Rechnerstrukturen

Michael Engel und Peter Marwedel

TU Dortmund, Fakultät für Informatik

Sommer 2014

Folien a. d. Basis von Materialien von Gernot Fink und Thomas Jansen

28. April 2014

1. Boolesche Funktionen und Schaltnetze
   - Rechner-Arithmetik
     - Addition (Wiederholung)
   - Multiplikation
     - Wallace-Tree
   - Subtraktion
     - Addition negativer Zahlen
   - Gleitkommazahlen-Arithmetik
     - Multiplikation von Gleitkommazahlen
     - Addition von Gleitkommazahlen

Addition von Binärzahlen

Beobachtungen zu den Überträgen

- „$1 + 1$“ erzeugt einen Übertrag
- „$0 + 0$“ eliminiert einen vorhandenen Übertrag
- „$0 + 1$“ und „$1 + 0$“ reichen einen vorhandenen Übertrag weiter
- Übertrag ist höchstens 1

Kann man Addition als boolesche Funktion ausdrücken?

$f_{\mathrm{HA}} \colon \{0,1\}^2 \to \{0,1\}^2$ mit

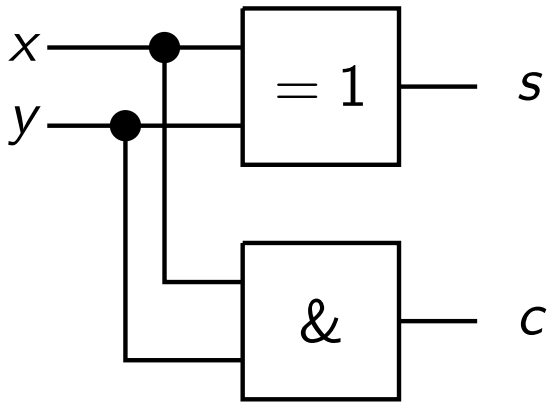
x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

realisiert Addition mit Summenbit s und Übertrag c (Carry)

Beobachtung $\quad c = x \wedge y \quad s = x \oplus y$

Halbaddierer

$c = x \wedge y \quad s = x \oplus y$



Größe 2

Tiefe 1

Drückt $f_{\mathsf{HA}} : \{0,1\}^2 \to \{0,1\}^2$ mit

x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

wirklich Addition aus?

Beobachtung: Nur für isolierte Ziffern, vorheriger Übertrag fehlt!

Realisierung von Addition als boolesche Funktion

$f_{\mathrm{VA}} : \{0,1\}^3 \to \{0,1\}^2$ mit

c_{alt}	x	y	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

realisiert Addition mit Summenbit s und Übertrag c (Carry)

Beobachtung $s = 1 \Leftrightarrow$ Anzahl der Einsen ungerade

$s = c_{\mathrm{alt}} \oplus x \oplus y$

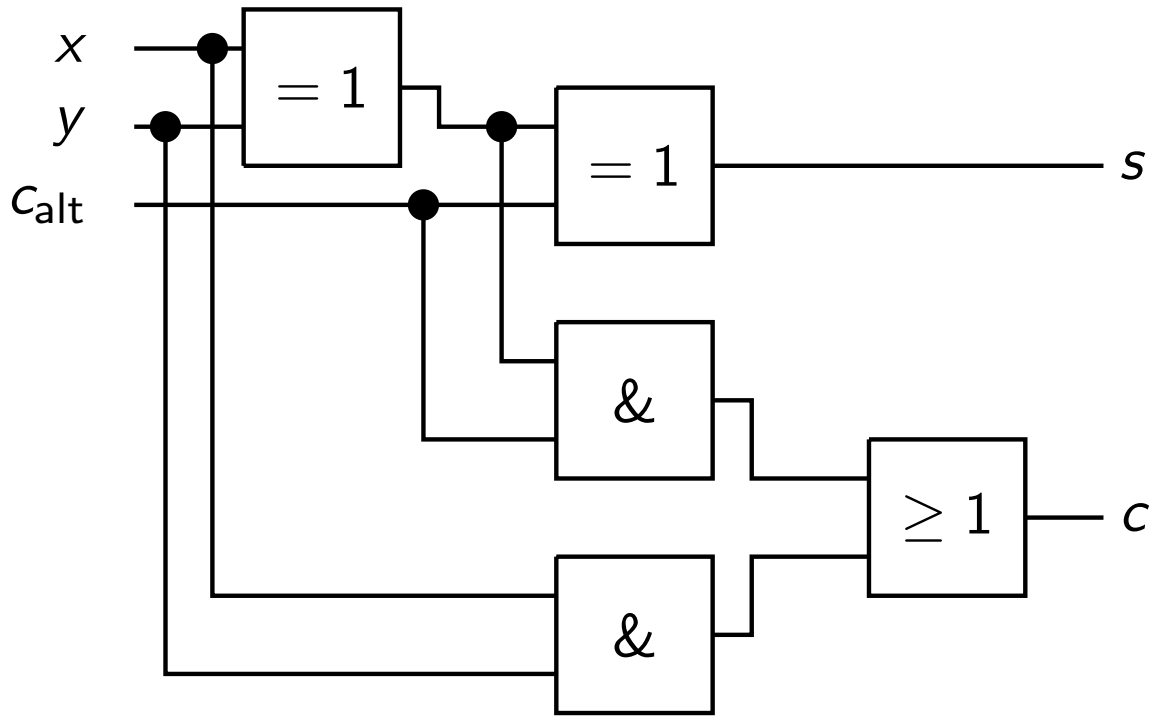
Beobachtung $c = 1 \Leftrightarrow$ Anzahl Einsen ≥ 2

