

Inhalt

1 Motivation

2 Integer- und Festkomma-Arithmetik

- Zahlendarstellungen
- Algorithmen für Integer-Operationen
- Integer-Rechenwerke
- Rechnen bei eingeschränkter Präzision

3 Gleitkomma-Arithmetik

- Zahlendarstellungen
- Algorithmen für Gleitkomma-Operationen
- Rundung und Ausnahmesituationen
- Fehleranalyse
- Gleitkomma-Rechenwerke

Zahlendarstellungen

Darstellung **positiver ganzer Zahlen** im **polyadischen** Zahlensystem (Stellenwertsystem):

- n -stellige ganze Dezimalzahl x :
$$x = (x_{n-1} x_{n-2} \dots x_2 x_1 x_0)_{10}$$
$$= 10^{n-1} x_{n-1} + 10^{n-2} x_{n-2} + \dots + 10^1 x_1 + 10^0 x_0$$
mit $x_i \in \{0,1,2,3,4,5,6,7,8,9\}$
- n -stellige ganze Binär- oder Dualzahl y :
$$y = (y_{n-1} y_{n-2} \dots y_2 y_1 y_0)_2$$
$$= 2^{n-1} y_{n-1} + 2^{n-2} y_{n-2} + \dots + 2^2 y_2 + 2^1 y_1 + 2^0 y_0$$
mit $y_i \in \{0,1\}$
- **allgemein:** n -stellige Zahl z zur *Basis* b :
$$z = (z_{n-1} z_{n-2} \dots z_2 z_1 z_0)_b$$
$$= b^{n-1} z_{n-1} + b^{n-2} z_{n-2} + \dots + b^2 z_2 + b^1 z_1 + b^0 z_0$$
mit *Ziffer* $z_i \in \{0,1,2, \dots, b-1\}$

Zahlendarstellungen (Forts.)

Darstellung von **Festkommazahlen** im polyadischen Zahlensystem:

- Zahl zur Basis b mit k **Vorkomma-** und m **Nachkommastellen:**

$$\begin{aligned}z &= (z_{k-1} z_{k-2} \dots z_1 z_0, z_{-1} z_{-2} \dots z_{-m})_b \\ &= b^{k-1} z_{k-1} + b^{k-2} z_{k-2} + \dots + b^2 z_2 + b^1 z_1 + b^0 z_0 \\ &\quad + b^{-1} z_{-1} + b^{-2} z_{-2} + \dots + b^{-m} z_{-m}\end{aligned}$$

- Ziffern $z_{k-1} z_{k-2} \dots z_1 z_0$ stellen *ganzzahligen* Teil,
Ziffern $z_{-1} z_{-2} \dots z_{-m}$ stellen *gebrochenen* Teil von z dar

- gesamte Stellenzahl: $n = k + m$ Ziffern

- Binärzahl y mit k **Vorkomma-** und m **Nachkommastellen:**

$$\begin{aligned}y &= (y_{k-1} y_{k-2} \dots y_1 y_0, y_{-1} y_{-2} \dots y_{-m})_2 \\ &= 2^{k-1} y_{k-1} + 2^{k-2} y_{k-2} + \dots + 2^2 y_2 + 2^1 y_1 + 2^0 y_0 \\ &\quad + 2^{-1} y_{-1} + 2^{-2} y_{-2} + \dots + 2^{-m} y_{-m}\end{aligned}$$

Zahlendarstellungen (Forts.)

Darstellung **positiver** und **negativer** Binärzahlen z :

- höchstwertige Ziffer z_{n-1} stellt das Vorzeichen dar (Zahl ist positiv bei $z_{n-1} = 0$, negativ bei $z_{n-1} = 1$)

- für positive Zahl gilt stets: $z = (0 z_{n-2} z_{n-3} \dots z_1 z_0)_2$

- drei Möglichkeiten für negative Zahlen:

A) **Zweier-Komplement** : $-z = (1 \overline{z_{n-2}} \overline{z_{n-3}} \dots \overline{z_1} \overline{z_0})_2 + 1 = 2^n - z$

B) **Einer-Komplement** : $-z = (1 \overline{z_{n-2}} \overline{z_{n-3}} \dots \overline{z_1} \overline{z_0})_2 = 2^n - 1 - z$

