Rechnerstrukturen

Michael Engel und Peter Marwedel

TU Dortmund, Fakultät für Informatik

SS 2013

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

23. April 2013

1. Boolesche Funktionen und Schaltnetze
   - Schaltnetze
   - Rechner-Arithmetik
     - Addition
   - Bessere Schaltnetze zur Addition
     - Carry-Look-Ahead-Addierer
   - Multiplikation
     - Wallace-Tree

Schaltnetze

bis jetzt Diskussion (theoretischer) Grundlagen

Wo bleibt die Hardware?

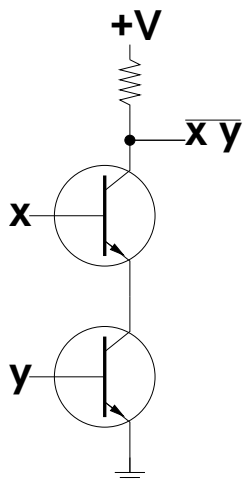
kommt jetzt aber immer noch abstrakt

Wunsch Realisierung boolescher Funktionen in Hardware

klar brauchen Realisierung einer funktional-vollständigen Menge boolescher Funktionen

Erinnerung Realisierung von NAND reicht aus

Beobachtung



realisiert NAND(x, y)

Gatter

Realisierung mit Transistoren . . . falsche Ebene!

Grundlage hier einfache logische Bausteine (Gatter)

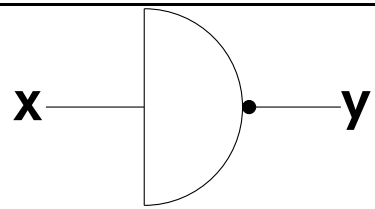
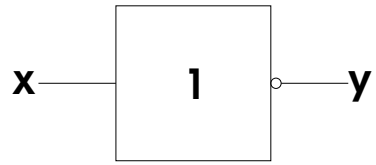
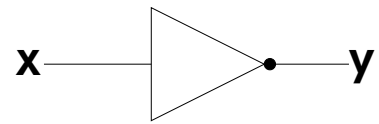
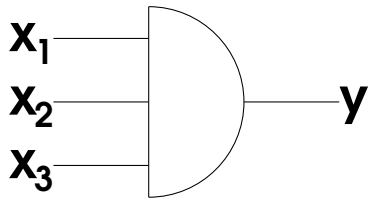
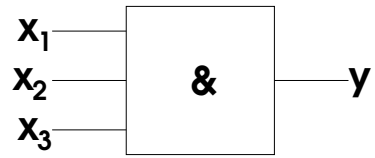
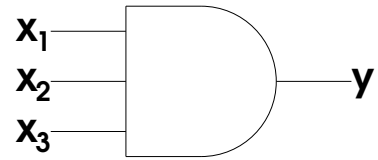
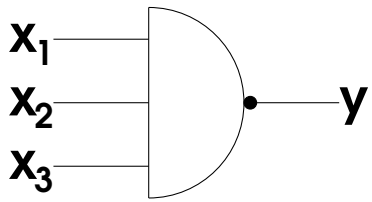
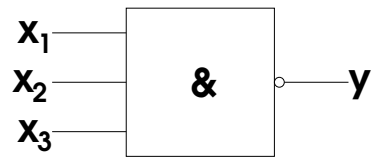
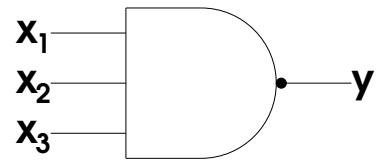
Bausteine für Negation, Konjunktion, Disjunktion, . . .

Spielregeln

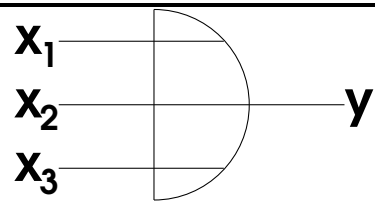
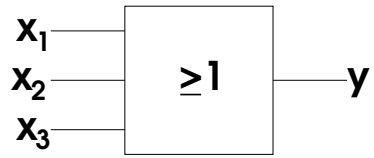
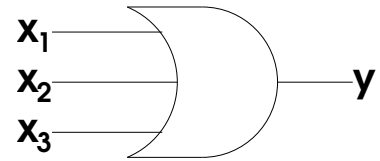
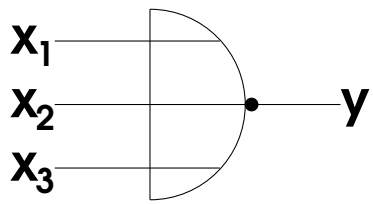
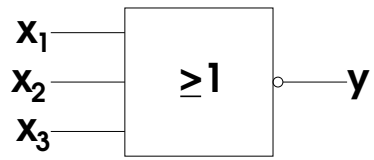
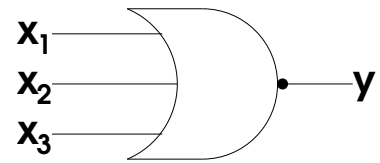
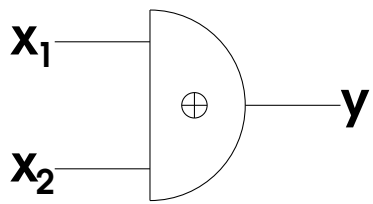
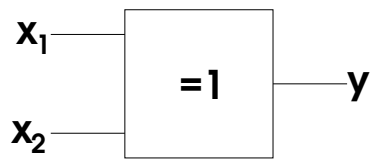
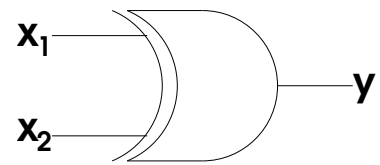
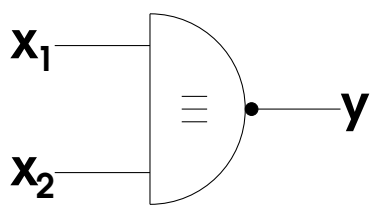
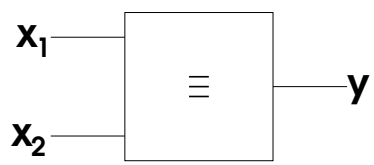
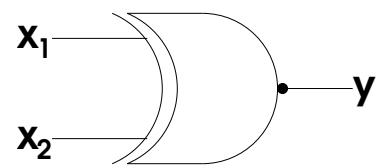
- Eingänge mit Variablen oder Konstanten belegt
- nur Verbindungen von Ausgängen zu Eingängen
- keine Kreise

Ergebnis heißt Schaltnetz

Symbole für Gatter (1)

Funktion	DIN 40700	DIN EN 60617	IEEE
$y = \overline{x}$			
$y = x_1 \wedge x_2 \wedge x_3$			
$y = \overline{x_1 \wedge x_2 \wedge x_3}$			

Symbole für Gatter (2)

Funktion	DIN 40700	DIN EN 60617	IEEE
$y = x_1 \vee x_2 \vee x_3$			
$y = \overline{x_1 \vee x_2 \vee x_3}$			
$y = x_1 \oplus x_2$			
$y = \overline{x_1}\,\overline{x_2} \vee x_1\,x_2$			

Schaltnetz-Bewertung

Und jetzt beliebige Schaltnetze entwerfen?

mindestens relevant Größe und Geschwindigkeit

- Schaltnetzgröße (= Anzahl der Gatter) wegen Kosten, Stromverbrauch, Verlustleistung, Zuverlässigkeit, ...
- Schaltnetztiefe (= Länge längster Weg Eingang $\rightsquigarrow$ Ausgang) wegen Schaltgeschwindigkeit
- Fan-In (= max. Anzahl eingehender Kanten) wegen Realisierungsaufwand
- Fan-Out (= max. Anzahl ausgehender Kanten) wegen Realisierungsaufwand
- ... (z. B. Anzahl Gattertypen, Testbarkeit, Verifizierbarkeit)

Was wir schon wissen

Jede boolesche Funktion kann mit einem $\{\wedge, \vee, \neg\}$- bzw. einem $\{\oplus, \wedge, \neg\}$-Schaltnetz der Tiefe 3 realisiert werden.