direkte Realisierung $c = x\,y \vee x\,c_{\mathrm{alt}} \vee y\,c_{\mathrm{alt}}$

aber für Realisierung im Schaltnetz

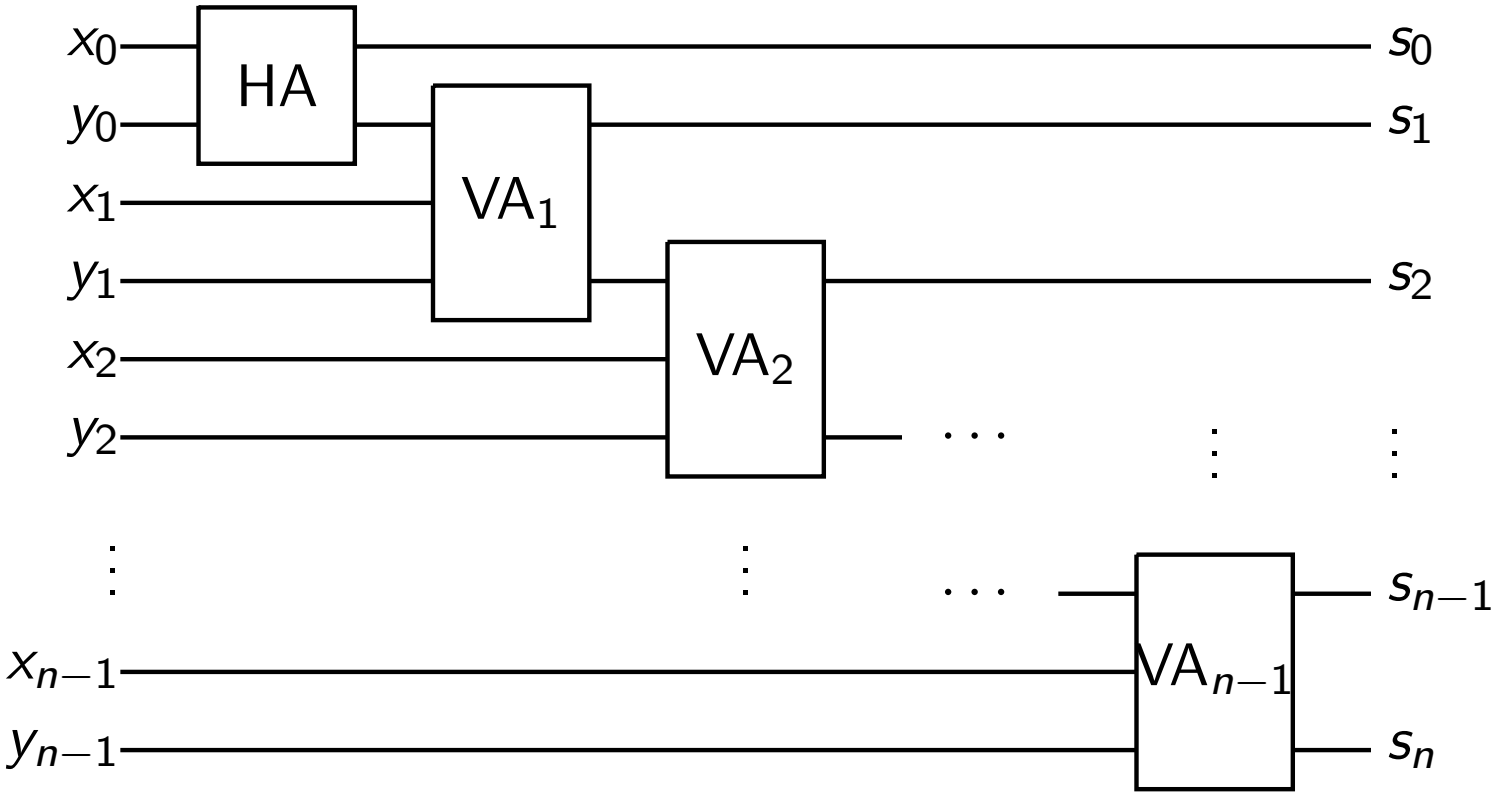
Schaltnetz Volladdierer

$s = x \oplus y \oplus c_{\text{alt}}$
$c = c_{\text{alt}} (x \oplus y) \vee x\, y$