C) **Betrag mit Vorzeichen** : $-z = (1 z_{n-2} z_{n-3} \dots z_1 z_0)_2$

- bei B und C hat die Zahl 0 zwei Darstellungen

- *Bemerkung:* auch Verallgemeinerung für beliebige Basis b (mit $\overline{z_i} = b-1-z_i$) möglich:

A) **b-Komplement** : $-z = (b-1 \overline{z_{n-2}} \overline{z_{n-3}} \dots \overline{z_1} \overline{z_0})_b + 1 = b^n - z$

B) **(b-1)-Komplement** : $-z = (b-1 \overline{z_{n-2}} \overline{z_{n-3}} \dots \overline{z_1} \overline{z_0})_b = b^n - 1 - z$

Algorithmen zur Integer-Addition

- in Abhängigkeit von der gewählten Zahlendarstellung A, B oder C existieren unterschiedliche Algorithmen zur Berechnung der n -bit Summe s zweier n -bit Binärzahlen a und b
- **Basis-Algorithmus** (für positive Zahlen): Addition modulo 2 aller n Bitstellen a_i und b_i von $i = 0$ bis $i = n-1$ (d.h. von rechts nach links) mit Berücksichtigung des Übertrags c_i aus Stelle $i-1$
- Beispiele für die binäre Addition zweier vorzeichenloser Zahlen (hier mit $n = 4$):

Beispiel 1:

$$\begin{array}{r} 0111 \quad (7) \\ + 0101 \quad (5) \\ \hline 111 \quad \leftarrow \text{Carry} \\ 1100 \quad (12) \end{array}$$

Beispiel 2:

$$\begin{array}{r} 0111 \quad (7) \\ + 1010 \quad (10) \\ \hline 111 \quad \leftarrow \text{Carry} \\ 10001 \quad (1) \end{array}$$

Überlauf \longrightarrow

Algorithmen zur Integer-Addition (Forts.)

A) Addition im Zweierkomplement, drei Fälle:

- 1) **Zahlen a und b sind positiv**: a und b werden stellenweise von rechts nach links addiert und jeweils der Übertrag c berücksichtigt:
 $s_0 = a_0 \oplus b_0$ und $c_1 = a_0 \cdot b_0$ (mit $\oplus = \text{XOR}$ bzw. Addition modulo 2)
 $s_i = a_i \oplus b_i \oplus c_i$ und $c_{i+1} = a_i \times b_i + a_i \times c_i + b_i \times c_i$ (für alle $i=1 \dots n-1$)
 Überlauf bei $s_{n-1} = c_{n-1} = 1$! (d.h. wenn $a + b > 2^{n-1} - 1$)
- 2) **Zahlen a und b sind negativ** (mit $a' = -a$ und $b' = -b$ positiv):
 $sum = a + b = (2^n - a') + (2^n - b') = 2 \cdot 2^n - (a' + b')$
 korrektes Ergebnis wäre jedoch: $s = 2^n - (a' + b') = sum - 2^n$
 \Rightarrow erforderliche Korrektur: Überlaufbit c_n ignorieren
 Überlauf bei $s_{n-1} = c_{n-1} = 0$! (d.h. wenn $a' + b' > 2^{n-1}$)
- 3) **Vorzeichen von a und b ist unterschiedlich** (o.B.d.A.: b ist negativ):
 $sum = a + b = a + (2^n - b') = 2^n - (b' - a)$ ist korrekt für $|b| > |a|$
 korrektes Ergebnis für $|b| < |a|$ wäre jedoch: $s = a - b' = sum - 2^n$
 \Rightarrow erforderliche Korrektur: Überlaufbit c_n ignorieren
 kein Überlauf möglich !

Algorithmen zur Integer-Addition (Forts.)

B) Addition im Einerkomplement, drei Fälle:

- 1) **Zahlen a und b sind positiv:**
wie beim Zweierkomplement
Überlauf bei $s_{n-1} = c_{n-1} = 1$! (d.h. wenn $a + b > 2^{n-1} - 1$)
- 2) **Zahlen a und b sind negativ** (mit $a' = -a$ und $b' = -b$ positiv):
 $sum = a + b = (2^n - 1 - a') + (2^n - 1 - b') = 2 \cdot 2^n - 2 - (a' + b')$
korrektes Ergebnis wäre jedoch: $s = 2^n - 1 - (a' + b') = sum + 1 - 2^n$
 \Rightarrow erforderliche Korrekturen: Überlaufbit c_n ignorieren und Addition von 1 (auch als „end-around carry“ bezeichnet)
Überlauf bei $s_{n-1} = c_{n-1} = 0$! (d.h. wenn $a' + b' > 2^{n-1} - 1$)
- 3) **Vorzeichen von a und b ist unterschiedlich** (o.B.d.A.: b ist negativ):
 $sum = a + b = a + (2^n - 1 - b') = 2^n - 1 - (b' - a)$
für $|b| > |a|$ ist Ergebnis bereits korrekt
für $|b| < |a|$ wäre korrektes Ergebnis jedoch: $s = a - b' = sum + 1 - 2^n$
 \Rightarrow erforderliche Korrekturen: Überlaufbit c_n ignorieren, Addition von 1
kein Überlauf möglich !

Algorithmen zur Integer-Addition (Forts.)

C) Addition bei Betrag mit Vorzeichen, zwei Fälle:

- 1) **Zahlen a und b sind beide positiv oder beide negativ:**
Addition der Beträge $|a|$ und $|b|$ wie bei Zweierkomplement
separate Bestimmung des Vorzeichenbits s_{n-1}
Überlauf bei $c_{n-1} = 1$! (d.h. wenn $|a| + |b| > 2^{n-1} - 1$)
- 2) **Vorzeichen von a und b ist unterschiedlich:**
 - a) für $|b| > |a|$ gilt: $|s| = |b| - |a|$, Vorzeichen $s_{n-1} = b_{n-1}$
 - b) für $|b| < |a|$ gilt: $|s| = |a| - |b|$, Vorzeichen $s_{n-1} = a_{n-1}$
 - c) für $|b| = |a|$ gilt: $|s| = 0$, Vorzeichen $s_{n-1} = 0$ oder $s_{n-1} = 1$*Problem:* Feststellung, ob a), b) oder c) zutrifft, erfordert Subtrahierer !
Lösung: Bilde Einerkomplement von $|b|$ und addiere dieses zu $|a|$:
 $sum = |a| + 2^{n-1} - 1 - |b| = 2^{n-1} - 1 + (|a| - |b|)$
Ist nun $|b| < |a|$, so ist $sum \geq 2^{n-1}$ und Überlauf tritt auf. Zur Bildung des korrekten Ergebnisses $s = |a| - |b|$ muß Überlaufbit c_{n-1} ignoriert und eine 1 addiert werden !
Ist nun $|b| > |a|$, so kann kein Überlauf auftreten; sum muß jedoch zur Bildung des korrekten Ergebnisses $s = |b| - |a|$ komplementiert werden

Algorithmen zur Integer-Addition (Forts.)

Zusammenfassung und Bewertung:

- **Zweierkomplement** :
 - aufwendige Generierung von $-z$ (benötigt einen Addierer)
 - Addition von positiven und negativen Zahlen jedoch einheitlich und einfach: lediglich Überlaufbit ist zu ignorieren !
 - **Einerkomplement**:
 - einfache Generierung von $-z$ durch Komplementbildung
 - korrigierende Korrektur um 1 nach Überlauf bei Addition negativer Zahlen erfordert zusätzliche zweite Addition !
 - **Betrag mit Vorzeichen**:
 - einfachste Generierung von $-z$ durch Invertierung des Vorzeichens
 - Addition von Zahlen unterschiedlichen Vorzeichens erfordert Vergleich; nur unter Zuhilfenahme des Einerkomplements zu realisieren !
- ⇒ *Zweierkomplement ist vorzuziehen, da die Addition die wichtigste arithmetische Operation darstellt !*

Implementierung

- Grundelement für die Realisierung eines Addierwerkes ist der **Volladdierer** (FA=„Full Adder“) der Summe s_i und Übertrag c_{i+1} aus a_i , b_i und c_i ermittelt:

Wahrheitstabelle:

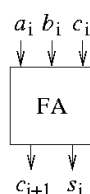
a_i	b_i	c_i	s_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Funktion:

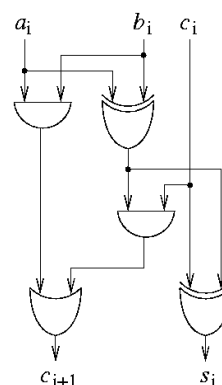
$$c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$$

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

Symbol:



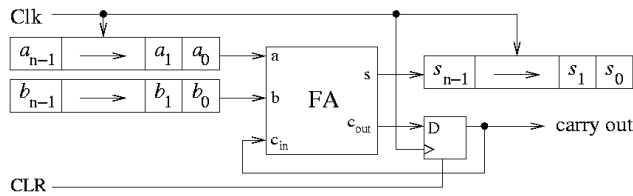
Realisierung:



- Verzögerung je nach Realisierung: $2\hat{\delta}$ bis $3\hat{\delta}$ (mit $\hat{\delta}$ = Gatterlaufzeit)

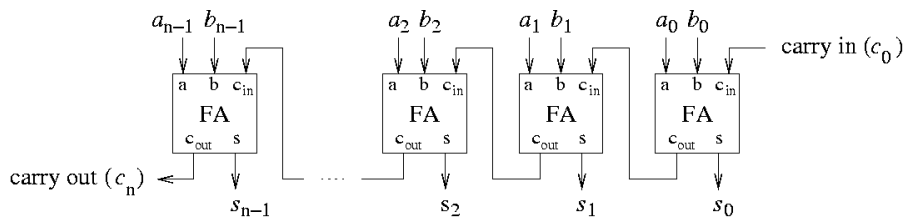
Einfache Addierwerke

- **serielles binäres Addierwerk** für zwei n -Bit Zahlen a und b :



serielle Addition
benötigt n Takte
bzw. mindestens
die Zeit $3n\hat{\delta}$
(mit Gatterlaufzeit $\hat{\delta}$)

- **paralleles binäres Addierwerk** für zwei n -Bit Zahlen a und b :



Carry-Ripple Addierer

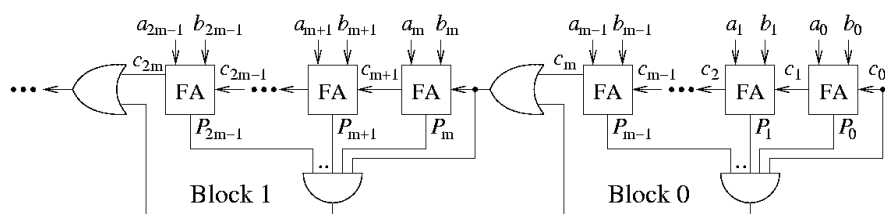
- paralleles n -stelliges Addierwerk wird auch als **Carry-Ripple Addierer** bezeichnet
- Aufwand: n FAs, entspricht $14n$ CUs („*Cost Units*“)
(einfaches Gatter mit k Eingängen: k CUs, bei XOR-Gatter: $2k$ CUs)
- im ungünstigsten Fall muß ein Carry-Signal c_1 aus Stelle 0 alle FAs durchlaufen, resultierende maximale Verzögerung:
 - $1\hat{\delta}$ zur Bildung aller $a_i \cdot b_i$ und $a_i \oplus b_i$ sowie von c_1
 - $(n - 2) \cdot 2\hat{\delta}$, damit Übertrag alle Stellen 1, 2, ..., $n-2$ durchlaufen kann
 - $2\hat{\delta}$ zur finalen Berechnung von s_{n-1} und c_n
 insgesamt: $(2n-1)\hat{\delta}$
- im Mittel läuft Carry-Signal jedoch nur über $\log_2(n)$ Stellen !
(lt. Omondi: *Computer Arithmetic Systems*, S. 25-26)
⇒ „*Carry Completion*“ Addierer mit zusätzlicher Logik zur Erkennung des Endes einer Carry-Propagation (→ Übung)

Carry-Skip Addierer

- *Idee:* Carry-Signal kann Bitpositionen i bis $i + m$ überspringen („skip“), wenn für alle $i, \dots, i + m$ gilt: $a_i = 1$ oder $b_i = 1$
- Aufbau eines **Carry-Skip Addierers**:
 - Addiererezellen werden in Gruppen der Größe m zusammengefasst (es sei hier angenommen, dass die Wortbreite n ein Vielfaches von m ist)
 - jede Addiererezelle i generiert zusätzlich ein Signal P_i , das angibt, ob Stelle i ein Carry-Signal propagiert ($p_i = 1$) oder nicht ($p_i = 0$)
Es gilt: $P_i = a_i + b_i$
 - zusätzliche Logik in jeder Gruppe generiert Signal C , das angibt, ob ein Block mit den Addiererezellen $i, i+1, \dots, i+m-1$ ein Carry-Signal an den nächsthöheren Block liefert
Es gilt: $C = (P_i \cdot P_{i+1} \cdot \dots \cdot P_{i+m-2} \cdot P_{i+m-1}) c_i + c_{i+m}$
- Aufwand je Block: $m \times \text{Aufwand(FA)} + 3m + 2 \text{ CUs} = 17m + 2 \text{ CUs}$
- Gesamtaufwand (bei n/m Blöcken): $17n + 2n/m \text{ CUs}$

Carry-Skip Addierer (Forts.)

- Realisierung eines Carry-Skip Addierers:

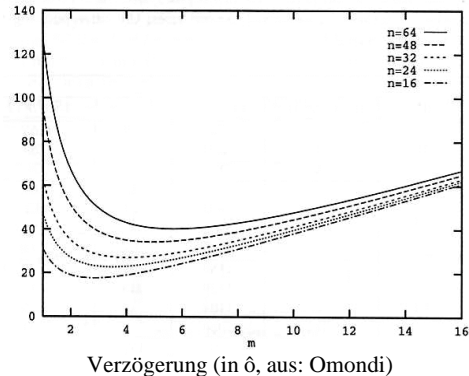
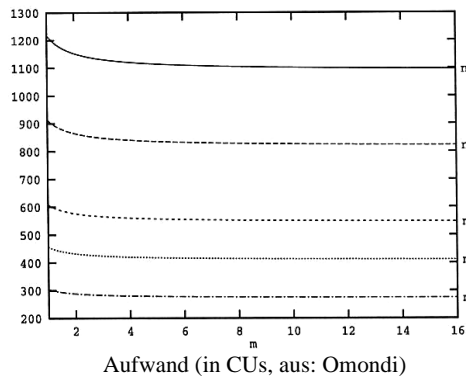


- Bestimmung der maximalen („worst case“) Verzögerung:
 - 1δ zur Generierung aller P_i und eines Carry an Bitstelle 0 in Block 0
 - in Block 0 wird Carry über $m-1$ Stellen propagiert: $2(m-1)\delta$
 - in Block $n/m-1$ wird Carry an höchstwertiger Stelle absorbiert: $2(m-1)\delta$
 - in den dazwischen liegenden Blöcken wird Carry propagiert: $2(n/m-2)\delta + \delta$
 - 1δ zur finalen Berechnung von s_{n-1}
 insgesamt: $(2n/m + 4m - 5)\delta$

Carry-Skip Addierer (Forts.)

Variation
von n
und m :

Wortbreite n	16			32				64				
Gruppengröße m	2	4	8	2	4	8	16	2	4	8	16	32
Aufwand (CUs)	288	300	330	576	600	660	786	1152	1200	1320	1572	2082
Verzögerung ($\hat{\delta}$)	18	14	18	34	22	22	34	66	38	30	38	66



Computer-Arithmetik, WS 2002/2003
A. Strey, Universität Ulm

Kapitel 2 : Integer-Arithmetik
15

Carry-Lookahead Addierer (CLA)

- **Idee: Vorausberechnung** der Carry-Signale c_i für alle n Stellen
- für i -ten Volladdierer gilt: $c_{i+1} = a_i b_i + (a_i + b_i) c_i := G_i + P_i c_i$
 - $G_i = a_i b_i$ gibt an, ob in Stelle i ein Carry-Signal erzeugt wird („Generate“)
 - $P_i = a_i + b_i$ gibt an, ob Stelle i das Carry-Signal propagiert (=1) oder nicht (=0)
- für die c_i der ersten Stellen ergibt sich:

$$c_1 = a_0 b_0 + (a_0 + b_0) c_0 := G_0 + P_0 c_0$$

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

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

$$c_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$
- alle Signale c_i lassen sich in der Zeit $3\hat{\delta}$ bestimmen, die Summe $s = a + b$ läßt sich (unabhängig von n) in der Zeit $4\hat{\delta}$ bestimmen (jedoch sind große UND-Gatter mit max. $n+1$ Eingängen und ODER-Gatter mit max. n Eingängen nötig \Rightarrow Annahme eines einheitlichen $\hat{\delta}$ unrealistisch !)

Computer-Arithmetik, WS 2002/2003
A. Strey, Universität Ulm

Kapitel 2 : Integer-Arithmetik
16