Beweis DNF, KNF oder RNF direkt umsetzen □

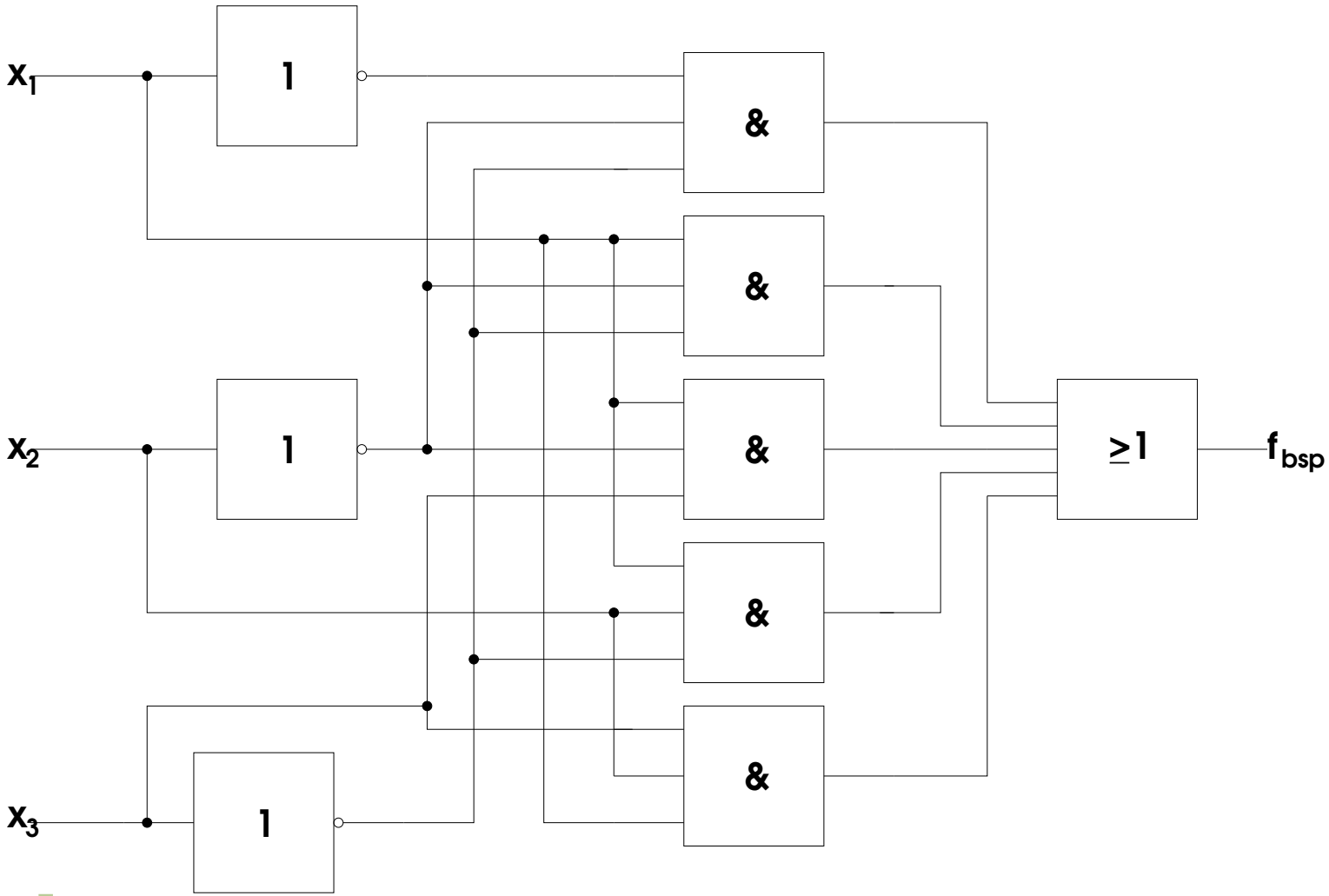
Probleme

- Fan-In des tiefsten Gatters kann extrem groß sein
- Größe des Schaltnetzes oft inakzeptabel

Beispiel: DNF

$f_{\text{bsp}}\colon B^3 \to B$, Wertevektor $(1,0,0,0,1,1,1,1)$

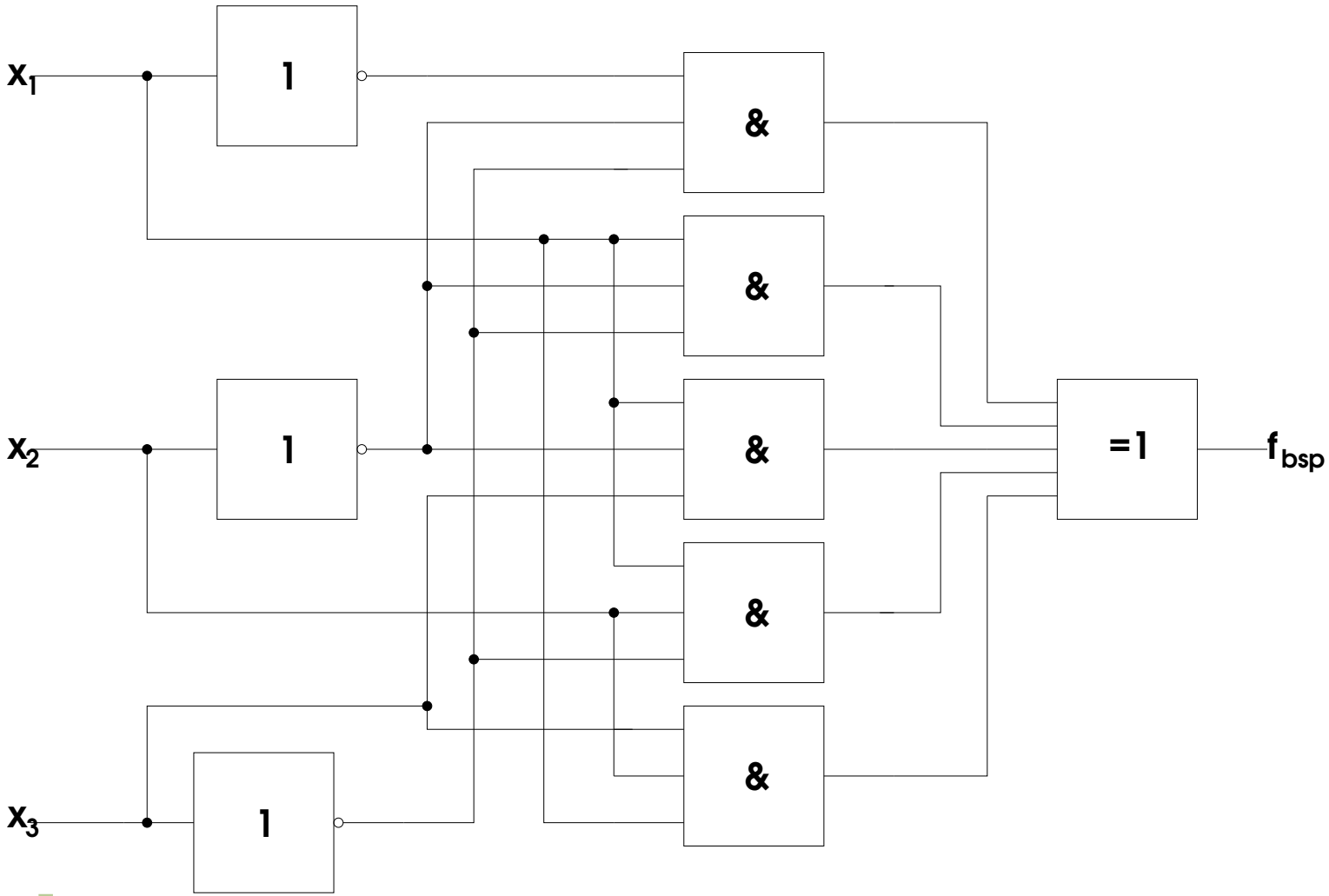
$f(x_1,x_2,x_3) = \overline{x_1}\,\overline{x_2}\,\overline{x_3} \vee x_1\,\overline{x_2}\,\overline{x_3} \vee x_1\,\overline{x_2}\,x_3 \vee x_1\,x_2\,\overline{x_3} \vee x_1\,x_2\,x_3$