Größe 5, Tiefe 3

Realisierung Addition



Größe $2 + (n-1) \cdot 5 = 5n - 3$

Tiefe $1 + (n-1) \cdot 3 = 3n - 2$

Ergebnis: Ripple-Carry Addierer

Realisierung „Addition von natürlichen Zahlen“

- sehr gut strukturiert
- Größe $5n - 3$ $\leftarrow$ sehr klein
- Tiefe $3n - 2$ $\leftarrow$ **viel** zu tief

Warum ist unser Schaltnetz so tief?

offensichtlich	Überträge brauchen sehr lange
Verbesserungsidee	Überträge früher berechnen
Erinnerung	Struktureinsicht

$(x_i, y_i) = (1, 1)$	generiert Übertrag
$(x_i, y_i) = (0, 0)$	eliminiert Übertrag
$(x_i, y_i) \in \{(0, 1), (1, 0)\}$	reicht Übertrag weiter

Multiplikation

direkt mit Binärzahlen...

1	0	1	0	1	1	0	0	·	1	1	1	0	1	0
														0
						1	0	1	0	1	1	0	0	
												0		
				1	0	1	0	1	1	0	0			
			1	0	1	0	1	1	0	0				
		1	0	1	0	1	1	0	0					
	1	0	0	1	1	0	1	1	1	1	1	0	0	0

also Multiplizieren heißt

- Nullen passend schreiben
- Zahlen passend verschoben kopieren
- viele Zahlen addieren

Multiplikation als Schaltnetz

Multiplikation ist

- Nullen passend schreiben
- Zahlen passend verschoben kopieren
- viele Zahlen addieren

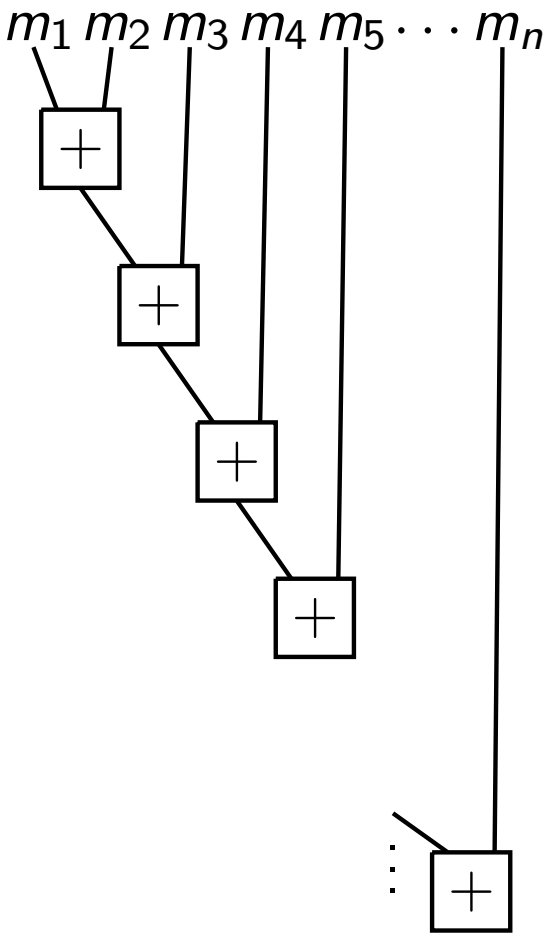
klar Nullen schreiben einfach und kostenlos,
Zahlen verschieben und kopieren einfach und kostenlos,
viele Zahlen addieren nicht ganz so einfach

Addition vieler Zahlen

klar Wir haben Addierer für die Addition zweier Zahlen.

Wie addieren wir damit n Zahlen?

erster Ansatz einfach nacheinander



Tiefe $(n-1) \cdot$ Tiefe$(+)$ Schrecklich!

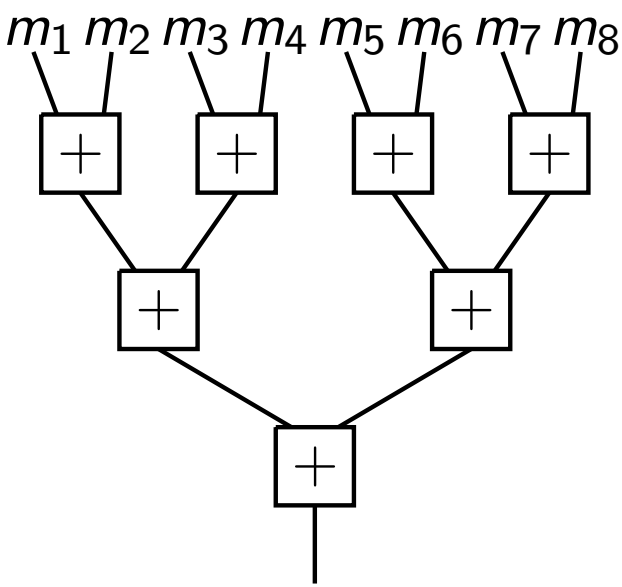
klar sehr naiv

merken in Schaltnetzen niemals alles nacheinander

Was geht gleichzeitig?

