniert ist oder nicht mit $B - A$ in $\mathbb{Z}$ übereinstimmt, müssen die Prädikate indirekt aus dem Ablauf der Subtraktion erschlossen werden.

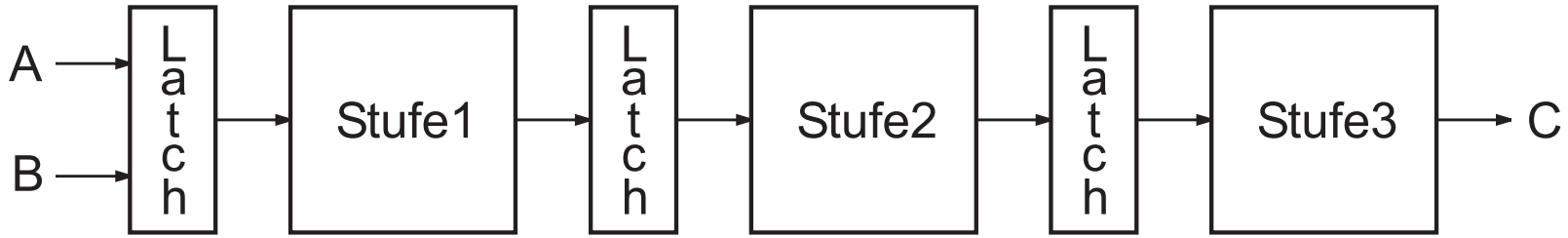
Test auf Null bzw. Gleichheit mittels „Wired-Or“:

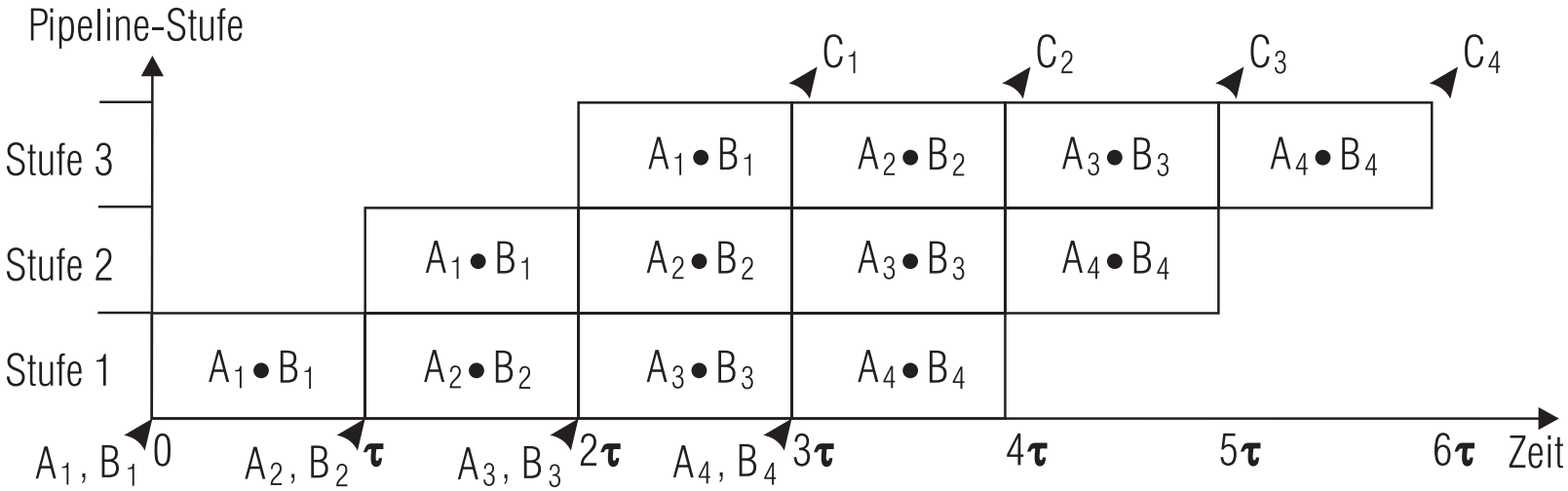




Pipelining: Erhöhung des Durchsatzes durch bessere Auslastung

Idee: Durch Einfügen von Latches in Schaltnetze können mehrere Datensätze überlappt die Gesamtschaltung durchlaufen. (Variante: Wave-Pipelining funktioniert auch ohne Latches.)