Beispiel: RNF

$f_{\text{bsp}}\colon B^3 \to B$, Wertevektor $(1,0,0,0,1,1,1,1)$

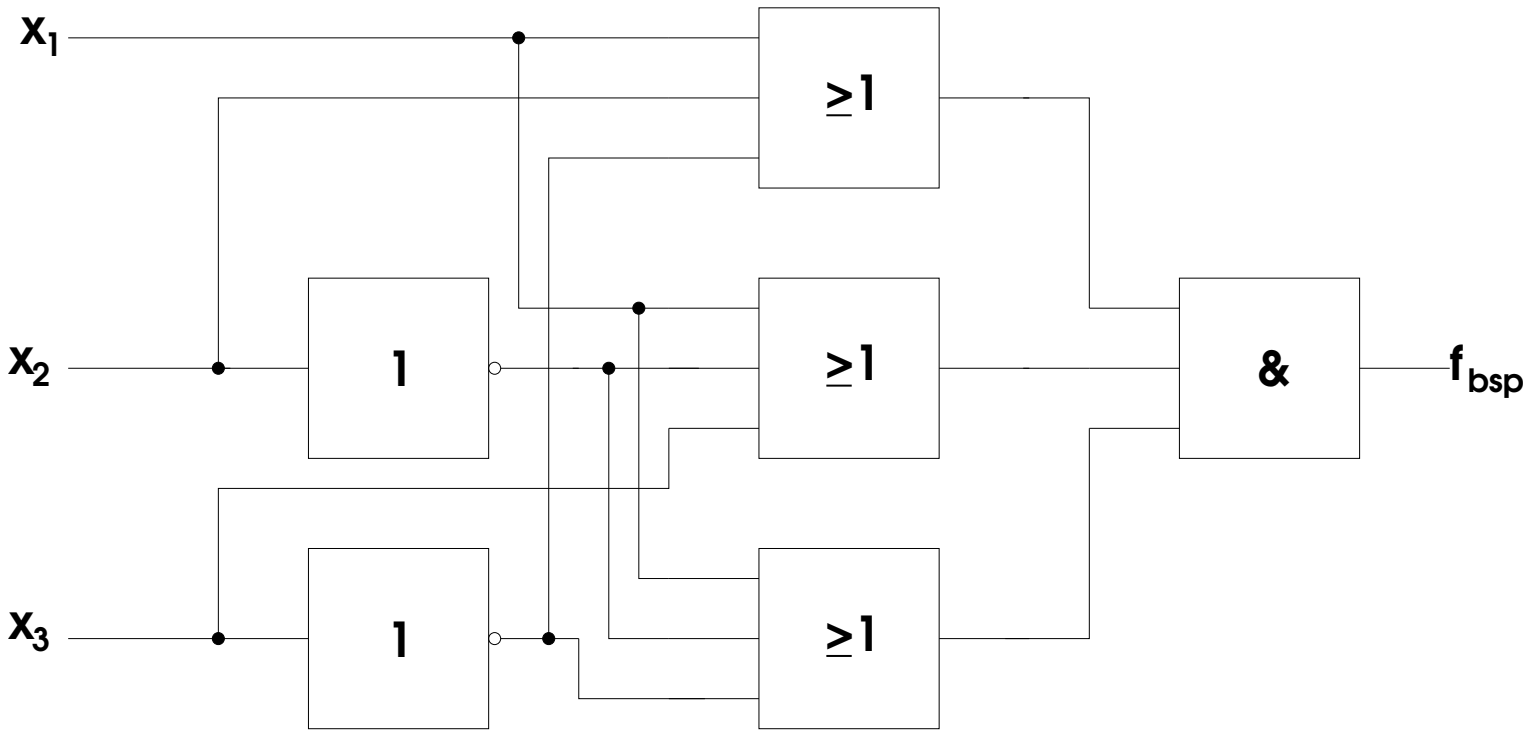
$f(x_1,x_2,x_3) = \overline{x_1}\,\overline{x_2}\,\overline{x_3} \oplus x_1\,\overline{x_2}\,\overline{x_3} \oplus x_1\,\overline{x_2}\,x_3 \oplus x_1\,x_2\,\overline{x_3} \oplus x_1\,x_2\,x_3$



Beispiel: KNF

$f_{\text{bsp}}\colon B^3 \to B$, Wertevektor $(1,0,0,0,1,1,1,1)$

$f(x_1,x_2,x_3) = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3})$



Multiplexer

eine weitere Beispielfunktion ...

aber eine in der Praxis sehr wichtige diesmal!

$\mathrm{MUX}_d\left(y_1, y_2, \ldots, y_d, x_0, x_1, \ldots, x_{2^d-1}\right) = x_{(y_1\, y_2 \cdots y_d)_2}$

Beispiel

y_1	y_2	y_3	$\mathrm{MUX}_3(y_1, y_2, y_3, x_0, x_1, \ldots, x_7)$
0	0	0	x_0
0	0	1	x_1
0	1	0	x_2
0	1	1	x_3
1	0	0	x_4
1	0	1	x_5
1	1	0	x_6
1	1	1	x_7

Multiplexer

Warum ist MUX in der Praxis wichtig?