Addition von n Zahlen

besserer Ansatz paarweise addieren



Anzahl Addierer $= \frac{n}{2} + \frac{n}{4} + \cdots 1 \approx n$

Gesamtgröße $\approx n \cdot$ Größe$(+)$

Tiefe auf i-ter Ebene

$2^{\log_2(n)-i}$ Addierer

also $\approx \log_2(n)$ Ebenen

Gesamttiefe

$$\approx \log_2(n) \cdot 2\log_2(n) = 2\log_2(n)^2$$

Geht es vielleicht noch schneller?

Addition von n Zahlen noch schneller

(triviale) Beobachtung — Addition ersetzt zwei Zahlen durch eine Zahl gleicher Summe

zentral für uns

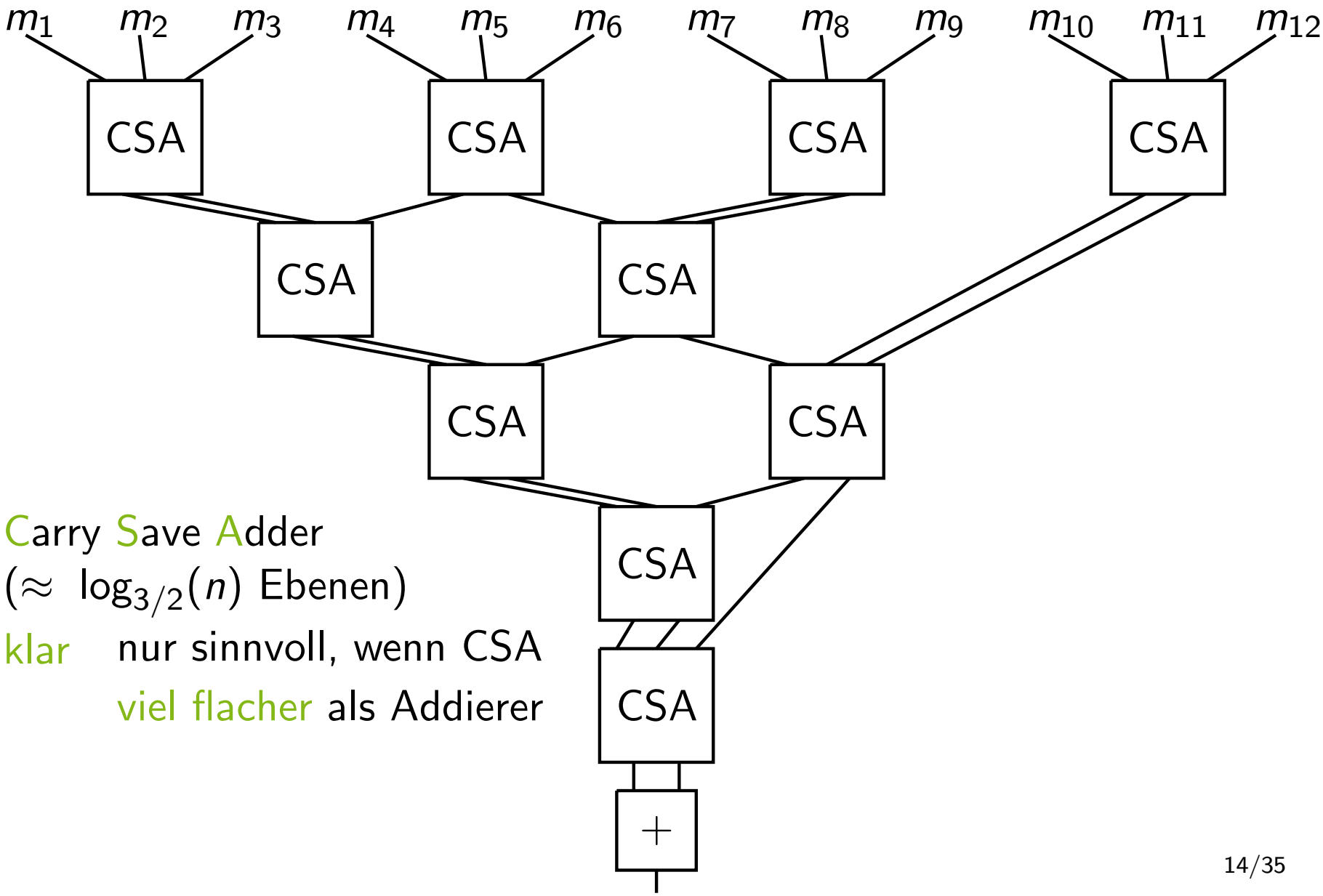
- gleiche Summe $\rightsquigarrow$ Korrektheit
- weniger Zahlen $\rightsquigarrow$ „Fortschritt“

(verrückte?) Idee — Vielleicht ist es einfacher, drei Zahlen zu ersetzen durch zwei Zahlen gleicher Summe?

klar — immer noch „gleiche Summe $\rightsquigarrow$ Korrektheit“

aber — „$3 \rightsquigarrow 2$“ statt „$2 \rightsquigarrow 1$“
weniger Fortschritt — Ist das schlimm?