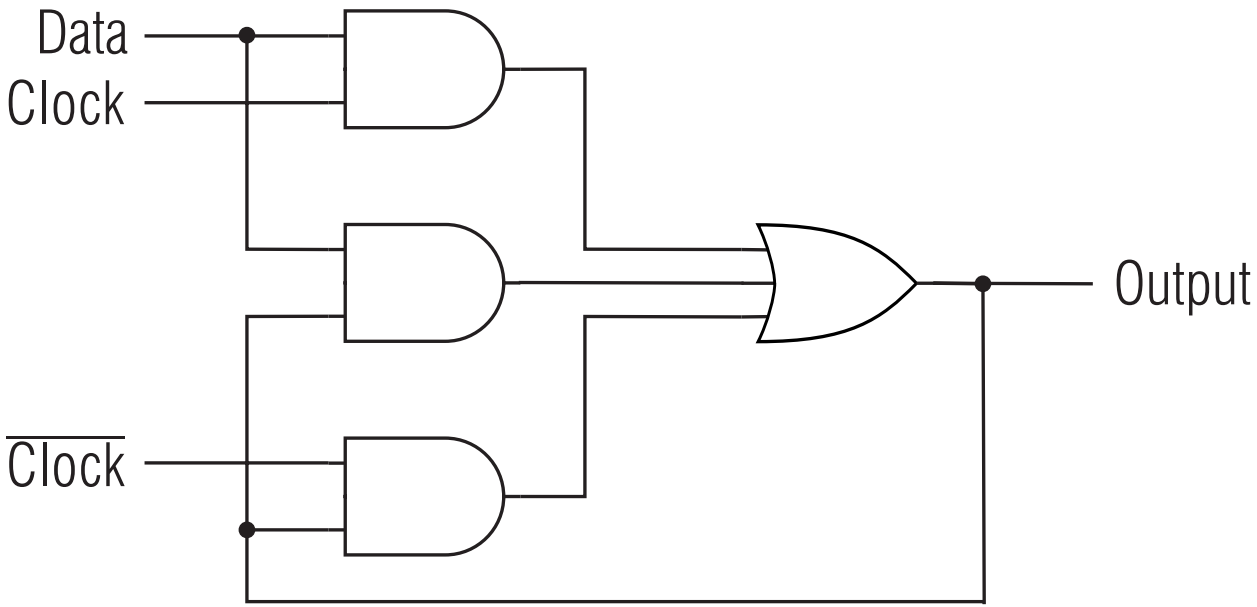
Performance einer k-stufigen Pipeline

Schaltnetz aus k Stufen mit Verzögerungen t_i und Aufwand a_i
Latenz: $\sum_i t_i$
Durchsatz: $1/\sum_i t_i$
Aufwand: $\sum_i a_i$

k-stufige Pipeline, $t_{stage} = \max_i t_i$
Latenz: $k \times (t_{stage} + t_{latch})$
Durchsatz: $1/(t_{stage} + t_{latch})$
Aufwand: $k \times a_{latch} + \sum_i a_i$

Beispiele
$t_i = t_{latch},\ k = 8 \Rightarrow$ Durchsatz vervierfacht, Latenz verdoppelt (Kompromiss).
$t_1 = t_{latch},\ t_2 = 2 \times t_{latch} \Rightarrow$ Durchsatz unverändert, Latenz verdoppelt (Verschlechterung).
$t_{latch} = 0,\ t_1 = t_2 = t_3 \Rightarrow$ Durchsatz verdreifacht, Latenz unverändert (Verbesserung).

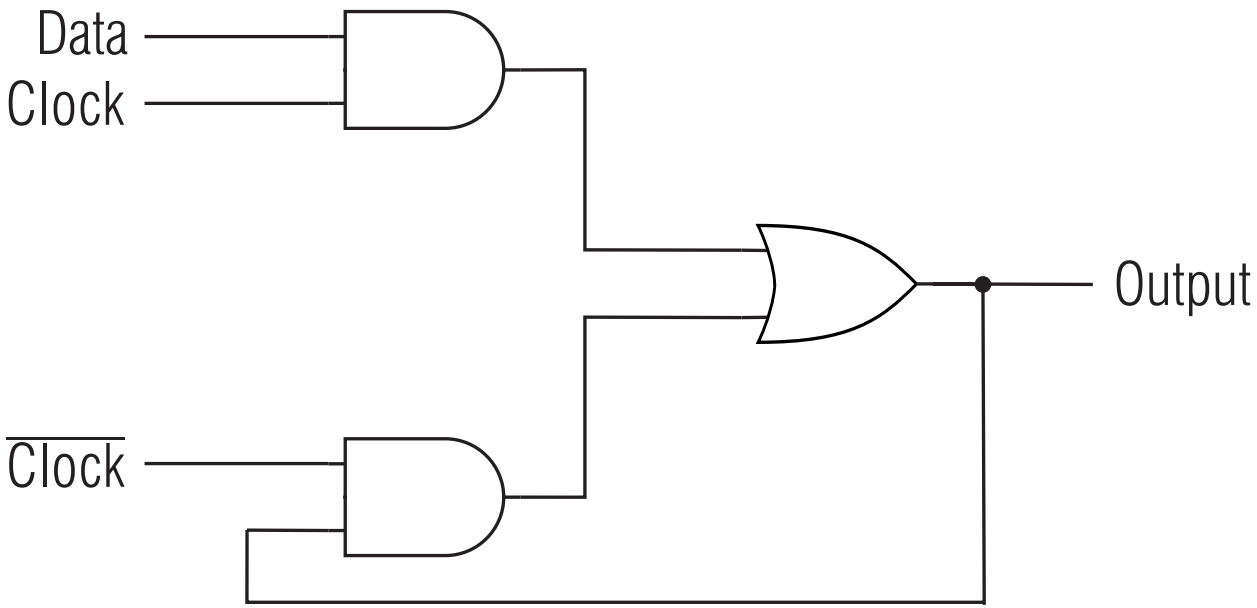
Earle-Latch: Reduzierung der Latenz einer Pipelinestufe



$$Output(t+\delta) = Data(t) \wedge Clock(t) \vee Data(t) \wedge Output(t) \vee Output(t) \wedge \neg Clock(t)$$