direkte Speicheradressierung

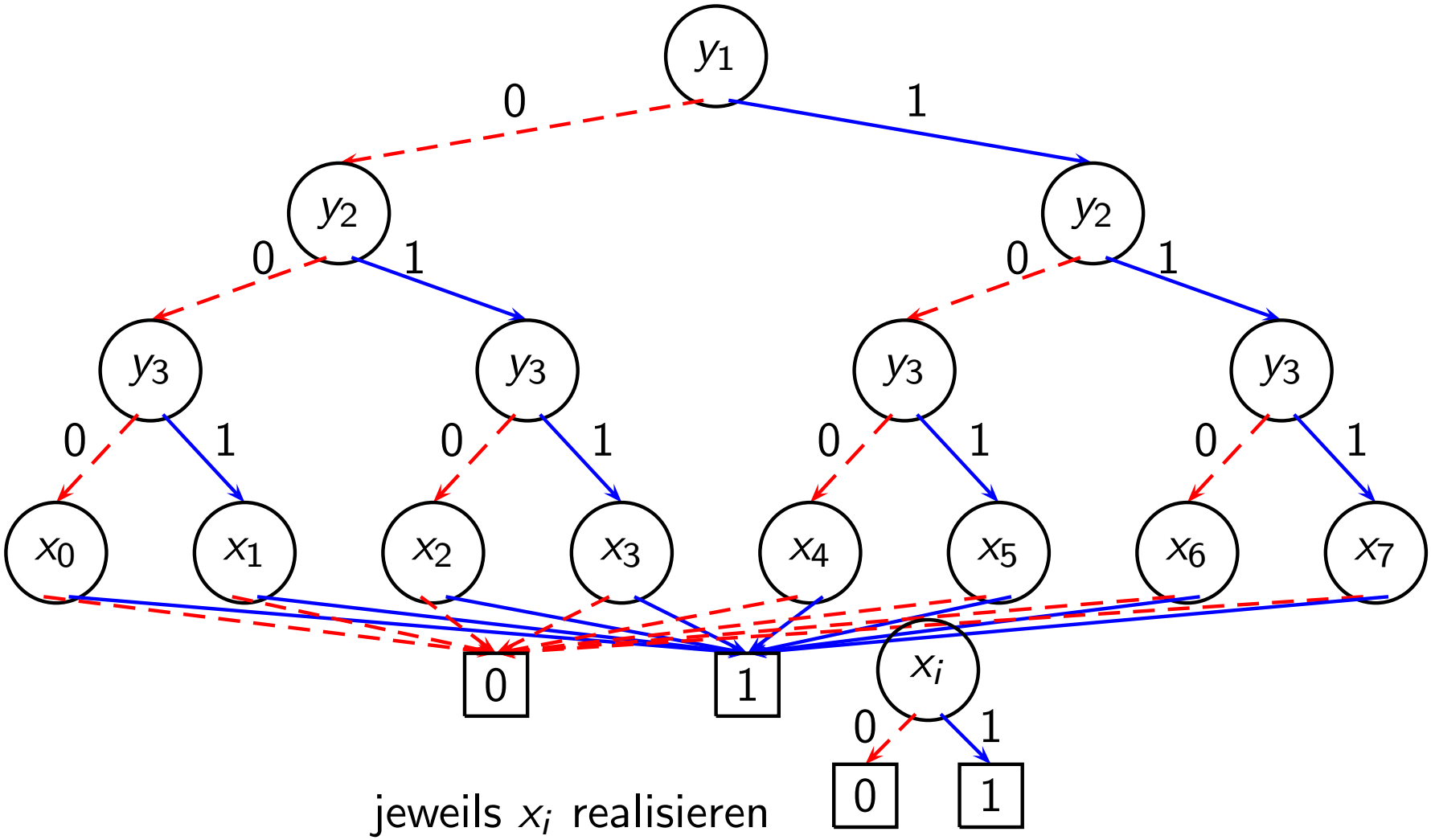
OBDD-Realisierung

Welche Variablenordnung π?

wohl sinnvoll	Adressvariablen zuerst
denn	Wir müssen uns sonst alle Datenvariablen merken!

OBDD-Realisierung MUX_3

Variablenordnung $\pi = (y_1, y_2, y_3, x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7)$



jeweils x_i realisieren

Einfluss von π auf OBDD-Größe

Beispiel MUX_d

gesehen Variablenordnung $\pi = (y_1, y_2, \ldots, y_d, x_0, x_1, \ldots, x_{2^d-1})$

Größe des reduzierten πOBDDs?

oben vollständiger Binärbaum über $y_1, y_2, \ldots, y_d$
$2^0 + 2^1 + \cdots + 2^{d-1} = 2^d - 1$ Knoten

unten je ein x_i-Knoten und zwei Senken
$2^d + 2$ Knoten

zusammen $2^d - 1 + 2^d + 2 = 2^{d+1} + 1$ Knoten

Einfluss von π auf OBDD-Größe

Beispiel MUX_d

gesehen Variablenordnung $\pi = (y_1, y_2, \ldots, y_d, x_0, x_1, \ldots, x_{2^d-1})$
Größe des reduzierten πOBDD $2^{d+1} + 1$

Betrachte Variablenordnung
$\pi = (x_0, x_1, \ldots, x_{2^d-1}, y_1, y_2, \ldots, y_d)$

Größe des reduzierten πOBDDs?

Behauptung $\forall x \neq x' \in \{0,1\}^{2^d}$: nach Lesen von x bzw. x' verschiedene Knoten erreicht

Beweis durch Widerspruch

Widerspruchsbeweis zur OBDD-Größe

Behauptung $\forall x \neq x' \in \{0,1\}^{2^d}$: nach Lesen von x bzw. x' verschiedene Knoten erreicht

Annahme für $x \neq x' \in \{0,1\}^{2^d}$ gleiche Knoten v erreicht

Betrachte $i = \min\{j \mid x_j \neq x'_j\}$

Betrachte $y_1, y_2, \ldots, y_d$ mit $(y_1 y_2 \cdots y_d)_2 = i$

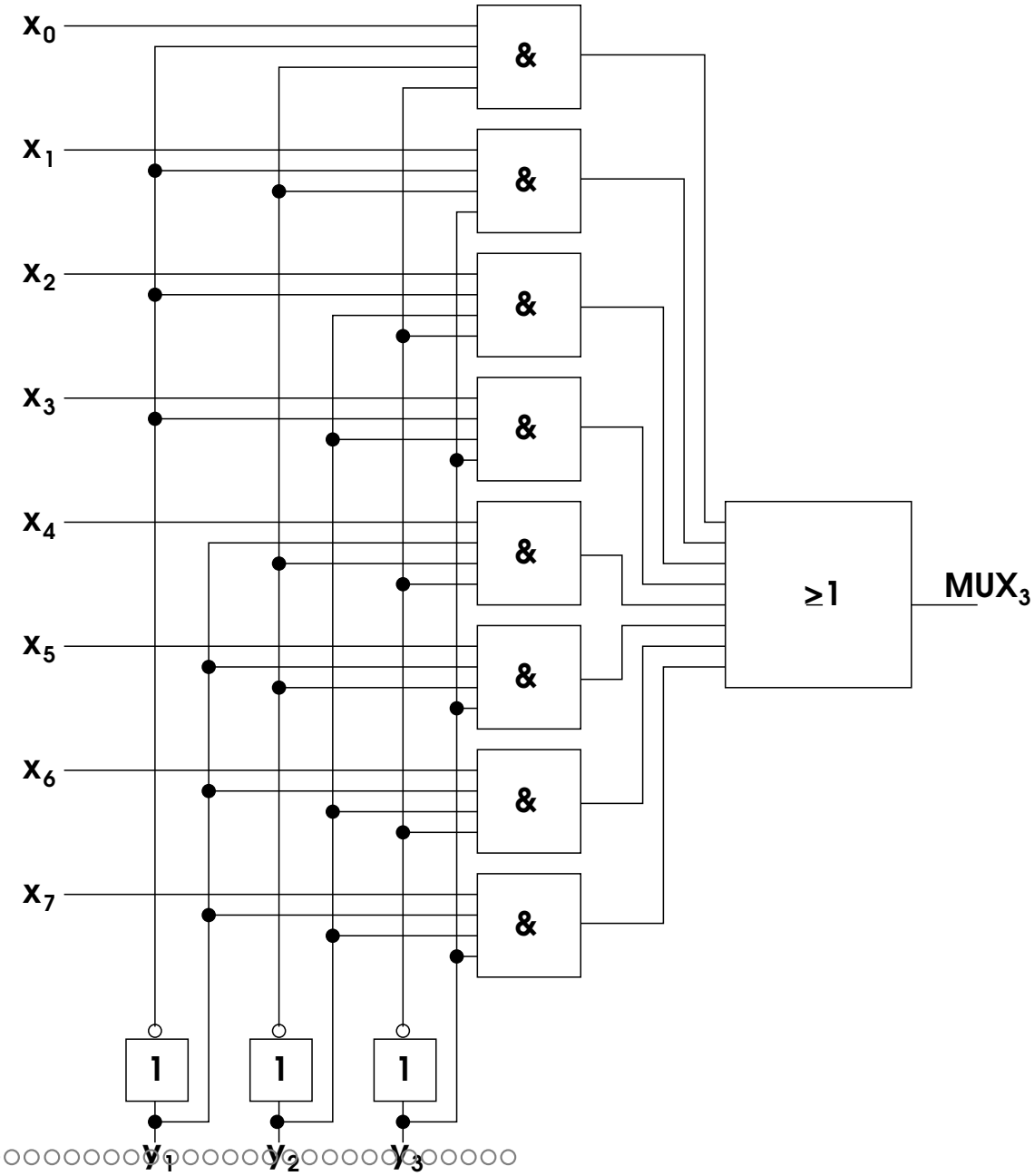
Beobachtung $\mathrm{MUX}_d(y_1, y_2, \ldots, y_d, x) = x_i$
$\neq x'_i = \mathrm{MUX}_d(y_1, y_2, \ldots, y_d, x')$

aber OBDD berechnet gleichen Wert, da gleicher Knoten erreicht

also minimales OBDD für $\pi = (x_0, x_1, \ldots, x_{2^d-1}, y_1, y_2, \ldots, y_d)$ hat Größe mindestens $2^{2^d} - 1$ (Binärbaum d. Tiefe 2^d)

d	min. OBDD-Größe $\pi = (d, x)$	min. OBDD-Größe $\pi = (x, d)$
2	9	≥ 15
4	33	$\geq 65\,535$
8	513	$\geq 1{,}1579 \cdot 10^{77}$

Schaltnetz für MUX_3



Rechner-Arithmetik

klar Rechnen zu können ist für Computer zentral.

Was wollen wir hier rechnen?

hier nur Grundrechenarten, aber keine Division, also

- Addition
- Subtraktion
- Multiplikation

Womit wollen wir hier rechnen?

- mit ganzen Zahlen
- mit rationalen Zahlen (IEEE 754-1985)

Addition

Wie haben wir das in der Schule gelernt?

1. Summand			9	3	8	9	9	8	9
2. Summand				9	7	9	8	9	8
Übertrag	+	1	1	1	1	1	1	1	
Summe		1	0	3	6	9	8	8	7

Übertragung auf Binärzahlen

1. Summand			1	1	0	0	1	1	1
2. Summand			1	0	0	1	0	1	1
Übertrag	+	1			1	1	1	1	
Summe		1	0	1	1	0	0	1	0

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

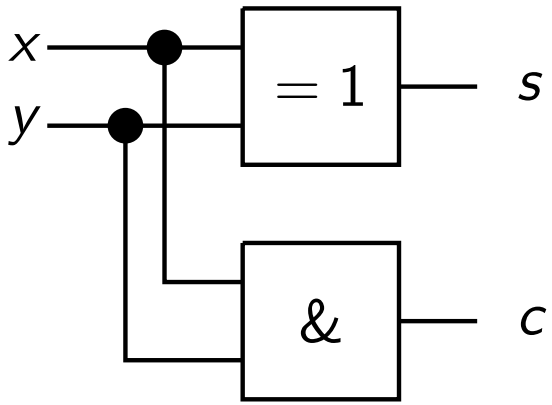
x	y	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

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

Bessere Realisierung Schaltnetz Volladdierer

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

Ziel für c 1 realisieren
für $(c_{\text{alt}}, x, y) \in \{011, 101, 110, 111\}$

Beobachtung $x \oplus y = 1 \Leftrightarrow$ genau eine 1 in x, y
also $c_{\text{alt}}\,(x \oplus y)$ realisiert 101, 110

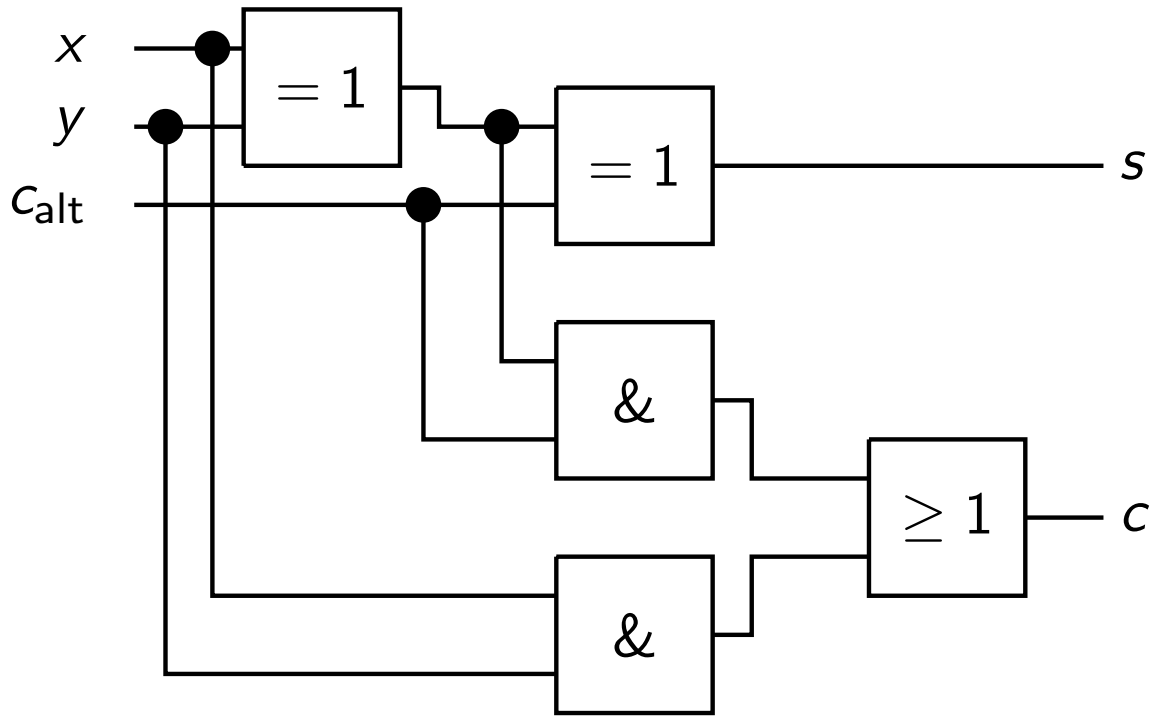
fehlt noch 011, 111
beides durch $x\,y$

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

Schaltnetz Volladdierer

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

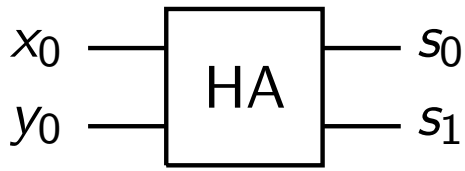
$c = c_{\text{alt}} (x \oplus y) \vee x\, y$



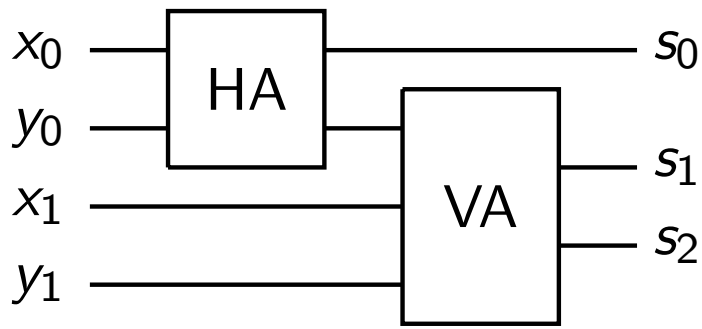
Größe 5, Tiefe 3

Vollständiger Addierer

$$\begin{array}{cccc} & & x_0 & \\ + & & y_0 & \\ \hline & s_1 & s_0 & \end{array}$$

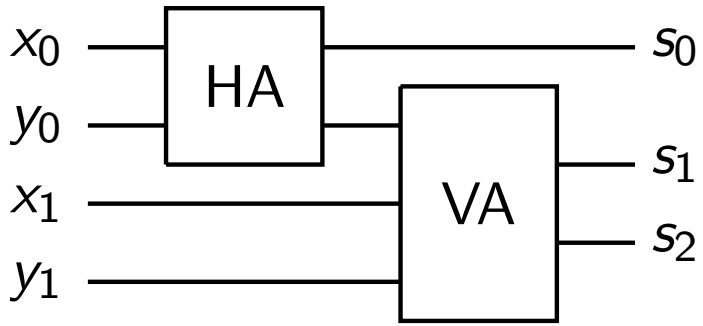


$$\begin{array}{cccc} & & x_1 & x_0 \\ + & & y_1 & y_0 \\ \hline & s_2 & s_1 & s_0 \end{array}$$

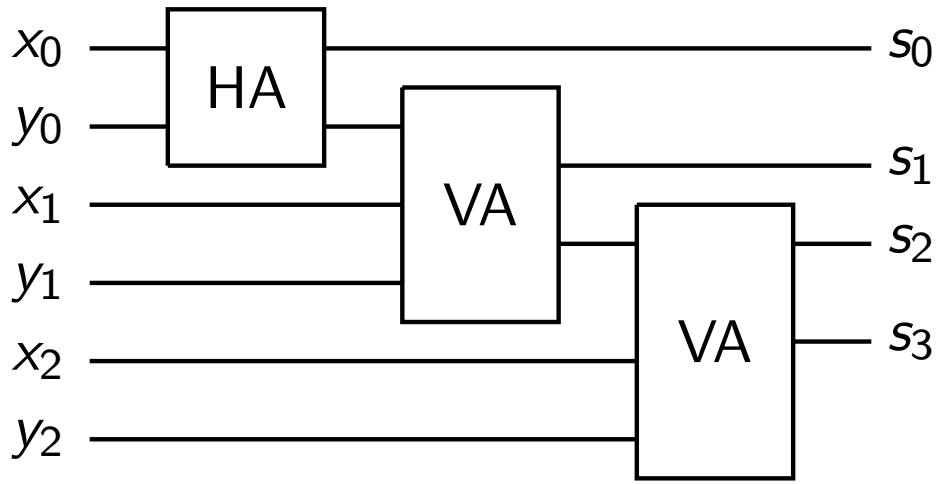


Vollständiger Addierer

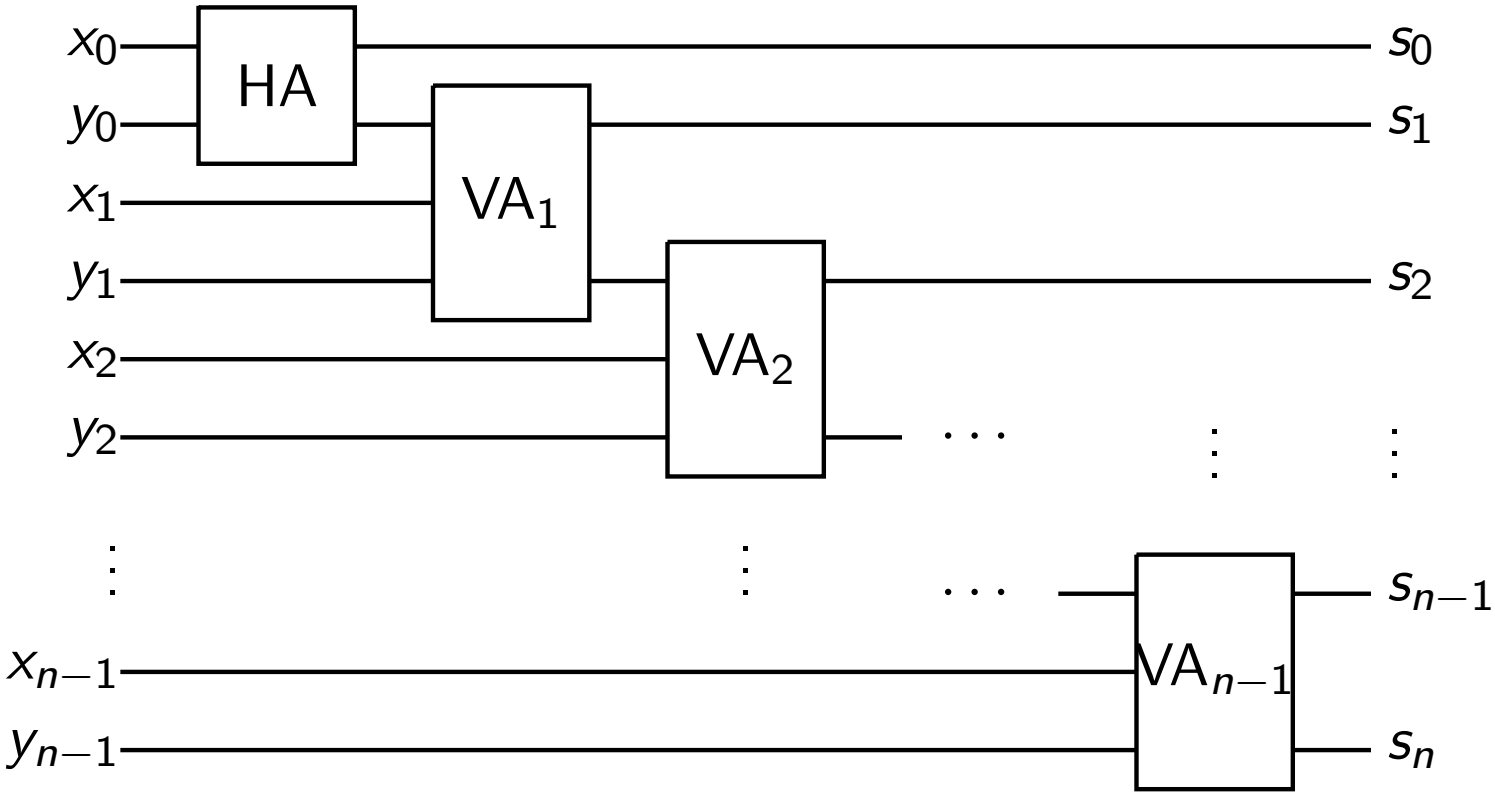
$$
\begin{array}{cccc}
 & & x_1 & x_0 \\
+ & & y_1 & y_0 \\
\hline
 & s_2 & s_1 & s_0
\end{array}
$$



$$
\begin{array}{ccccc}
 & & x_2 & x_1 & x_0 \\
+ & & y_2 & y_1 & y_0 \\
\hline
 & s_3 & s_2 & s_1 & s_0
\end{array}
$$



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

Ausnutzen von Einsichten

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

Beobachtung sehr gut durch Halbaddierer realisierbar

x_i	y_i	u_i	v_i	Übertrag
0	0	0	0	eliminieren
0	1	0	1	weiterreichen
1	0	0	1	weiterreichen
1	1	1	0	generieren

also $u_i = 1$ generiert Übertrag

$v_i = 1$ reicht Übertrag weiter

zentrale Beobachtung alle HA in Tiefe 1 parallel realisierbar

Realisierung als Schaltnetz

Notation $f_{\mathrm{HA}}(x_i, y_i) = (u_i, v_i)$

$u_i = 1$ generiert Übertrag c_{i+1}

$v_i = 1$ reicht Übertrag c_{i+1} weiter

$$
\begin{aligned}
c_i &= u_{i-1} \vee u_{i-2}\, v_{i-1} \vee u_{i-3}\, v_{i-2}\, v_{i-1} \vee \cdots \vee u_0\, v_1\, v_2 \cdots v_{i-1} \\
&= \bigvee_{j=0}^{i-1} \left(u_j \wedge \bigwedge_{k=j+1}^{i-1} v_k \right)
\end{aligned}
$$

Gesamttiefe: 4

alle Halbaddierer (u_i, v_i)	Tiefe 1
alle Und-Gatter $u_j \wedge \bigwedge_{k=j+1}^{i-1} v_k$	Tiefe 1
großes Oder-Gatter	Tiefe 1
$n\times$ $\oplus$-Gatter f. korr. Summenbits	Tiefe 1

Realisierung als Schaltnetz

Mit dem Ansatz

$u_i = 1$ generiert Übertrag c_{i+1}

$v_i = 1$ reicht Übertrag c_{i+1} weiter

$$
\begin{aligned}
c_i &= u_{i-1} \vee u_{i-2}\, v_{i-1} \vee u_{i-3}\, v_{i-2}\, v_{i-1} \vee \cdots \vee u_0\, v_1\, v_2 \cdots v_{i-1} \\
&= \bigvee_{j=0}^{i-1} \left(u_j \wedge \bigwedge_{k=j+1}^{i-1} v_k \right)
\end{aligned}
$$

beliebig lange Zahlen in Tiefe 4 addieren...

Ist das fair gezählt? **Nein!**

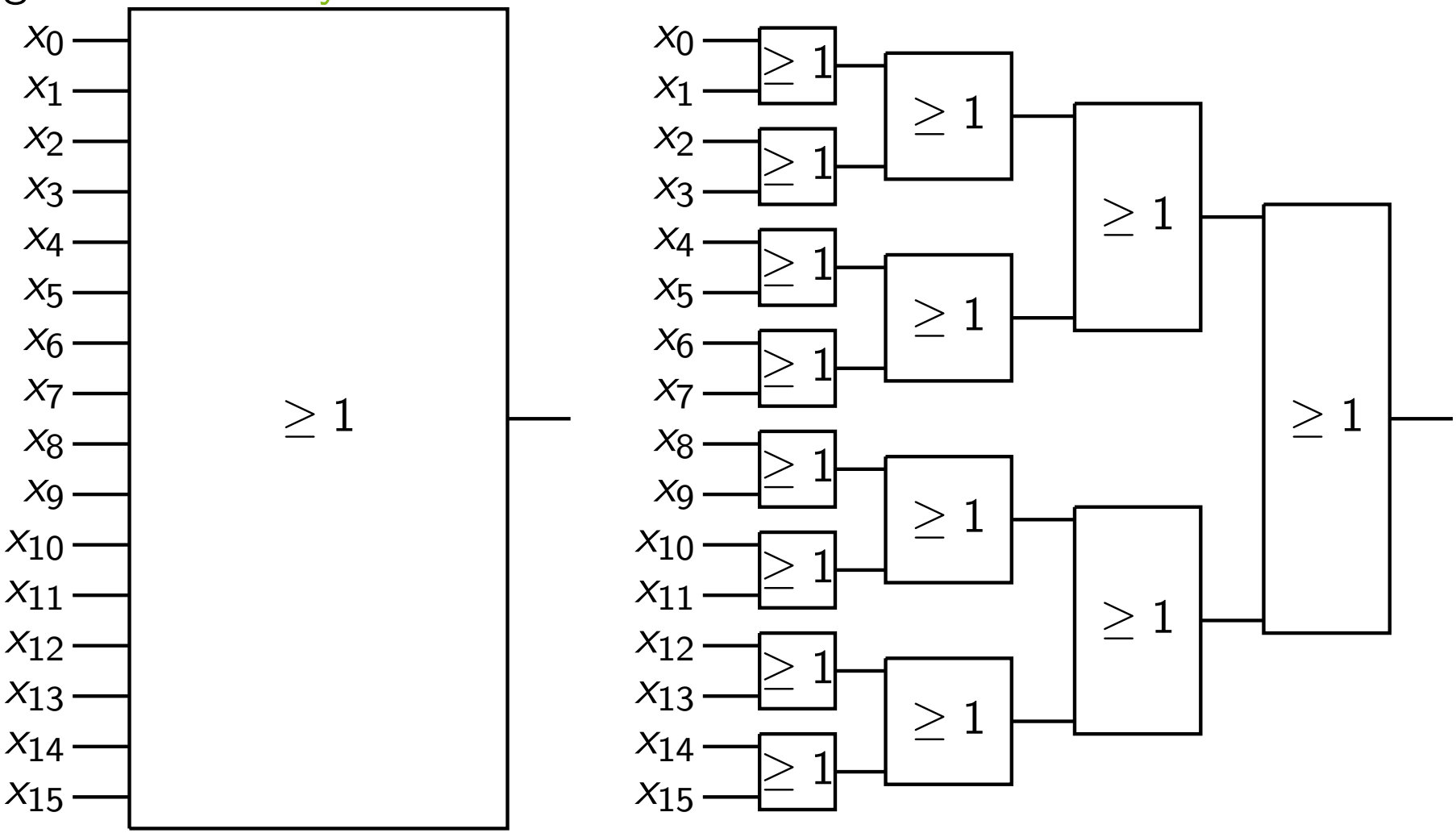
Begriff Anzahl Eingänge eines Gatters heißt Fan-In

Problem Wir vergleichen Schaltnetz mit max. Fan-In 2
mit Schaltnetz mit max. Fan-In n.

Ziel Vergleichbarkeit herstellen

Vermeidung von großem Fan-In

großen Fan-In systematisch verkleinern



Beispiel ODER, Fan-In 16

Vermeidung von großem Fan-In

am Beispiel

aus	Fan-In 16	bei Tiefe 1	(Größe 1)
wird	Fan-In 2	bei Tiefe 4	bei Größe 15

Wie ist das allgemein?

Größe

bei jeder Stufe nur noch halb so viele Gatter

$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 1 \approx n$

Tiefe

so oft halbieren, bis man bei 1 Gatter ist

$\approx \log_2 n$

Carry Look-Ahead Addierer (CLA)

Realisierung „Addition von natürlichen Zahlen“

- gut strukturiert
- Größe $\approx n^2$ $\leftarrow$ ziemlich groß
- Tiefe $\approx 2 \log_2 n$ $\leftarrow$ **ziemlich** flach

Haben wir jetzt im Vergleich zum *Ripple-Carry-Addierer* gewonnen?

n	Ripple-Carry Größe	Ripple-Carry Tiefe	Carry Look-Ahead Größe	Carry Look-Ahead Tiefe
8	37	22	64	6
16	77	46	256	8
32	157	94	1 024	10
64	317	190	4 096	12

Noch einmal nachdenken. . .

Wir wissen doch schon

Jede boolesche Funktion in Tiefe 3 realisierbar!
(DNF, KNF, RNF)

Wieso nicht auf die Addition?

- klar geht in Tiefe 3
- aber Größe inakzeptabel (exponentiell)

Fazit vorhandenes Vorwissen ausnutzen,
Normalformen nur bei kleinen oder unstrukturierten Funktionen

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

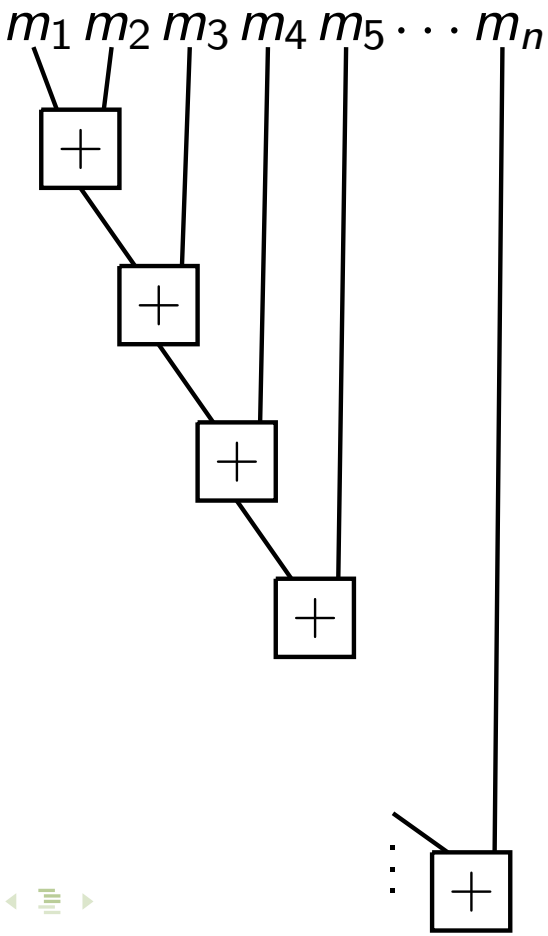
klar Nullen schreiben einfach und kostenlos,
Zahlen verschieben und kopieren einfach und kostenlos,
viele Zahlen addieren nicht ganz so einfach

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!

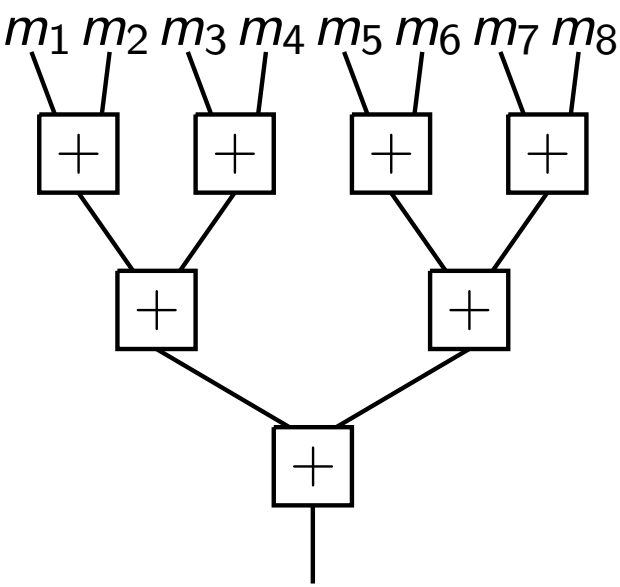
klar sehr naiv

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

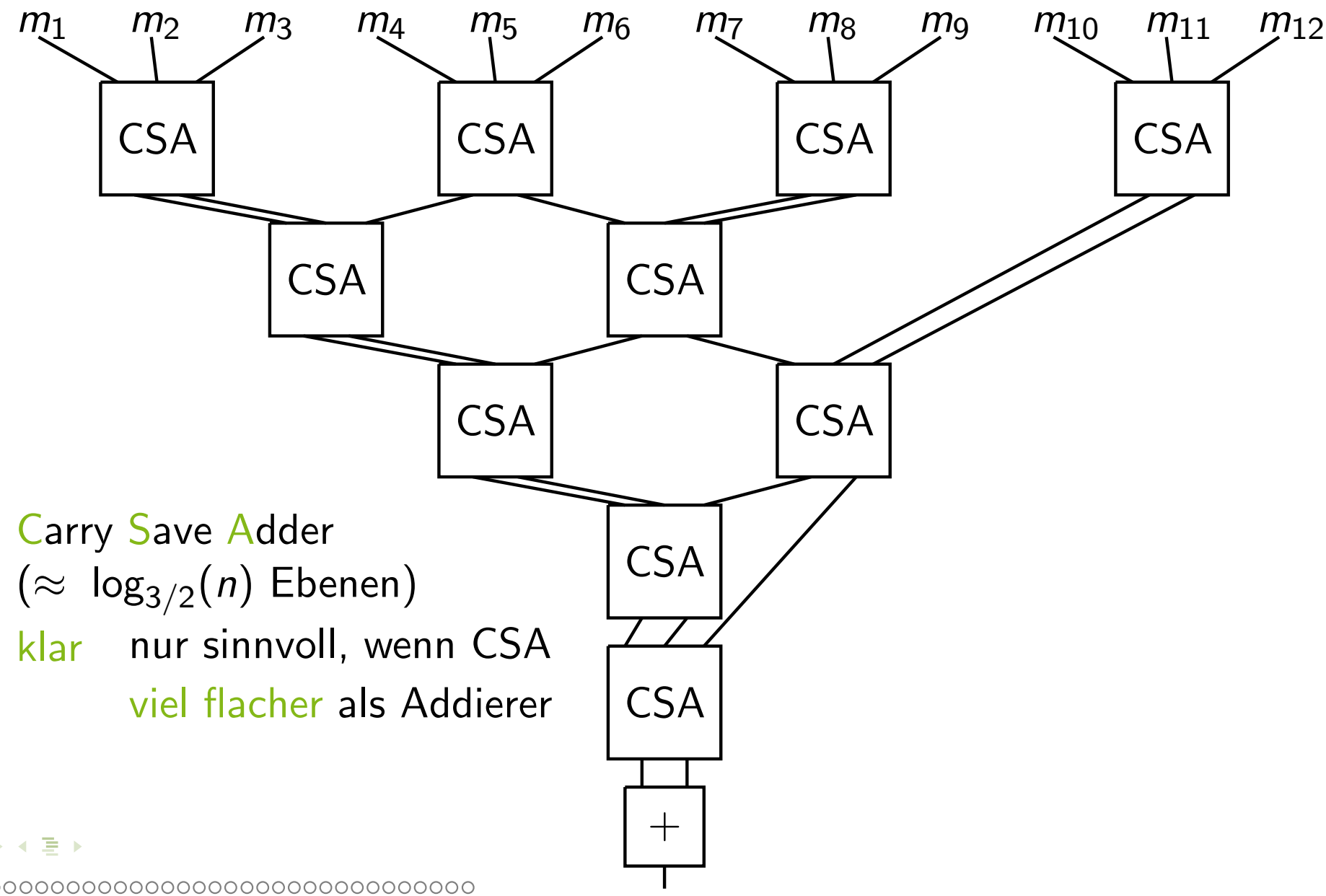
- gleiche Summe $\rightsquigarrow$ Korrektheit
- weniger Zahlen $\rightsquigarrow$ „Fortschritt“

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

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