Wallace-Tree



Carry Save Adder

($\approx \log_{3/2}(n)$ Ebenen)

klar nur sinnvoll, wenn CSA viel flacher als Addierer

Carry Save Adder

gesucht Zahlen a, b mit $a + b = x + y + z$

Beobachtung für $x, y, z \in \{0, 1\}$ schon bekannt
$x + y + z = 2 \cdot u + v$ (Volladdierer)

Beobachtung Es genügt, das parallel
für alle Stellen zu tun.

		x_3	x_2	x_1	x_0
$+$		y_3	y_2	y_1	y_0
$+$		z_3	z_2	z_1	z_0
	$(2u_3 + v_3)$	$(2u_2 + v_2)$	$(2u_1 + v_1)$	$(2u_0 + v_0)$	
	u_3	u_2	u_1	u_0	
$+$		v_3	v_2	v_1	v_0

Korrektheit der CSA-Realisierung

aus Volladdierer $x_i + y_i + z_i = 2u_i + v_i$

$$
\begin{aligned}
x + y + z &= \left(\sum_{i=0}^{n-1} x_i \cdot 2^i\right) + \left(\sum_{i=0}^{n-1} y_i \cdot 2^i\right) + \left(\sum_{i=0}^{n-1} z_i \cdot 2^i\right) \\
&= \sum_{i=0}^{n-1} \left((x_i + y_i + z_i) \cdot 2^i\right) \\
&= \sum_{i=0}^{n-1} \left((2u_i + v_i) \cdot 2^i\right) \\
&= \left(\sum_{i=0}^{n-1} u_i \cdot 2^{i+1}\right) + \left(\sum_{i=0}^{n-1} v_i \cdot 2^i\right)
\end{aligned}
$$

Multiplikation mit Wallace-Tree

klar für drei n-Bit-Zahlen reichen n Volladdierer

also
	Größe	$5n$
	Tiefe	3

Gesamtgröße $\sim n^2$

Gesamttiefe $\approx \underbrace{3 \cdot \log_{3/2}(n)}_{\text{Wallace-Tree}} + \underbrace{2 \log_2(n)}_{\text{Addierer}} \approx 7{,}13 \log_2(n)$

Fazit Multiplikation wesentlich „teurer“ als Addition, aber nicht wesentlich langsamer!

Subtraktion

Beobachtung Niemand muss subtrahieren!

Begründung Statt „$x - y$“ einfach „$x + (-y)$“ rechnen!

also Idee Ersetze Subtraktion durch

1. Vorzeichenwechsel
2. Addition einer eventuell negativen Zahl

noch zu untersuchen

1. Wie schwierig ist der Vorzeichenwechsel?
2. Wie funktioniert die Addition von negativen Zahlen?

Vorzeichenwechsel

Repräsentation	**Vorgehen**	**Kommentar**
Vorzeichen-Betrag	Vorzeichen-Bit invertieren	sehr einfach
Einerkomplement	alle Bits invertieren	einfach
Zweierkomplement	alle Bits invertieren, 1 addieren	machbar
Exzess	Subtraktion von $2y$	schwierig

Beobachtung für Exzessdarstellung funktioniert
Addierer selbst bei positiven Zahlen nicht
„$x + y$“ $\rightsquigarrow (b + x) + (b + y) = (b + x + y) + b$

also Exzessdarstellung fürs Rechnen weitgehend ungeeignet,
nur günstig für Vergleiche

Addition negativer Zahlen

klar unsere Schaltnetze für die Addition (Schulmethode, Carry-Look-Ahead) sind für Betragszahlen entworfen

Müssen wir für negative Zahlen komplett neu entwerfen?

klar Das hängt von der Repräsentation ab.

Exzessdarstellung Betrachten wir gar nicht, weil wir damit nicht einmal addieren können.