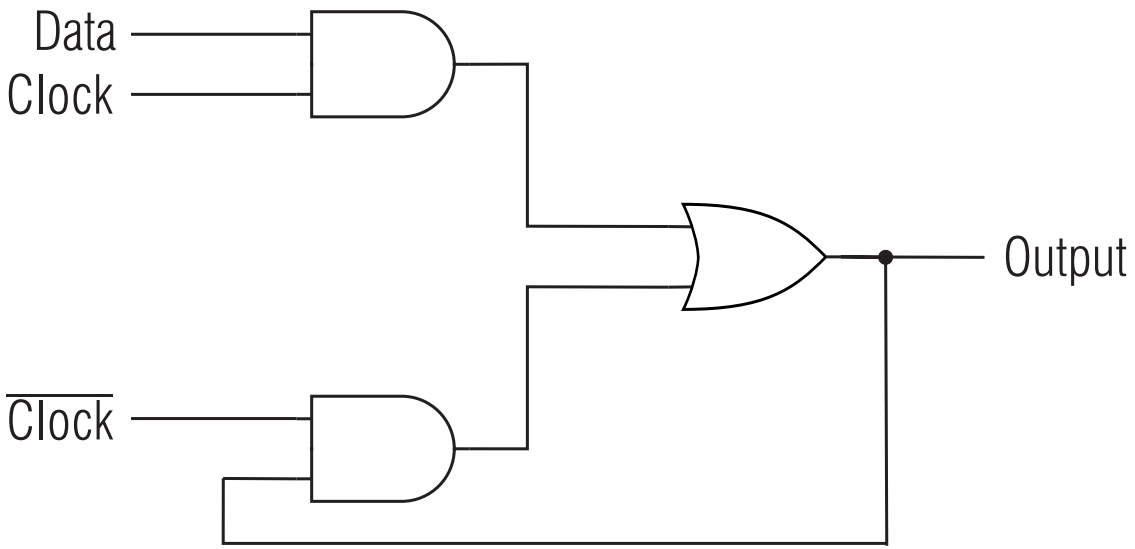
- einfaches D-Latch (pegelgesteuertes D-Flipflop)
- logische Überdeckung von Strukturhasards
- nicht getaktetes Schaltwerk, d. h., Eingänge dürfen sich nur einzeln ändern
- bei Substitution einer booleschen Funktion für *Data* und Expandierung zur disjunktiven Normalform keine Verzögerung zusätzlich zur vorgeschalteten Logik