Vorzeichen-Betrag positive und negative fast gleich dargestellt, darum neuer Schaltnetzentwurf erforderlich

Addition negativer Zahlen im Einerkomplement

auf dieser und nächster Folie

- Notation $\overline{y}$ ist Komplement von y
- Wir wechseln frei zwischen Zahlen und ihren Repräsentationen.
- feste Darstellungslänge ℓ

Beobachtung $y + \overline{y} = 2^\ell - 1$
$\Leftrightarrow \overline{y} = 2^\ell - 1 - y$

Rechne
$x - y = x + (-y) = x + \overline{y} = x + 2^\ell - 1 - y = 2^\ell + (x - y) - 1$

Wichtig Darstellungslänge $\Rightarrow 2^\ell$ „passt nicht“ (Überlauf)

also Überlauf ignorieren $\Rightarrow$ noch 1 addieren $\rightsquigarrow$ korrekt

Addition negativer Zahlen im Zweierkomplement

Beobachtung $y + \overline{y} = 2^\ell - 1$
$\Leftrightarrow \overline{y} = 2^\ell - 1 - y$

Rechne
$x - y = x + (-y) = x + \overline{y} + 1 = x + 2^\ell - 1 - y + 1 = 2^\ell + (x - y)$

wichtig Darstellungslänge $\Rightarrow 2^\ell$ „passt nicht“ (Überlauf)

also Überlauf ignorieren $\Rightarrow$ korrekt

also Addierer rechnet richtig auch für negative Zahlen

$\rightsquigarrow$ Zweierkomplement verbreitetste Darstellung ganzer Zahlen

Überträge bei Addition im Zweierkomplement

Wann ist das Ergebnis korrekt und wann nicht darstellbar?

1. Addition zweier positiver Zahlen
   klar Ergebnis positiv
   Beobachtung kein Überlauf möglich
   also Ergebnis korrekt, wenn positiv
2. Addition einer positiven und einer negativen Zahl
   klar Ergebnis kleiner als größte darstellbare Zahl
   klar Ergebnis größer als kleinste darstellbare Zahl
   also Ergebnis immer korrekt
3. Addition zweier negativer Zahlen
   gesehen Überlauf entsteht ($\rightsquigarrow$ ignorieren)
   klar Ergebnis negativ
   also Ergebnis korrekt, wenn negativ

Gleitkommazahlen-Arithmetik

Darstellung gemäß IEEE 754-1985

$$\begin{aligned} x &= (-1)^{s_x} \cdot m_x \cdot 2^{e_x} \\ y &= (-1)^{s_y} \cdot m_y \cdot 2^{e_y} \end{aligned}$$

s Vorzeichenbit

m Mantisse (Binärdarstellung, inklusive impliziter 1)

e Exponent (Exzessdarstellung, $b = 2^{\ell-1} - 1$)

Ergebnis $z = (-1)^{s_z} \cdot m_z \cdot 2^{e_z}$

Vereinfachung Wir ignorieren das Runden.

Aber: Wichtiger Teil des IEEE 754 Standards!

Weitere Details z.B. in:

David Goldberg (1991): What every computer scientist should know about floating-point arithmetic. ACM Computing Serveys 23(1):5–48.

Multiplikation von Gleitkommazahlen

$$\begin{aligned} x &= (-1)^{s_x} \cdot m_x \cdot 2^{e_x} \\ y &= (-1)^{s_y} \cdot m_y \cdot 2^{e_y} \\ z = x \cdot y &= (-1)^{s_z} \cdot m_z \cdot 2^{e_z} \end{aligned}$$

Beobachtung $z = (-1)^{s_x \oplus s_y} \cdot (m_x \cdot m_y) \cdot 2^{e_x + e_y}$

also

1. $s_z := s_x \oplus s_y$ (trivial)
2. $m_z := m_x \cdot m_y$ (Multiplikation von Betragszahlen wie gesehen, implizite Einsen nicht vergessen!)
3. $e_z := e_x + e_y$ (Addition, wegen Exzessdarstellung $e_x + e_y - b$ berechnen)

Beispiel Multiplikation Gleitkommazahlen

x	1	1000	0101	101	0000	0000	0000	0000	0000
y	1	1000	0111	110	1000	0000	0000	0000	0000

Vorzeichen $s_z = 1 \oplus 1 = 0$

Exponent Bias ist $2^{\ell-1} - 1 = (1000\,0000)_2 - 1$

$(1000\,0111)_2 - ((1000\,0000)_2 - 1) = (111)_2 + 1 = (1000)_2$

$(1000\,0101)_2 + (1000)_2 = (1000\,1101)_2$

$(1000\,1101)_2$ ist vorläufiger Exponent

Mantisse

	1,	1	0	1	·	1,	1	1	0	1
	1	1	0	1						
		1	1	0	1					
			1	1	0	1				
				0	0	0	0			
+					1	1	0	1		
1	0,	1	1	1	1	0	0	1		

Normalisieren:

Komma 1 Stelle nach links

Exponent zum Ausgleich +1

implizite Eins streichen

z	0	1000	1110	011	1100	1000	0000	0000	0000

Addition von Gleitkommazahlen

$$\begin{aligned} x &= (-1)^{s_x} \cdot m_x \cdot 2^{e_x} \\ y &= (-1)^{s_y} \cdot m_y \cdot 2^{e_y} \\ z = x + y &= (-1)^{s_z} \cdot m_z \cdot 2^{e_z} \end{aligned}$$

Beobachtung einfach, wenn $e_x = e_y$
dann $m_1 \cdot 2^e + m_2 \cdot 2^e = (m_1 + m_2) \cdot 2^e$

Plan

1. Ergebnis wird „so ähnlich“ wie Zahl mit größerem Exponenten, darum Mantisse der Zahl mit kleinerem Exponenten anpassen
2. Mantissen auf jeden Fall addieren, bei unterschiedlichen Vorzeichen dazu eine Mantisse negieren (Zweierkomplement)
3. anschließend normalisieren

Algorithmus zur Addition

1. Falls $e_x < e_y$, dann x und y komplett vertauschen.
2. Falls Vorzeichen ungleich, dann Vorzeichen von s_y invertieren und Übergang von y zu $-y$ im Zweierkomplement („$\overline{y} + 1$“). $s_z := s_x$
3. Mantisse m_y um $e_x - e_y$ Stellen nach rechts verschieben (Exponenten „virtuell“ jetzt angeglichen)
   Achtung Kann zum “Verlust” signifikanter Stellen führen!
4. $m_z := m_x + m_y$
   Falls $e_x = e_y$, Vorzeichenwechsel möglich. Dann s_z invertieren.
5. $e_z := e_x$. Ergebnis normalisieren

Achtung Bei Mantissen an implizite Einsen denken!

klar Keine separate Subtraktion erforderlich.
Vorzeichenwechsel trivial.
Addition negativer Zahlen enthalten durch Zweierkomplement.

Beispiel Addition Gleitkommazahlen

x	1	1001	0101	111	0010	0000	0000	0000	0000
y	1	1001	0100	110	0001	1000	0000	0000	0000

$e_x > e_y$, Vorzeichen gleich, also zunächst nur $s_z := 1$

Mantisse m_y um $e_x - e_y = 1$ Stelle nach rechts verschieben

$\rightsquigarrow$ 0,1111000011

Mantissen addieren

		1,	1	1	1	0	0	1		
+	0,	1	1	1	0	0	0	0	1	1
1	0,	1	1	0	0	0	1	0	1	1

Normalisieren

- Komma um 1 Stelle nach links verschieben $\rightsquigarrow$ 1,0110001011
- Exponent um 1 vergrößern $\rightsquigarrow$ 1001 0110

z	1	1001	0110	011	0001	0110	0000	0000	0000

Noch ein Beispiel zur Addition

x	1	1000	0101	010	0000	0000	0000	0000	0000
y	0	1000	0100	101	1010	0000	0000	0000	0000

klar $e_x > e_y$ ✓

weil $s_x \neq s_y$ s_y invertieren
Vorzeichenwechsel bei m_y
in Zweierkomplementdarstellung

x	1	1000	0101	1	,	010	0000	0000	0000	0000	0000
aus	0	1000	0100	01	,	101	1010	0000	0000	0000	0000
wird y	1	1000	0100	10	,	010	0110	0000	0000	0000	0000

Noch ein Beispiel zur Addition (2)

x	1	1000	0101	1	,	010	0000	0000	0000	0000	0000
y	1	1000	0100	10	,	010	0110	0000	0000	0000	0000

jetzt e_y an e_x anpassen, m_y verschieben

x	1	1000	0101	1	,	010	0000	0000	0000	0000	0000
y	1	1000	0101	11	,	001	0011	0000	0000	0000	0000
z	1	1000	0101	100	,	011	0011	0000	0000	0000	0000

Erinnerung „überfließende“ 1 einfach ignorieren

z	1	1000	0101	0	,	011	0011	0000	0000	0000	0000

Normalisieren Komma um zwei Stellen nach rechts verschieben
Exponent zum Ausgleich um zwei verkleinern

z	1	1000	0011	100	1100	0000	0000	0000	0000

Fehlerquellen bei der Gleitkommaarithmetik

- Rundung, wenn Berechnungsergebnis zur korrekten Darstellung *mehr* signifikante Bits (d.h. i.d. Mantisse) erfordert als verfügbar
  (kann bei Multiplikation *und* Addition auftreten)
- "Verlust" niederwertiger Bits durch Angleich der Exponenten während der Addition
  Worst Case: $x \gg y$ und $y \neq 0$ aber $x + y = x$

  Und nicht zu vergessen: Darstellung nur einer *extrem kleinen* Auswahl der rationalen Zahlen möglich, variiert mit der Größenordnung der repräsentierten Zahlen!