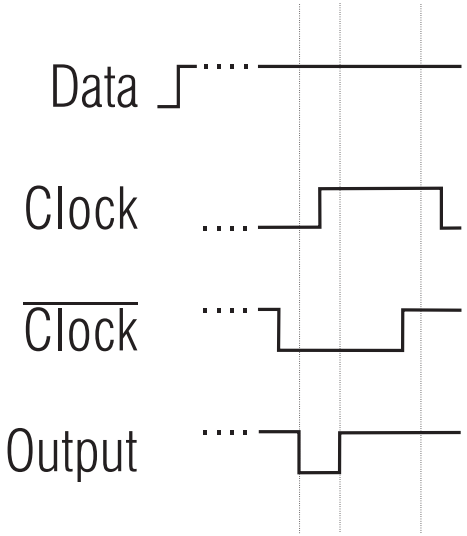
Polarity-Hold-Latch

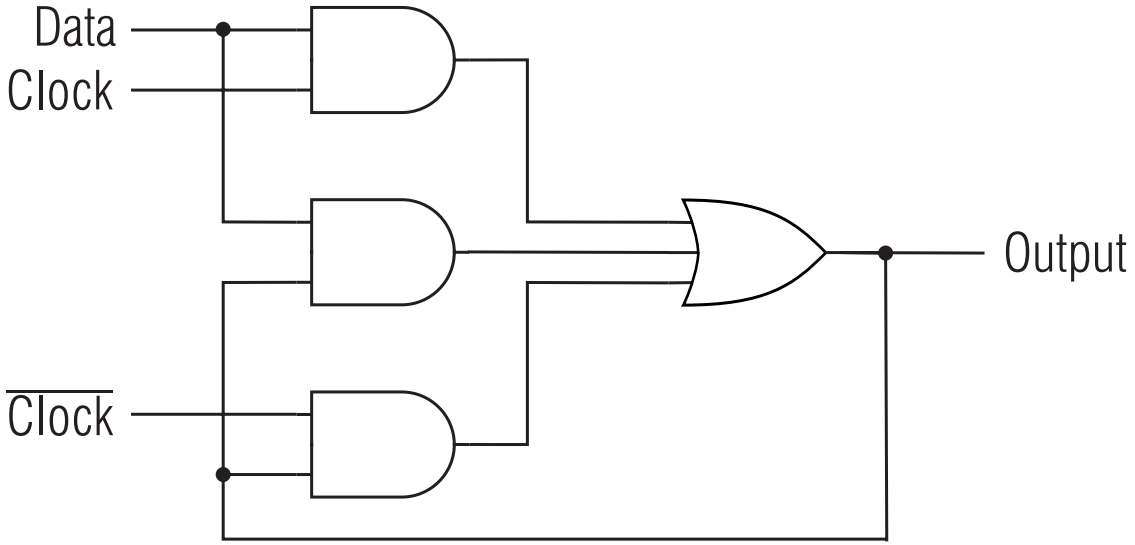


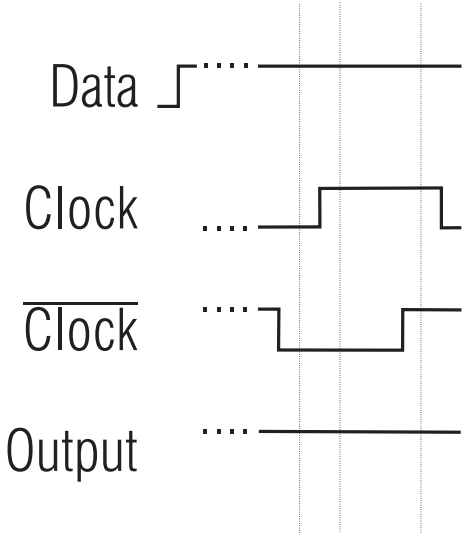
- vereinfachte Form des Earle-Latch
- gegenüber Earle-Latch etwa halber Aufwand an Gattern bzw. Transistoren
- gegenüber Earle-Latch etwa halber Fan-in am Ausgangsgatter
- zulässig, wenn Clock-Skew kontrolliert werden kann
- eingesetzt z. B. in Prozessoren von CDC, Cray, Amdahl, IBM

Earle-Latch vs. Polarity-Hold-Latch: Hasards

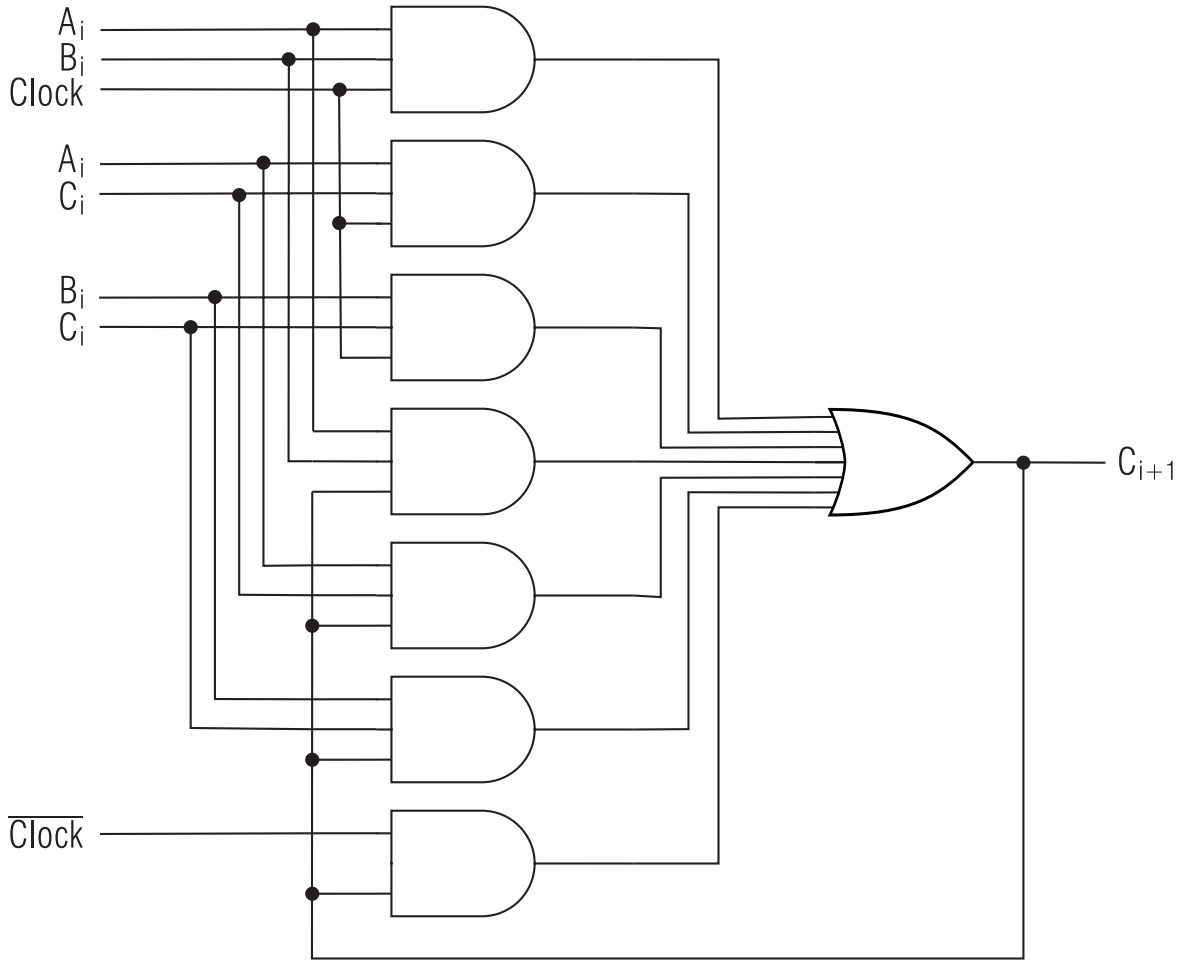






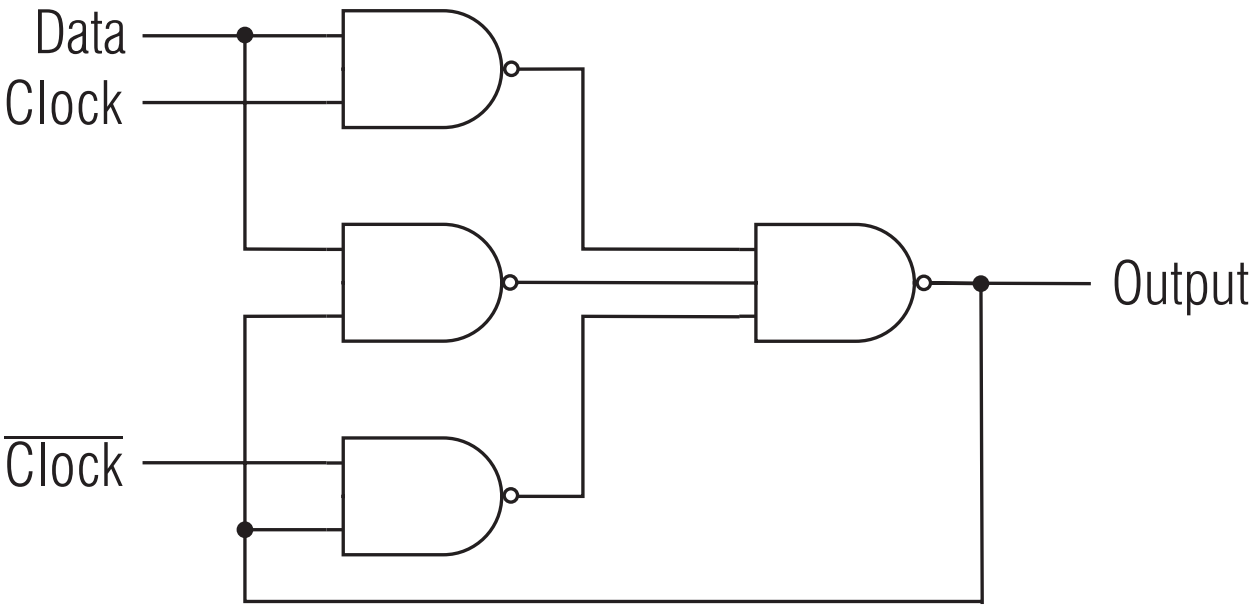


Earle-Latch: Beispiel Übertragsberechnung im Volladdierer

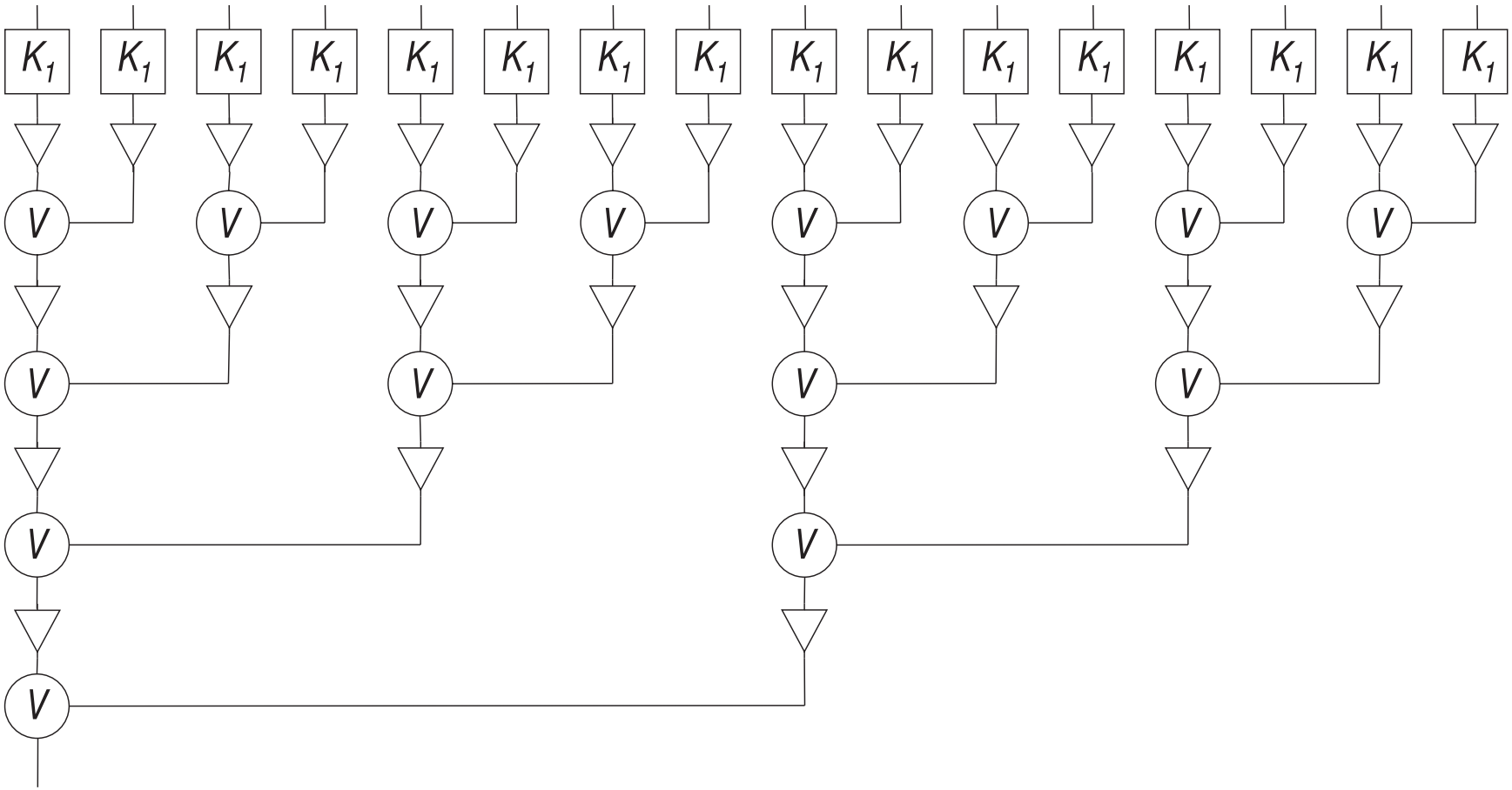


Substitution mit $C_{i+1} = A_i \wedge B_i \vee A_i \wedge C_i \vee B_i \wedge C_i$ für *Data*.

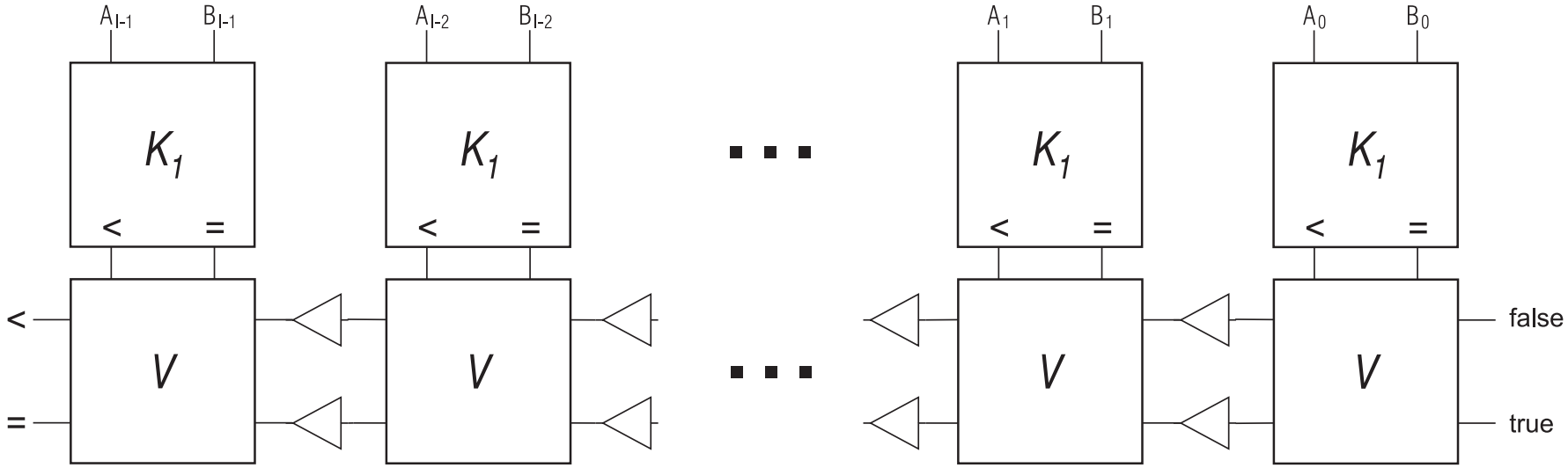
Earle-Latch: Realisierung durch NAND-Gatter



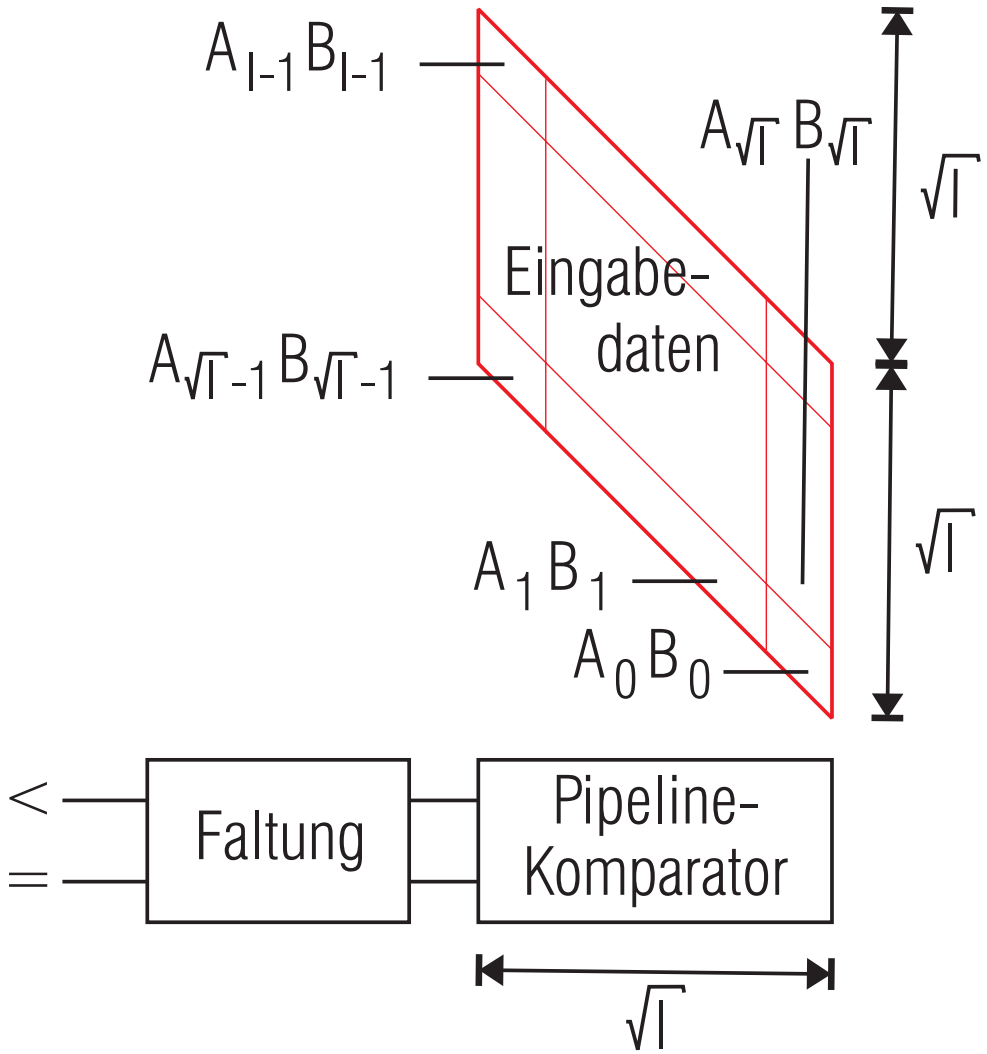
Pipelining eines baumartigen Komparators: ($\log_2$ /)-facher Durchsatz



Pipelining eines linearen Komparators: l-facher Durchsatz



Systolischer Komparator mit Datenverteilung



Komparatoren für $\mathrm{Int}_2(l)$

Definition des Prädikats $<^{(2)}$

$$(A_{l-1},\ldots,A_0) <^{(2)} (B_{l-1},\ldots,B_0) \quad \Leftrightarrow \quad (\overline{A_{l-1}},A_{l-2},\ldots,A_0) < (\overline{B_{l-1}},B_{l-2},\ldots,B_0)$$

alternativ:

$$(A_{l-1},\ldots,A_0) <^{(2)} (B_{l-1},\ldots,B_0) \quad \Leftrightarrow \quad (B_{l-1},A_{l-2},\ldots,A_0) < (A_{l-1},B_{l-2},\ldots,B_0)$$

Definition des Prädikats $=^{(2)}$

$$(A_{l-1},\ldots,A_0) =^{(2)} (B_{l-1},\ldots,B_0) \quad \Leftrightarrow \quad (A_{l-1},\ldots,A_0) = (B_{l-1},\ldots,B_0)$$

Grundlegende Rekursion für das Prädikat $<^{(2)}$

$$\forall k, 1 \le k \le l-1: \quad (A_{l-1},\ldots,A_0) <^{(2)} (B_{l-1},\ldots,B_0) \quad \Leftrightarrow$$
$$(A_{l-1},\ldots,A_k) <^{(2)} (B_{l-1},\ldots,B_k) \ \vee$$
$$(A_{l-1},\ldots,A_k) = (B_{l-1},\ldots,B_k) \ \wedge \ (A_{k-1},\ldots,A_0) < (B_{k-1},\ldots,B_0)$$

Komparatoren für $\mathrm{Int}_1(l)$

Definition des Prädikats $<^{(1)}$

$$(A_{l-1},\ldots,A_0) <^{(1)} (B_{l-1},\ldots,B_0) \quad \Leftrightarrow$$
$$(\overline{A_{l-1}},A_{l-2},\ldots,A_0) < (\overline{B_{l-1}},B_{l-2},\ldots,B_0) \quad \wedge$$
$$\neg((A_{l-1},\ldots,A_0) =_0^{(1)} \wedge (B_{l-1},\ldots,B_0) =_0^{(1)})$$

Definition des Prädikats $=^{(1)}$

$$(A_{l-1},\ldots,A_0) =^{(1)} (B_{l-1},\ldots,B_0) \quad \Leftrightarrow$$
$$(\forall i\colon A_i = B_i) \vee (A_{l-1},\ldots,A_0) =_0^{(1)} \wedge (B_{l-1},\ldots,B_0) =_0^{(1)}$$

Definition des Prädikats $=_0^{(1)}$

$$(A_{l-1},\ldots,A_0) =_0^{(1)} \quad \Leftrightarrow \quad (\forall i\colon A_i = 0) \vee (\forall i\colon A_i = 1)$$

Komparatoren für $\mathrm{Int}_{VB}(l)$

Definition des Prädikats $<^{(VB)}$

$$
\begin{aligned}
&(A_{l-1}, \ldots, A_0) <^{(VB)} (B_{l-1}, \ldots, B_0) \quad \Leftrightarrow \\
&A_{l-1} = B_{l-1} = 0 \ \wedge \ (A_{l-2}, \ldots, A_0) < (B_{l-2}, \ldots, B_0) \quad \vee \\
&A_{l-1} = B_{l-1} = 1 \ \wedge \ (B_{l-2}, \ldots, B_0) < (A_{l-2}, \ldots, A_0) \quad \vee \\
&A_{l-1} = 1 \ \wedge \ B_{l-1} = 0 \ \wedge \ \neg \left((A_{l-1}, \ldots, A_0) =_0^{(VB)} \ \wedge (B_{l-1}, \ldots, B_0) =_0^{(VB)} \right)
\end{aligned}
$$

Definition des Prädikats $=^{(VB)}$

$$
\begin{aligned}
&(A_{l-1}, \ldots, A_0) =^{(VB)} (B_{l-1}, \ldots, B_0) \quad \Leftrightarrow \\
&(\forall i \colon A_i = B_i) \ \vee \ (A_{l-1}, \ldots, A_0) =_0^{(VB)} \ \wedge (B_{l-1}, \ldots, B_0) =_0^{(VB)}
\end{aligned}
$$

Definition des Prädikats $=_0^{(VB)}$

$$
(A_{l-1}, \ldots, A_0) =_0^{(VB)} \quad \Leftrightarrow \quad \forall i \leq l-2 \colon A_i = 0
$$

Definition der Prädikate $<_0^{(*)}$ und $>_0^{(*)}$ mit $* \in \{1, 2, VB\}$

$$(A_{l-1}, \ldots, A_0) <_0^{(2)} \quad \Leftrightarrow \quad A_{l-1} = 1$$

$$(A_{l-1}, \ldots, A_0) <_0^{(1)} \quad \Leftrightarrow \quad A_{l-1} = 1 \ \wedge \ \exists i \colon A_i = 0$$

$$(A_{l-1}, \ldots, A_0) <_0^{(VB)} \quad \Leftrightarrow \quad A_{l-1} = 1 \ \wedge \ \exists i \leq l-2 \colon A_i = 1$$

$$(A_{l-1}, \ldots, A_0) >_0^{(*)} \quad \Leftrightarrow \quad A_{l-1} = 0 \ \wedge \ \exists i \colon A_i = 1$$

Rechnerarithmetik

Vorlesung im Sommersemester 2008

Eberhard Zehendner

FSU Jena

Notation / Stand 17.07.2008

Mengen

$\mathbb{N}$	die natürlichen Zahlen (mit Null)
$\mathbb{Q}$	die rationalen Zahlen
$[s, t]$	abgeschlossenes Intervall von s bis t
$\{s_1, s_2, \ldots, s_n\}$	Menge mit den Elementen s_1, s_2 bis s_n

$\mathbb{Z}$	die ganzen Zahlen
$\mathbb{R}$	die reellen Zahlen
$\mathbb{C}$	die komplexen Zahlen

$M \times N$	kartesisches Produkt von M und N
M^n	n-faches kartesisches Produkt $M \times \ldots \times M$
$(s_1, s_2, \ldots, s_n)$	Tupel mit den Komponenten s_1, s_2 bis s_n
$M \cap N$	Mengendurchschnitt (M geschnitten mit N)
$M \cup N$	Mengenvereinigung (M vereinigt mit N)
$M \setminus N$	Mengendifferenz (M ohne N)

M^*	$M \setminus \{0\}$
M^+	$\{x \in M : x > 0\}$
M^-	$\{x \in M : x < 0\}$
$s \in M$	s enthalten in M
$s \notin M$	s nicht enthalten in M
$\lvert M \rvert$	Kardinalität von M

$f\colon M \to N$	Abbildung f von M nach N
$s \equiv t \pmod{k}$	s kongruent t modulo k

$A \cong B$	A isomorph B
$\mathbb{Z}_k$	Restklassenring $\mathbb{Z}$ mod k

Operationen

$s + t$	Addition (s plus t)	$s - t$	Subtraktion (s minus t)
$s \times t$	Multiplikation(s mal t)	s/t	Division (s durch t)
$s \div t$	s durch t (ganzzahliger Anteil)	$s \,\%\, t$	Rest bezüglich $s \div t$
$s \bmod t$	s reduziert modulo t	$\sum$	Summe
$\pm s$	plus oder minus s	$\prod$	Produkt

$P \wedge Q$	P und Q	$\neg P$	nicht P
$P \vee Q$	P oder Q	$\forall$	für alle ...
$P \Rightarrow Q$	aus P folgt Q	$\exists$	es gibt ...
$P \Leftrightarrow Q$	P genau dann, wenn Q		

$P\,Q$	And-Gatter	$P \oplus Q$	Xor-Gatter
$P + Q$	Or-Gatter	$\overline{P}$	Inverter

Operatoren nach Bindungsstärke absteigend geordnet:

- Arithmetische Operatoren (Punkt vor Strich; von links nach rechts)
- Prädikate
- Logische Operatoren ($\neg$ vor $\wedge$ vor $\vee$ vor $\Rightarrow$ und $\Leftrightarrow$)
- Quantoren (von rechts nach links)

Genormte Schaltsymbole