Probleme bei der Addition: Ein Szenario

Gegeben: Folge von n Gleitkommazahlen $[x_i]$ mit $0 \leq i \leq n$
(z.B. gespeichert in einem Feld/Array `x[i]`)

Aufgabe: Berechne Summe S ... möglichst exakt
(d.h. mit den Möglichkeiten der Gleitkommaarithmetik)

Naive Lösung: Direkte Summation, d.h. berechne:

$$S\uparrow = \sum_{i=0}^{n-1} x_i \quad \text{oder lieber} \quad S\downarrow = \sum_{i=n-1}^{0} x_i$$

~~Theorie/Intuition: Beide Summationen liefern dasselbe Ergebnis!~~

Praxis: $S\uparrow$ und $S\downarrow$ sind i.a. nicht gleich! $\longrightarrow$ Beispielprogramm (in C)

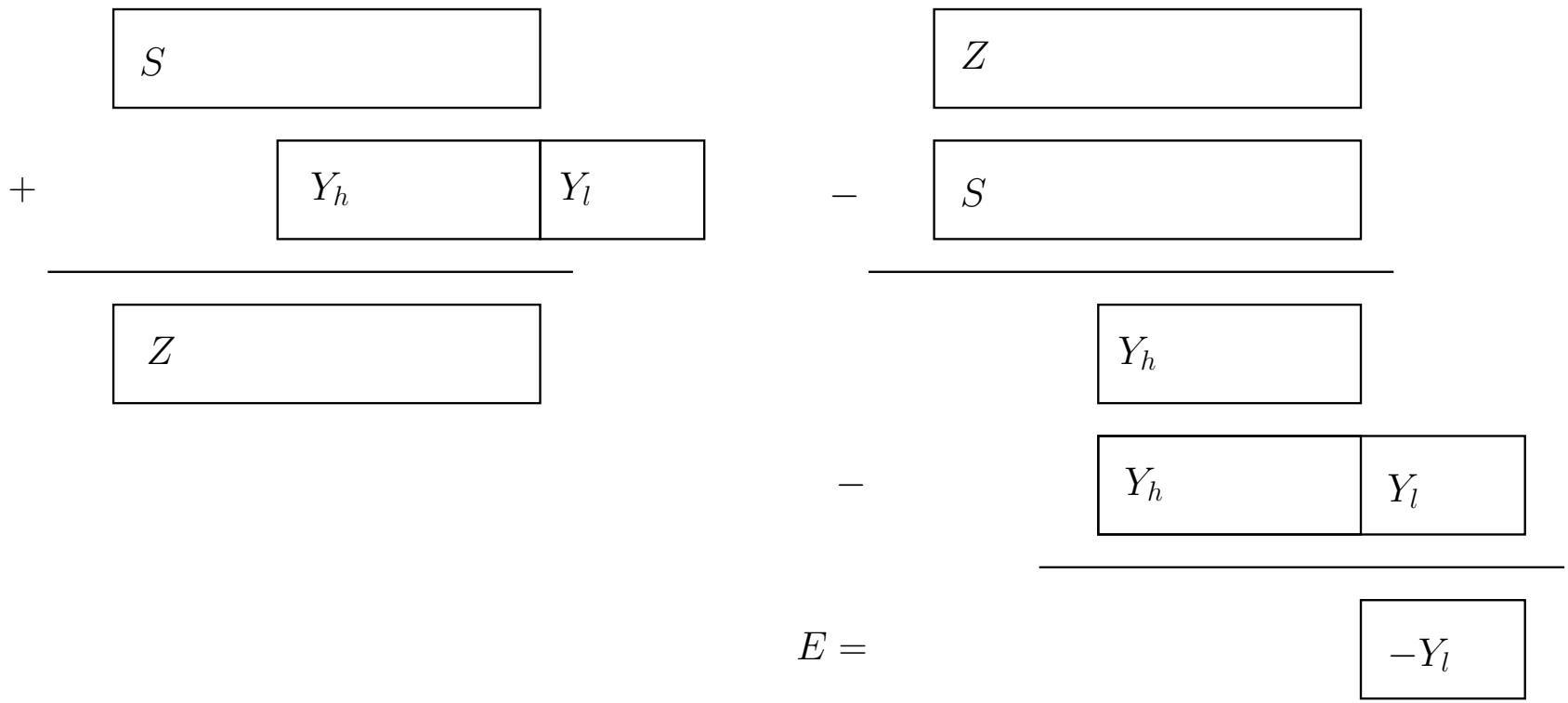
Mögliche Abhilfe: Erhöhung der Genauigkeit (i.d.R. schwierig)
... oder "schlauere" Berechnung ;-)

Fehlerreduktion: Kahan-Summation

Algorithmus zur numerisch stabileren Berechnung von $S = \sum_{i=0}^{n-1} x_i$:

```
S = 0;                          /* Summe */
E = 0;                          /* geschaetzter Fehler */
for i = 0 to n-1 {
        Y = x[i] - E;           /* bish. Fehler beruecksichtigen *
        Z = S + Y;              /* neues Summationsergebnis */
        E = (Z - S) - Y; /* neue Fehlerschaetzung */
        S = Z;
        }
```

⟶ Beispielprogramm (in C)

Fehlerreduktion: Kahan-Summation (2)

Veranschaulichung des fehlerkompensierenden Berechnungsablaufs: