

2. Hardwareentwurf -- die Disziplinen

Zur Vorlesung Rechenanlagen

SS 2019



2.1 Entwurf kombinatorischer Module

Zur Vorlesung Rechenanlagen

SS 2019



2.1.1 Kombinatorische Schaltkreise

Definition

Ein Schaltkreis C heißt **kombinatorisch**, genau dann, wenn

- die Bausteine in C kombinatorisch sind,
- C wohlgeformt und
- C rückkopplungsfrei ist.

Diese Untermenge der Menge aller Schaltkreise ist sehr wichtig, weil man durch sie schon alle Schaltfunktionen realisieren und ihr Verhalten sehr leicht getrennt unter rein statischen, funktionellen Aspekten und rein zeitlichen Aspekten analysieren kann (wenn auch nicht sehr exakt!).

Kombinatorische Schaltkreise ff

Definition

Sei C ein kombinatorischer Schaltkreis. Dann ordnen wir jedem Signal s , dessen lokale Funktion sich aus den Signalen s_{i_1}, \dots, s_{i_k} bestimmt, eine Schaltfunktion $F[s]$ über den Primäreingängen,

$$F[s] := C[s](s_{i_1} = F[s_{i_1}], \dots, s_{i_k} = F[s_{i_k}])$$

die **globale Funktion** von s zu, wobei für einen Primäreingang s zu $C.x$ $F[s] := x$ gilt.

Wir können damit jedes Signal in einem kombinatorischen Schaltkreis auch als Funktion über den Eingängen auffassen. Ferner sortieren wir von nun an die $m = \#N$ Signale s_1, \dots, s_m stets **topologisch**, d.h. s_1, \dots, s_n sind Primäreingänge und sonst ist $\text{tiefe}(s_i) \leq \text{tiefe}(s_j)$ für $i < j$

Kombinatorische Schaltkreise ff

Satz

Sei C ein kombinatorischer Schaltkreis. Dann gibt es zu jeder Belegung p_1, \dots, p_n der Primäreingänge genau eine stabile Belegung p_1, \dots, p_m mit

$$p_i = F[s_i](p_1, \dots, p_n)$$

Beweis:

Wir zeigen durch Induktion nach der Tiefe, dass für jedes Signal s_i in einer stabilen Belegung p gilt:

$$p_i = F[s_i](p_1, \dots, p_n)$$

$\text{tiefe}(s_i) = 0$: Dann ist s_i Primäreingang und damit

$$F[s_i](p_1, \dots, p_n) = x_i(p_1, \dots, p_n) = p_i$$

Beweis ff

$\text{tiefe}(s_i) > 0$: Sei für $k < \text{tiefe}(s_i)$ die Behauptung schon gezeigt. Dann ist p stabil, nur wenn

$$1 = (p_i \equiv C[s_i](p_{i_1}, \dots, p_{i_k}))$$

$$\Leftrightarrow$$

$$p_i = C[s_i](s_{i_1} = p_{i_1}, \dots, s_{i_k} = p_{i_k})$$

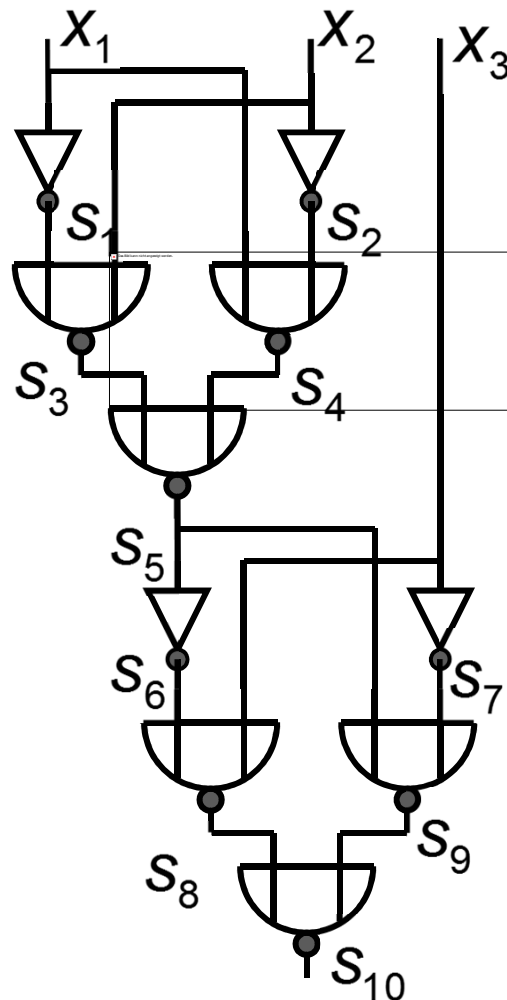
und mit I.A. $p_{i_j} = F[s_{i_j}](p_1, \dots, p_n)$

$$\Leftrightarrow$$

$$\begin{aligned} p_i &= C[s_i](s_{i_1} = F[s_{i_1}], \dots, s_{i_k} = F[s_{i_k}])(p_1, \dots, p_n) \\ &= F[s_i](p_1, \dots, p_n) \end{aligned}$$

Beispiel:

Man kann also jedem Signal durch sukzessive Substitution eine Funktion über den Primäreingängen zuordnen:



$$F[s_1] = C[s_1](x_1 = F[x_1]) = \overline{x_1}, \quad F[s_2] = \overline{x_2}$$

$$F[s_3] = C[s_3](s_1 = F[s_1], x_2 = F[x_2])$$

$$= \overline{F[s_1] \vee F[x_2]} = \overline{\overline{x_1} \vee x_2}$$

$$F[s_4] = \overline{x_1 \vee \overline{x_2}}$$

$$F[s_5] = \overline{\overline{\overline{x_1 \vee x_2 \vee x_1 \vee \overline{x_2}}}}$$

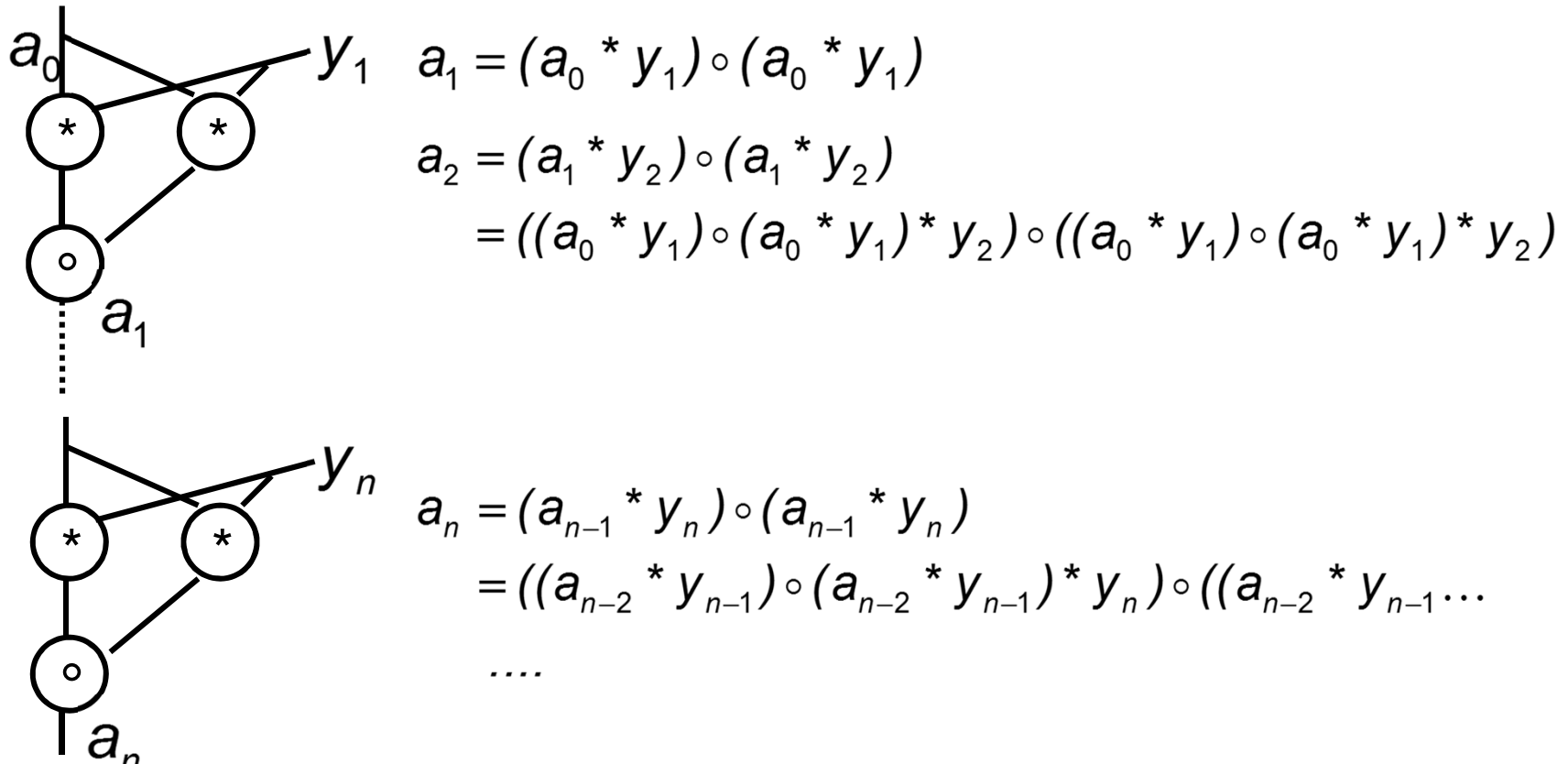
$$F[s_6] = \overline{\overline{\overline{x_1 \vee x_2 \vee x_1 \vee \overline{x_2}}}}, \quad F[s_7] = \overline{x_3}$$

$$F[s_8] = \overline{\overline{\overline{\overline{\overline{x_1 \vee x_2 \vee x_1 \vee \overline{x_2} \vee x_3}}}}}$$

$$F[s_9] = \overline{\overline{\overline{\overline{\overline{\overline{x_1 \vee x_2 \vee x_1 \vee \overline{x_2} \vee x_3}}}}}}}$$

Beispiel ff:

Vereinfacht man die Ausdrücke nicht, dann können sie enorm wachsen: Wann immer ein Signal s zur Berechnung eines Signals t mehrfach benutzt wird, tritt der Ausdruck für $F[s]$ zunächst mehrfach im Ausdruck für $F[t]$ auf:



Beispiel ff:

$$\begin{aligned} \text{D.h. } |a_n| &= 2|a_{n-1}| + 9 = \dots = 2^n |a_0| + 9 \sum_{i=0}^{n-1} 2^i \\ &= 2^n + 9 \cdot (2^n - 1) = 5 \cdot 2^{n+1} - 9 \end{aligned}$$

Der Ausdruck wächst exponentiell in der Größe der Schaltung!

Selbst wenn man im Beispiel $F[s_{10}]$ vereinfacht, erhält man

$$\begin{aligned} F[s_{10}] &= \overline{\overline{\overline{\overline{\overline{x_1} \vee x_2 \vee x_1 \vee \overline{x_2} \vee x_3} \vee \overline{\overline{\overline{\overline{\overline{x_1} \vee x_2 \vee x_1 \vee \overline{x_2} \vee \overline{x_3}}}}}}}}}} \\ &\stackrel{(\text{De Morgan})}{=} (\overline{\overline{\overline{\overline{\overline{x_1} \vee x_2 \vee x_1 \vee \overline{x_2} \vee x_3}}}}) \cdot (\overline{\overline{\overline{\overline{\overline{x_1} \vee x_2 \vee x_1 \vee \overline{x_2} \vee \overline{x_3}}}}}) \\ &\stackrel{(\text{De Morgan})}{=} (\overline{x_1 \overline{x_2} \vee \overline{x_1} x_2 \vee x_3}) \cdot ((\overline{x_1} \vee x_2)(x_1 \vee \overline{x_2}) \vee \overline{x_3}) \\ &\stackrel{(\text{ausmult.})}{=} (\overline{x_1 \overline{x_2} \vee \overline{x_1} x_2 \vee x_3}) \cdot (\overline{x_1} \overline{x_2} \vee x_1 x_2 \vee \overline{x_3}) \\ &= \overline{x_3} (\overline{x_1 \overline{x_2} \vee \overline{x_1} x_2}) \vee x_3 (\overline{x_1} \overline{x_2} \vee x_1 x_2) \end{aligned}$$

Zusammenfassung:

Wir wissen nun

Satz

Jede Schaltfunktion $f \in \mathbf{S}_n$ ist durch einen booleschen Ausdruck über x_1, \dots, x_n darstellbar, wobei x_1, \dots, x_n die Projektionen in \mathbf{S}_n sind.

Satz

Jede Schaltfunktion $f \in \mathbf{S}_n$ ist eindeutig darstellbar durch ihre disjunktive Normalform.

$$f = \bigvee_{\substack{p \in \mathbf{B}^n \\ f(p)=1}} x_1^{p_1} \cdots x_n^{p_n}$$

2.1.2 Konstruierbarkeit

Andererseits wissen wir auch, dass jedes Signal in einem kombinatorischen Schaltkreis eine Schaltfunktion über den Primäreingängen realisiert. Es gilt sogar:

Satz

Ist A ein kombinatorisches Bausteinsystem, und gibt es kombinatorische Schaltkreise $Cand$, $Cnot$, bzw. Cor , $Cnot$ über A mit zwei (einem) Eingängen a, b (a) und einem Ausgang y , so dass

$$F[y] = \begin{cases} a \cdot b & \text{in } Cand \\ a \vee b & \text{in } Cor \\ \bar{a} & \text{in } Cnot \end{cases}$$

dann ist jede Schaltfunktion auf einem Ausgang eines entsprechenden Schaltkreises über A realisierbar.

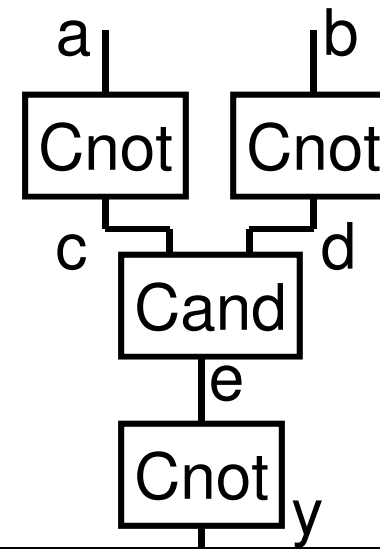
Beweis

Sei f eine beliebige Schaltfunktion. Dann gibt es einen booleschen Ausdruck, der f darstellt (schlimmstenfalls die DNF). Sei w ein solcher Ausdruck, und $T(w)$ ein Syntaxbaum dazu.

Da *Cand* und *Cnot* oder *Cor* und *Cnot* schon existieren, muss nur noch die ggf. fehlende boolesche Operation realisiert werden. Dies ist aber nach der de Morgan'schen Regel sehr einfach:

$$F[y] = \overline{F[e]} = \overline{F[c]F[d]} = \overline{\overline{a}\overline{b}} = a \vee b$$

Also liefert diese Schaltung eine Schaltung mit der Eigenschaft für *Cor*. (Analog erhält man im anderen Falle *Cand*)



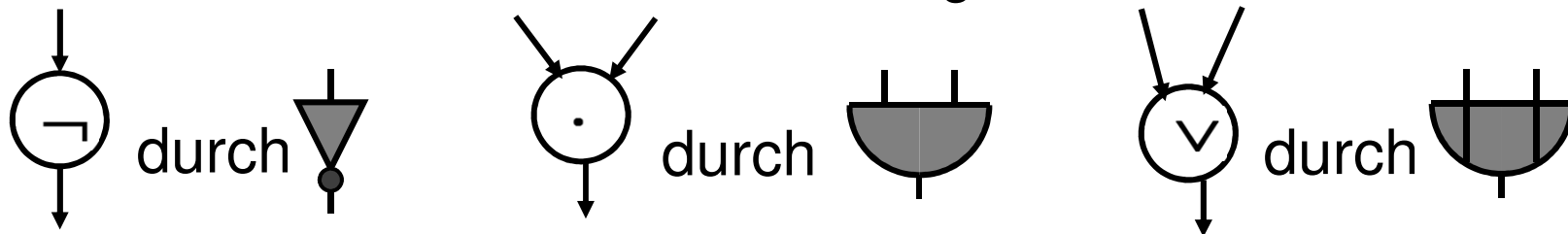
Beweis ff

Wir können also ohne Einschränkung annehmen, dass wir zu jeder booleschen Operation einen Schaltkreis über A haben, und nehmen folgende Symbole als Kürzel dafür:



Nun transformieren wir den Syntaxbaum $T(w)$ zu f wie folgt in einen kombinatorischen Schaltkreis:

1. Ordne allen Knoten der Markierung $\boxed{x_i}$ das Signal x_i zu.
2. Ersetze alle Knoten der Markierung



unter Beibehaltung der Verbindungen.

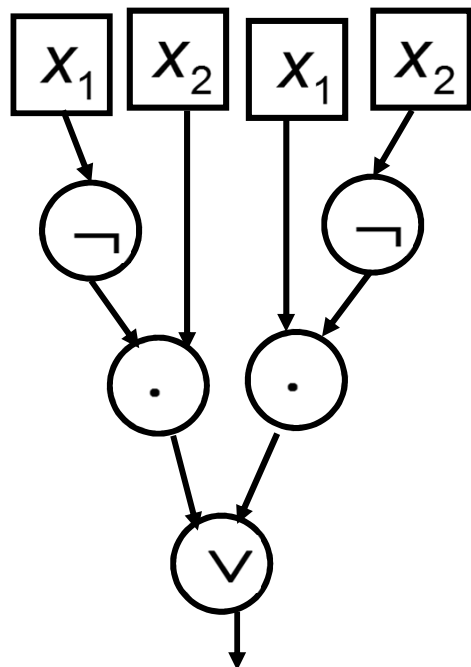
Beweis ff und Beispiel:

Dann gilt für den Ausgang y des so entstandenen Schaltkreises

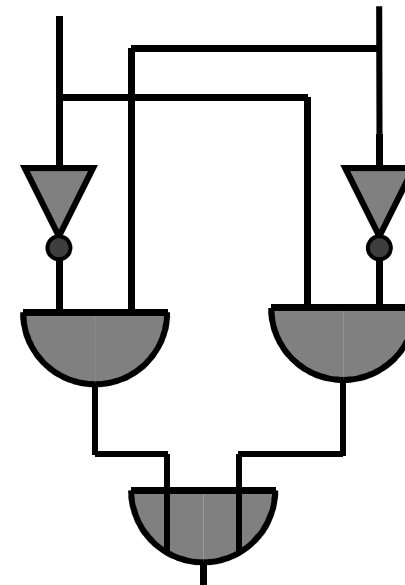
$$F[y] = \mathbf{I}(w) = f$$

Beispiel:

$$w = \bar{x}_1 x_2 \vee x_1 \bar{x}_2$$



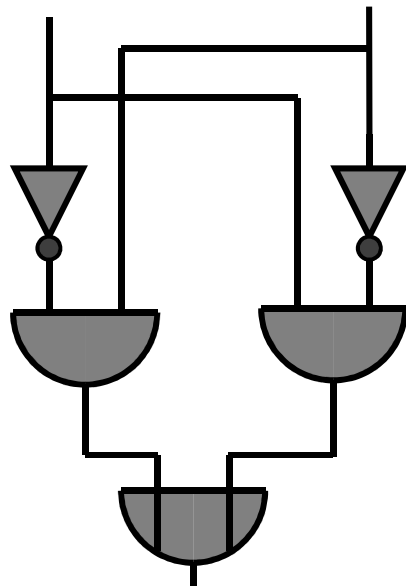
Syntaxbaum



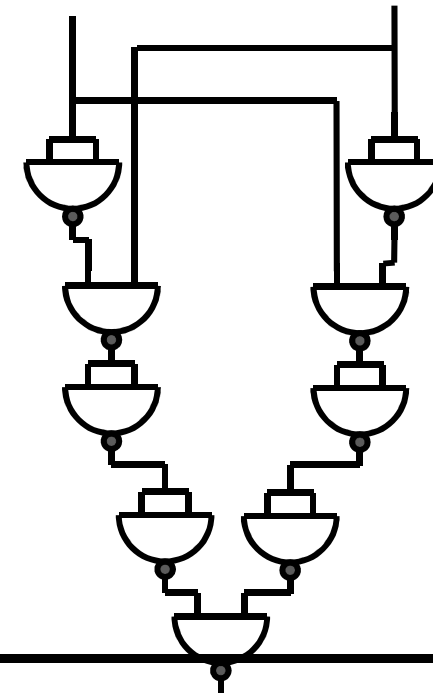
Transformation

Beispiel:

Wir nehmen nun an, dass unser Bausteinsystem nur aus einem *NAND2* Baustein besteht. Dann können wir natürlich *Cand*, *Cor*, *Cnot* realisieren:



liefert dann

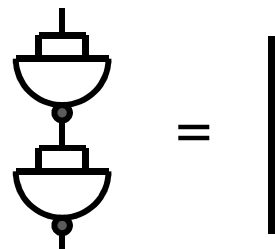


Beispiel:

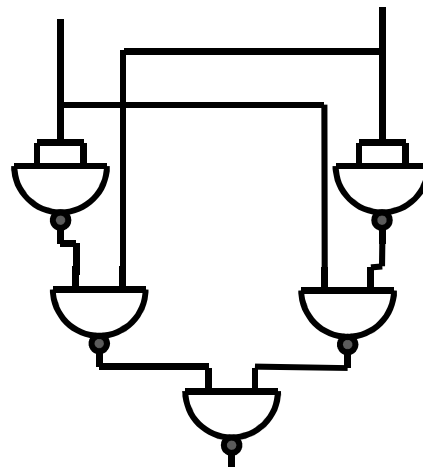
Man erhält also eine Schaltung aus **9** Gattern!?!

Diese Realisierung ist „lausig“.

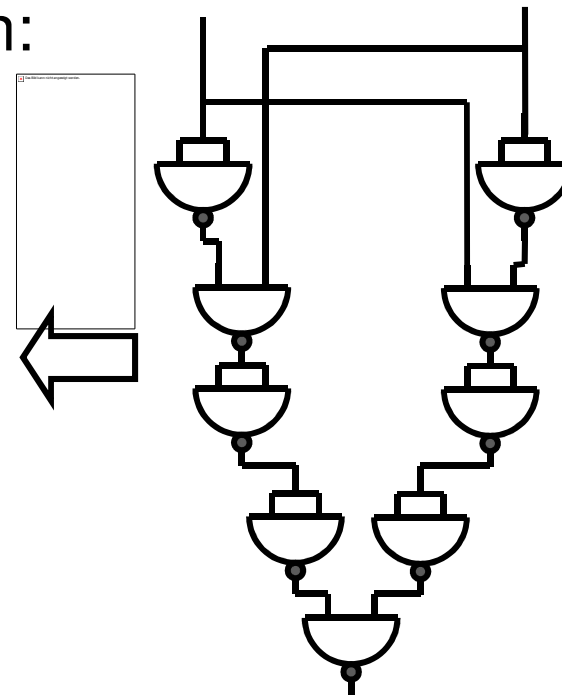
Natürlich kann man nun Vereinfachungen vornehmen, die die Funktion am Ausgang nicht ändern, wie etwa das Entfernen von Doppelnegationen:



Und erhält

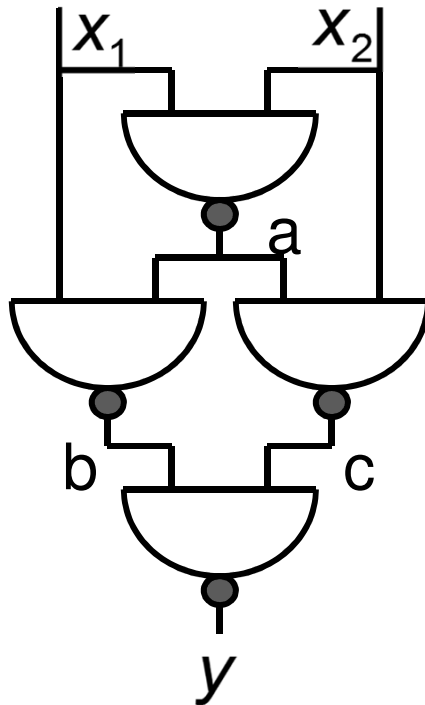


5 Gatter



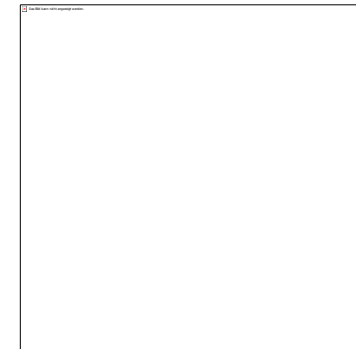
Beispiel:

Betrachte



$$\begin{aligned} F[y] &= \overline{F[b]F[c]} = \overline{x_1F[a] \cdot x_2F[a]} \\ &= \overline{x_1x_1x_2} \cdot \overline{x_2x_1x_2} = \overline{x_1x_1x_2} \vee \overline{x_2x_1x_2} \\ &= x_1(\overline{x_1} \vee \overline{x_2}) \vee x_2(\overline{x_1} \vee \overline{x_2}) \\ &= x_1\overline{x_2} \vee \overline{x_1}x_2 \end{aligned}$$

Es geht sogar mit 4 Gattern!



Bemerkungen

Dieses Beispiel verdeutlicht:

1. Wir müssen Schaltfunktionen durch Schaltkreise, nicht durch Ausdrücke, realisieren.
2. Jedem Ausdruck kann man aber einen Schaltkreis zuordnen, dessen Gatterzahl proportional zur Länge des Ausdrucks ist.
3. Es gibt allerdings Schaltkreise mit sehr viel weniger Gattern als die Länge des kürzesten Ausdrucks, der die gleiche Funktion darstellt.
4. Um gute Lösungen zu finden, muss man die Eigenarten der Technologie, d.h. des Bausteinsystems, ausnutzen können.

Bemerkungen

Ein Kernproblem des Hardwareentwurfs ist also:

Syntheseproblem für kombinatorische Schaltkreise

Gegeben: Eine Schaltfunktion in irgendeiner Darstellung (Ausdruck, Schaltkreis, ...) und die Beschreibung einer Technologie in Form eines kombinatorischen Bausteinsystems A .

Gesucht: Ein kombinatorischer Schaltkreis C , der die Funktion berechnet unter minimaler Gatterzahl (minimaler Laufzeit) bei gegebener Laufzeit (Gatterzahl), falls dieser existiert.

Für dieses Problem gibt es bis heute kein exaktes Lösungsverfahren, das nicht alle Schaltungen erschöpfend aufzählt!

2.1.3 Disjunktive Formen

Für den Sonderfall kombinatorischer Schaltkreise zu Ausdrücken in disjunktiver Form gibt es Minimierungsverfahren, die auf **Quine und Mc Cluskey** zurückgehen. Solche und weitergehende Minimierungsverfahren für mehrstufige kombinatorische Schaltkreise sind in modernen Synthesewerkzeugen implementiert. Ihre Schilderung würde den Rahmen der Vorlesung sprengen. Wir wollen aber am Beispiel disjunktiver Formen zeigen, welche Aufgaben solche Synthesewerkzeuge lösen.

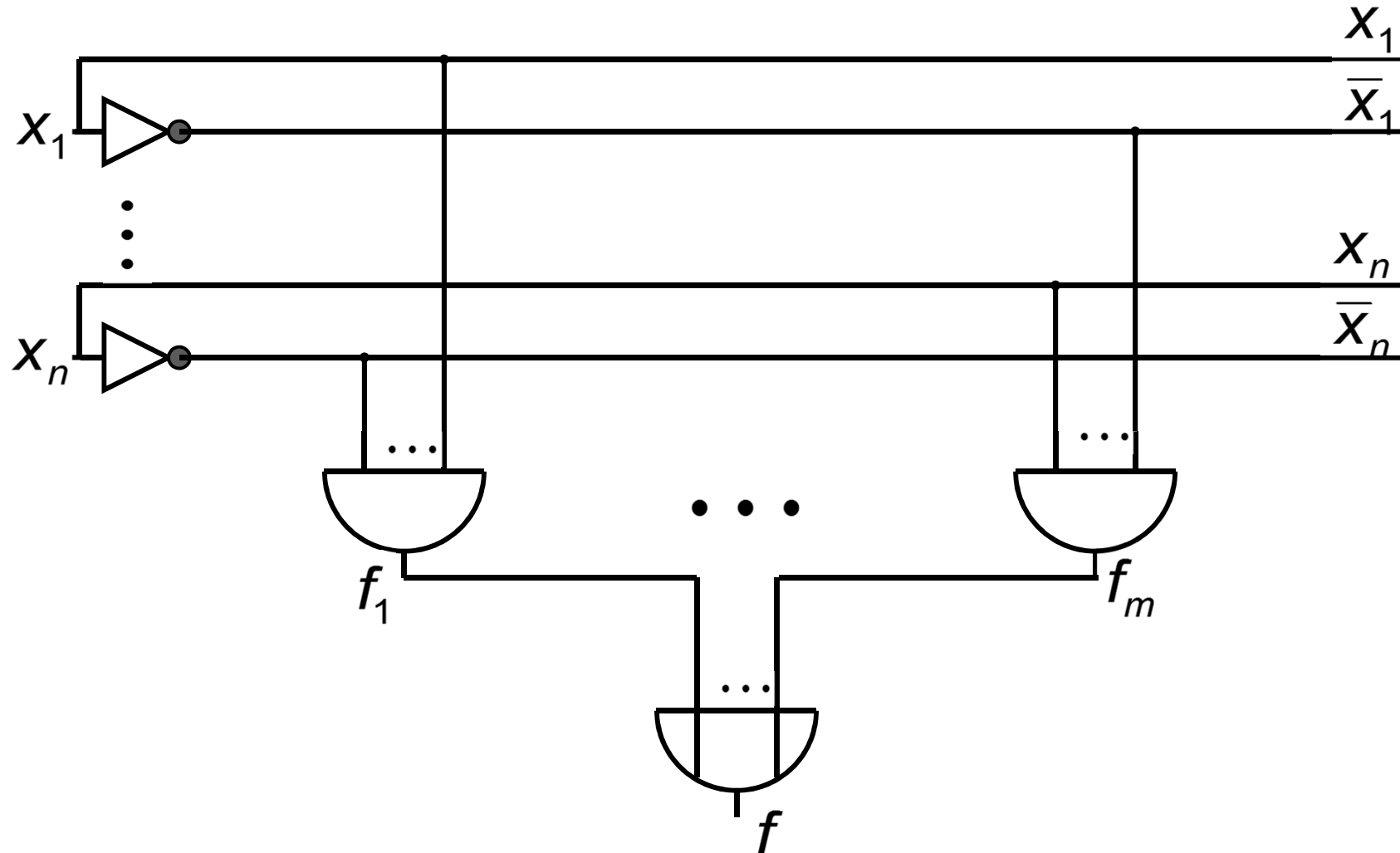
Gegeben sei eine disjunktive Form $f = f_1 \vee \dots \vee f_m$

wobei die f_i Produkte über den Variablen x_1, \dots, x_n sind.

Dann kann man einer solchen disjunktiven Form f einen kanonischen Schaltkreis $C(f)$ zuordnen, der eine Und-Stufe und eine Oder-Stufe hat. Wir nennen $C(f)$ daher auch einen **zweistufigen Schaltkreis**.

Zweistufiger Schaltkreis zur DF

$$f = f_1 \vee \dots \vee f_m$$



Zweistufige Schaltkreise

Man realisiert also einfach

- alle Literale, die in Produkten von f vorkommen, dann
- alle Produkte von f durch Und-Gatter, und dann
- f selbst durch eine m -stelliges Oder-Gatter.

Ähnlich kann man für eine Funktion f in k Ausgängen vorgehen:

Stelle f dar als

$$\begin{array}{ll} f_1 &= f_{11} \vee \dots \vee f_{1m_1} \\ \vdots & \quad \quad \quad \vdots \\ f_k &= f_{k1} \vee \dots \vee f_{km_k} \end{array} \quad \begin{array}{l} \text{Realisiere dann die benötigten} \\ \text{Literale, dann alle Produkte } f_{ij} \\ \text{und schließlich alle } f_i \end{array}$$

Wir nennen solche Schaltkreise „**zweistufig**“.

Beispiel:

Aufgabe: Realisiere eine Schaltung zur Umkodierung von 3 Bit Zahlen im Binärcode in den **Graycode**:

Anm.: Der Graycode ist eine Zahlendarstellung bei der sich die Darstellung von i und $i+1$ nur um ein Bit unterscheiden.

Wir definieren die Funktion durch folgende Tabelle:

x_1	x_2	x_3	$f_1(x)$	$f_2(x)$	$f_3(x)$
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	1
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	1	1	1
1	1	0	1	0	1
1	1	1	1	0	0

Ansatz: Realisiere jede Komponente in DNF

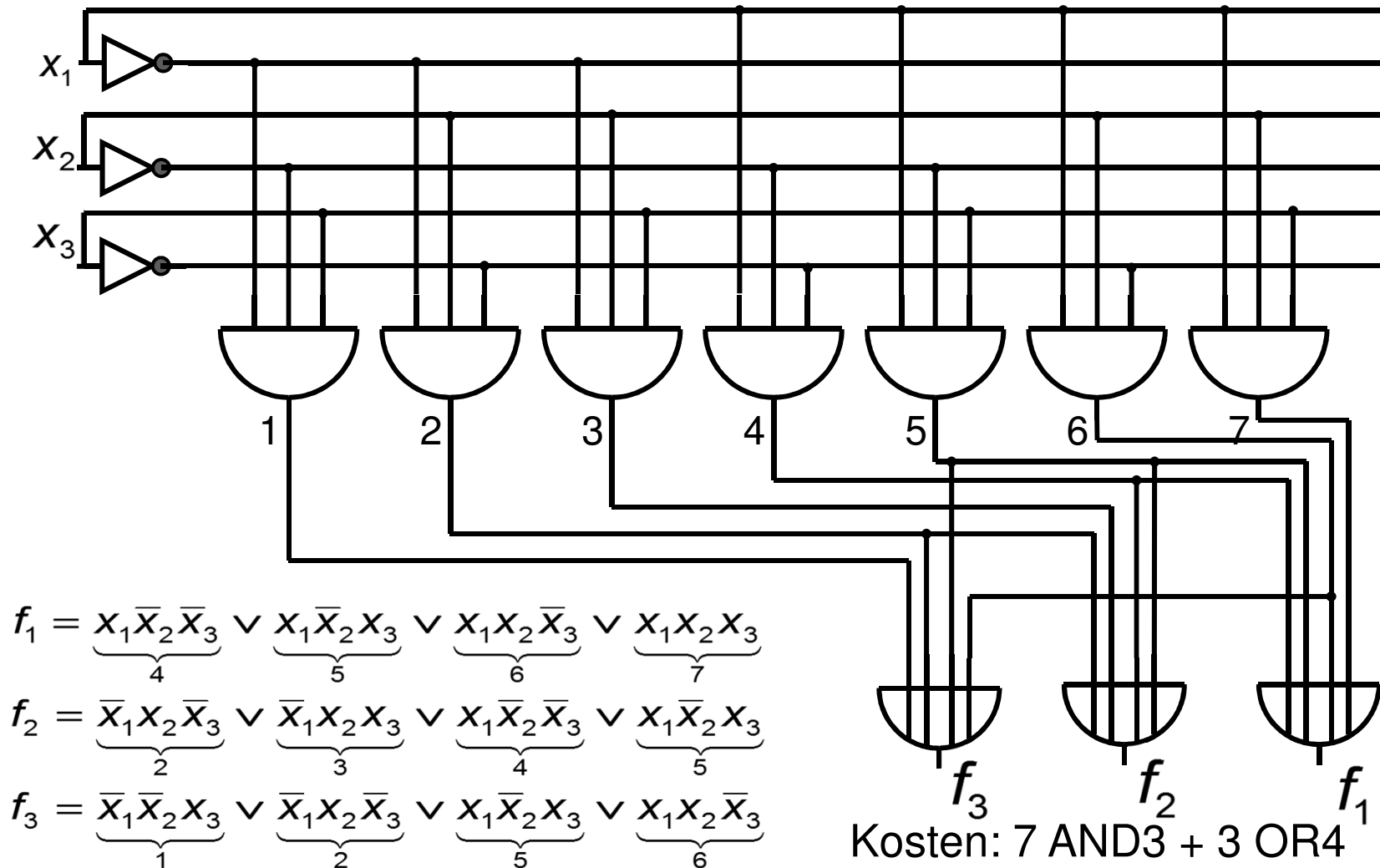
$$f_1 = \underbrace{x_1 \bar{x}_2 \bar{x}_3}_4 \vee \underbrace{x_1 \bar{x}_2 x_3}_5 \vee \underbrace{x_1 x_2 \bar{x}_3}_6 \vee \underbrace{x_1 x_2 x_3}_7$$

$$f_2 = \underbrace{\bar{x}_1 x_2 \bar{x}_3}_2 \vee \underbrace{\bar{x}_1 x_2 x_3}_3 \vee \underbrace{x_1 \bar{x}_2 \bar{x}_3}_4 \vee \underbrace{x_1 \bar{x}_2 x_3}_5$$

$$f_3 = \underbrace{\bar{x}_1 \bar{x}_2 x_3}_1 \vee \underbrace{\bar{x}_1 x_2 \bar{x}_3}_2 \vee \underbrace{x_1 \bar{x}_2 x_3}_5 \vee \underbrace{x_1 x_2 \bar{x}_3}_6$$

Beispiel:

Dann liefert diese Darstellung folgende Realisierung:



Beispiel ff:

Die Realisierung kostet uns also 7 AND3 plus 3 OR4. Das sind, zählt man einmal nur die benötigten Gattereingänge insgesamt: $7 \cdot 3 + 3 \cdot 4 = \mathbf{33}$

Es gibt aber auch andere disjunktive Darstellungen, z.B:

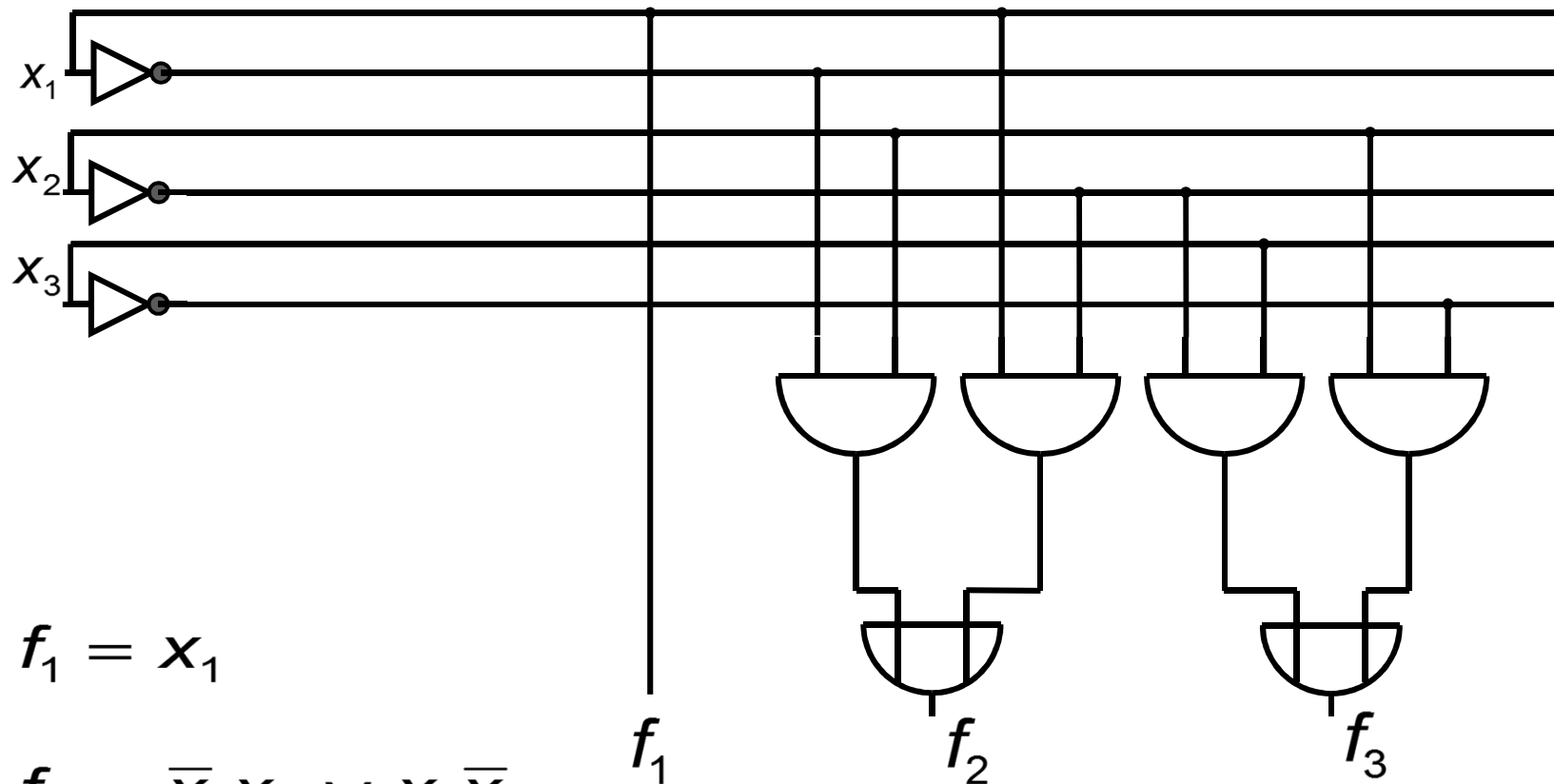
$$\begin{aligned} f_1 &= x_1 \bar{x}_2 \bar{x}_3 \vee x_1 \bar{x}_2 x_3 \vee x_1 x_2 \bar{x}_3 \vee x_1 x_2 x_3 \\ &= x_1 \bar{x}_2 (\bar{x}_3 \vee x_3) \vee x_1 x_2 (\bar{x}_3 \vee x_3) \\ &= x_1 \bar{x}_2 \vee x_1 x_2 = x_1 (\bar{x}_2 \vee x_2) = \boxed{x_1} \end{aligned}$$

$$\begin{aligned} f_2 &= \bar{x}_1 x_2 \bar{x}_3 \vee \bar{x}_1 x_2 x_3 \vee x_1 \bar{x}_2 \bar{x}_3 \vee x_1 \bar{x}_2 x_3 \\ &= \bar{x}_1 x_2 (\bar{x}_3 \vee x_3) \vee x_1 \bar{x}_2 (\bar{x}_3 \vee x_3) = \boxed{\bar{x}_1 x_2 \vee x_1 \bar{x}_2} \end{aligned}$$

$$\begin{aligned} f_3 &= \bar{x}_1 \bar{x}_2 x_3 \vee \bar{x}_1 x_2 \bar{x}_3 \vee x_1 \bar{x}_2 x_3 \vee x_1 x_2 \bar{x}_3 \\ &= \bar{x}_1 (\bar{x}_2 x_3 \vee x_2 \bar{x}_3) \vee x_1 (\bar{x}_2 x_3 \vee x_2 \bar{x}_3) = \boxed{\bar{x}_2 x_3 \vee x_2 \bar{x}_3} \end{aligned}$$

Beispiel:

Diese Vereinfachung liefert folgende Realisierung:



$$f_1 = x_1$$

$$f_2 = \bar{x}_1 x_2 \vee x_1 \bar{x}_2$$

$$f_3 = \bar{x}_2 x_3 \vee x_2 \bar{x}_3$$

Kosten: 4 AND2 + 2 OR2

12 statt 33

Syntheseproblem für 2 stufige Schaltungen

Wir haben an diesem Beispiel gesehen, dass man Freiheitsgrade bei der disjunktiven Darstellung geschickt ausnutzen kann, um bessere Schaltungen zu erhalten. Wir haben es also mit folgendem Optimierungsproblem zu tun:

Gegeben: Eine Schaltfunktion $f \in \mathcal{S}_{n,k}^D$

Gesucht: Eine disjunktive Form zu f mit minimalen Kosten.

Wir können dieses Problem im Rahmen der Vorlesung nicht vollständig behandeln, wollen aber skizzieren, wie man es lösen kann.

Instanzen dieses Problems treten auf dem Weg von einer WüHDL Beschreibung zu einem Schaltkreis hin nicht selten auf:

- Minimiere die Formel der rechten Seite einer Signalzuweisung.
- Erzeuge zu einer Funktion von Bitvektoren nach Bitvektoren eine DF.

Redundanz

Satz

Sei \mathbf{I}_D die Interpretation über \mathcal{S}_n^D . Sei ferner r ein Ausdruck mit $ON(\mathbf{I}(r)) = \mathbf{B}^n \setminus D$ und g, h boolesche Ausdrücke über $X = \{x_1, \dots, x_n\}$ unter Standardinterpretation \mathbf{I} , dann gilt

$$g \equiv_{\mathbf{I}_D} h \iff g \vee r \equiv_{\mathbf{I}} h \vee r$$

Definition

Wir nennen eine(n Ausdruck) Funktion r , deren ON-Set genau aus den Elementen außerhalb des Definitionsbereichs D einer partiellen Funktion f besteht auch einfach (Darstellung der) **Redundanz** von f .

Beweis

$$g \vee r \equiv_{\mathbf{I}} h \vee r \Leftrightarrow \forall p \in \mathbf{B}^n : \mathbf{I}(g \vee r)(p) = \mathbf{I}(h \vee r)(p)$$

$$\Leftrightarrow \forall p \in \mathbf{B}^n : \mathbf{I}(g)(p) \vee \mathbf{I}(r)(p) = \mathbf{I}(h)(p) \vee \mathbf{I}(r)(p)$$

$$\Leftrightarrow \forall p \in \mathbf{B}^n : \begin{cases} 1 = 1 & p \notin D \\ \mathbf{I}(g)(p) = \mathbf{I}(h)(p) & \text{sonst} \end{cases}$$

$$\Leftrightarrow \forall p \in D : \mathbf{I}(g)(p) = \mathbf{I}(h)(p)$$

$$\Leftrightarrow g \equiv_{\mathbf{I}_D} h$$

Neuformulierung des Problems:

Wir können nun das Problem neu über disjunktiven Formen unter Standardinterpretation formulieren:

Minimierung zweistufiger Schaltungen

Gegeben: disjunktive Formen r, f_1, \dots, f_k

Gesucht: disjunktive Formen $g=(g_1, \dots, g_k)$ mit

$$(i) \quad g_i \vee r = f_i \vee r \text{ für alle } 1 \leq i \leq k$$

$$(ii) \quad cost(g) = \min\{cost(h) \mid h \text{ erfüllt (i)}\}$$

Dabei sei $cost(g) = (m, l)$, wobei m die Zahl der Produkte in g und l die Summe über die Zahl der Literale aller Produkte ist. Die Ordnung sei die lexikographische Ordnung auf Tupeln.

Implikanten

Wir beschränken uns hier auf das Problem für $k=1$, d.h.

$$f \in \mathbf{S}_n^D$$

Gegeben: disjunktive Formen r, f

Gesucht: g minimaler Kosten, mit $g \vee r = f \vee r$

Definition

Ein Produkt p heißt **Implikant** einer Funktion $f \in \mathbf{S}_n^D$ genau dann, wenn

$$p \cdot (f \vee r) = p \quad (\Leftrightarrow p \leq f \vee r \Leftrightarrow p \vee f \vee r = f \vee r)$$

Implikanten ff

Lemma

Sei $g \vee r = f \vee r$ für eine disjunktive Form g und eine partielle Funktion f , dann ist jedes g_i Implikant von f .

Beweis:

$$\begin{aligned} g_i \cdot (f \vee r) &= g_i \cdot (g \vee r) \\ &= g_i \cdot g \vee g_i \cdot r \\ &= g_i \cdot g_i \vee g_i \cdot \left(\bigvee_{j \neq i} g_j \right) \vee g_i \cdot r \\ &= g_i \vee g_i \cdot \left(\bigvee_{j \neq i} g_j \vee r \right) \\ &= g_i \end{aligned}$$

Primimplikanten

Wir brauchen also nur DF's zu untersuchen, die ausschließlich aus Implikanten der Funktion bestehen.

Dies kann man sogar weiter einschränken:

Definition

Ein Implikant p einer partiellen Funktion f heißt **Primimplikant** einer Funktion f genau dann, wenn es keinen Implikanten $p' \neq p$ von f gibt, mit

$$p \cdot p' = p \quad (\Leftrightarrow \quad p \leq p' \Leftrightarrow p \vee p' = p')$$

Beispiel:

Wir betrachten wieder unsere DNF-Lösung des Umkodierers: Es war

$$f_3 = \bar{x}_1 \bar{x}_2 x_3 \vee \bar{x}_1 x_2 \bar{x}_3 \vee x_1 \bar{x}_2 x_3 \vee x_1 x_2 \bar{x}_3$$

Die Minterme sind natürlich Implikanten. Aber auch $\bar{x}_2 x_3$

denn

$$\begin{aligned}\bar{x}_2 x_3 \cdot f_3 &= \bar{x}_1 \bar{x}_2 x_3 \vee 0 \vee x_1 \bar{x}_2 x_3 \vee 0 \\ &= (\bar{x}_1 \vee x_1) \bar{x}_2 x_3 \\ &= \bar{x}_2 x_3\end{aligned}$$

Ist dieser Implikant prim?

Dazu müssen wir alle Produkte $p \neq \bar{x}_2 x_3$ mit $p \bar{x}_2 x_3 = \bar{x}_2 x_3$ auf ihre Implikanteneigenschaft hin untersuchen. In Frage kommen dazu nur: $p \in \{\bar{x}_2, x_3, 1\}$

Die 1 entfällt, da die Funktion nicht konstant 1 ist.

Beispiel ff

Ferner ist

$$\begin{aligned}\bar{x}_2 \cdot f_3 &= \bar{x}_1 \bar{x}_2 x_3 \vee 0 \vee x_1 \bar{x}_2 x_3 \vee 0 \\ &= (\bar{x}_1 \vee x_1) \bar{x}_2 x_3 = \bar{x}_2 x_3\end{aligned}$$

Also \bar{x}_2 kein Implikant!

Ferner ist

$$\begin{aligned}x_3 \cdot f_3 &= \bar{x}_1 \bar{x}_2 x_3 \vee 0 \vee x_1 \bar{x}_2 x_3 \vee 0 \\ &= (\bar{x}_1 \vee x_1) \bar{x}_2 x_3 \\ &= \bar{x}_2 x_3\end{aligned}$$

Also x_3 kein Implikant!

Damit ist $\bar{x}_2 x_3$ Primimplikant von f_3

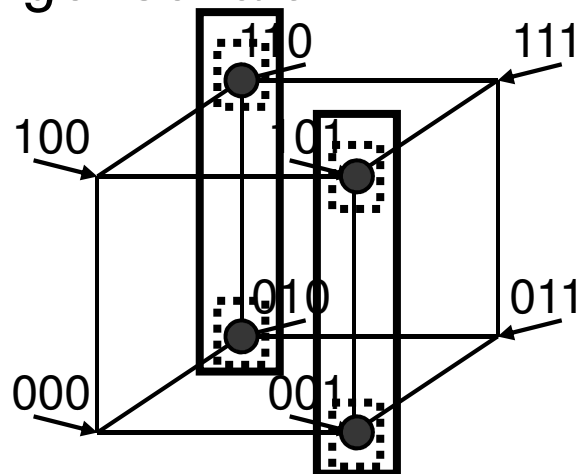
Bemerkung: Für zwei Produkte p und q gilt $p \cdot q = q$ nur dann, wenn $Lit(p) \subseteq Lit(q)$, d.h. „größere“(im Verband) Produkte sind Teilprodukte (Teiler) von „kleineren“. Daher der Begriff „prim“.

Geometrische Anschauung:

Wir hatten uns überlegt, dass der ON-Set eines Produktes stets einen Unterwürfel eines n -dimensionalen Einheitswürfels bildet.

Ein Implikant einer Funktion ist demnach ein Unterwürfel, dessen Ecken nur Punkte aus dem ON-Set und der Redundanz der Funktion enthält.

Ein Primimplikant ist ein maximaler Unterwürfel mit dieser Eigenschaft:



$$f_3 = \bar{x}_1 \bar{x}_2 x_3 \vee \bar{x}_1 x_2 \bar{x}_3 \vee x_1 \bar{x}_2 x_3 \vee x_1 x_2 \bar{x}_3$$

⋮- Implikanten

□- Primimplikanten

Das Primimplikantentheorem

Satz (Primimplikantentheorem)

Sei g eine disjunktive Form für $f \in \mathcal{S}_n^D$ unter Redundanz r mit minimalen Kosten. Dann ist jedes Produkt g_i von g ein Primimplikant.

Beweis: indirekt

Annahme: g sei kostenminimal, enthalte aber ein Produkt p , das nicht prim ist, d.h. g hat die Form: $g = p \vee h$

Dann ist $cost(g) = cost(p) + cost(h) = (1, \#Lit(p)) + cost(h)$

Da p nicht prim ist, gibt es einen Implikanten q , mit

$$q \neq p \text{ und } p \cdot q = p, \text{ d.h. } Lit(q) \subset Lit(p)$$

Beweis ff

Demnach ist $cost(q \vee h) = cost(h) + (1, \#Lit(q))$
 $< cost(h) + (1, \#Lit(p)) = cost(g)$
da $Lit(q) \subset Lit(p)$

Andererseits ist, da auch q Implikant von f , aber

$$\begin{aligned} f \vee r &= f \vee r \vee q \\ &= g \vee r \vee q \\ &= p \vee h \vee r \vee q \\ &= \underbrace{p \vee q}_{=q} \vee h \vee r = (q \vee h) \vee r \end{aligned}$$

Also wäre $q \vee h$ eine Darstellung von f , aber

$$cost(q \vee h) < cost(g) \quad \text{↯} \quad (g \text{ kostenminimal})$$

Beispiel:

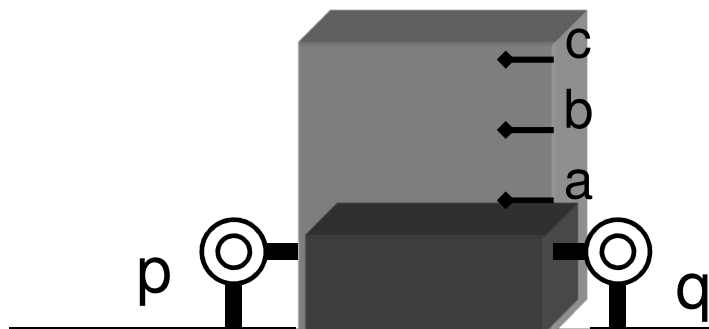
Das Primimplikantentheorem bringt uns der Sache ein gutes Stück näher:

Man braucht nur noch Darstellungen zu betrachten, die ausschließlich aus Primimplikanten bestehen!

Wir verdeutlichen dies an einem einfachen

Beispiel:

Gegeben sei ein Tank mit zwei Pumpen p, q und Sensoren a, b, c für den Füllstand:



$a=1$: mindestens $1/3$ voll

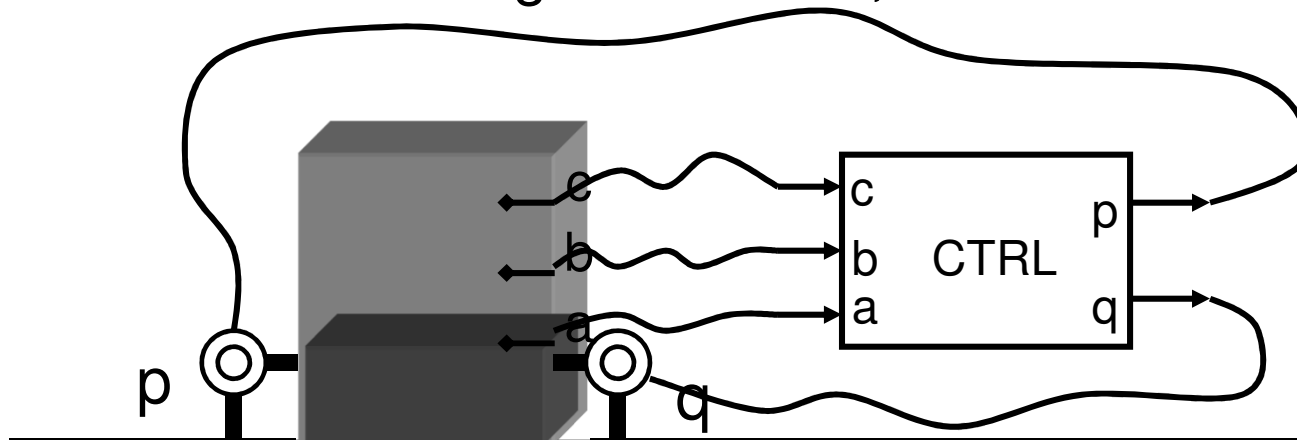
$b=1$: mindestens $2/3$ voll

$c=1$: voll

Beispiel ff

Gesucht ist eine Schaltung *CTRL* zur Steuerung der Pumpen nach folgenden Regeln:

- ☞ Die schwächere Pumpe p ($p=1$) soll laufen, wenn der Behälter mindestens $2/3$ voll, nicht jedoch wenn er voll ist.
- ☞ Ist der Behälter weniger als $2/3$ aber mindestens $1/3$ voll, soll die stärkere Pumpe q ($q=1$) laufen.
- ☞ Sinkt der Pegel unter $1/3$, sollen beide Pumpen laufen.



Beispiel ff

Wir stellen eine Funktionstafel dazu auf:

a	b	c	p	q	r
0	0	0	1	1	0
0	0	1	*	*	1
0	1	0	*	*	1
0	1	1	*	*	1
1	0	0	0	1	0
1	0	1	*	*	1
1	1	0	1	0	0
1	1	1	0	0	0

Redundant!

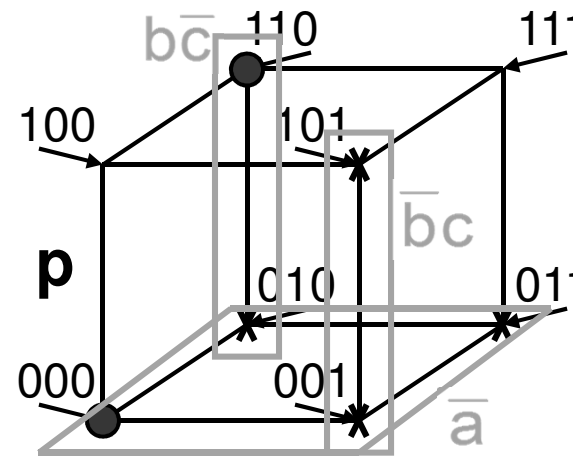
Nicht alle Eingangskombinationen kommen vor, da für die Sensorsignale einbaubedingt stets $a \geq b \geq c$ gelten muss.

Beispiel ff

Wir haben also partielle Funktionen $p, q \in S_3^D$ zu realisieren, wobei die Redundanz $r = \bar{a}\bar{b}c \vee \bar{a}b\bar{c} \vee \bar{a}bc \vee a\bar{b}c$

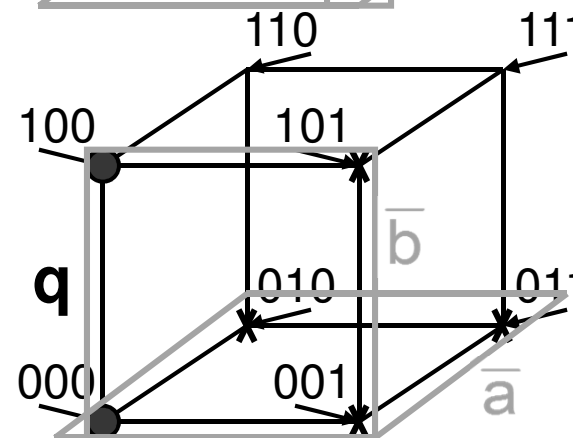
a	b	c	p	q	r
0	0	0	1	1	0
0	0	1	*	*	1
0	1	0	*	*	1
0	1	1	*	*	1
1	0	0	0	1	0
1	0	1	*	*	1
1	1	0	1	0	0
1	1	1	0	0	0

Ziel: Finde die Primimplikanten für p, q



Primimplikanten
zu p:

\bar{a} , $b\bar{c}$, $\bar{b}c$



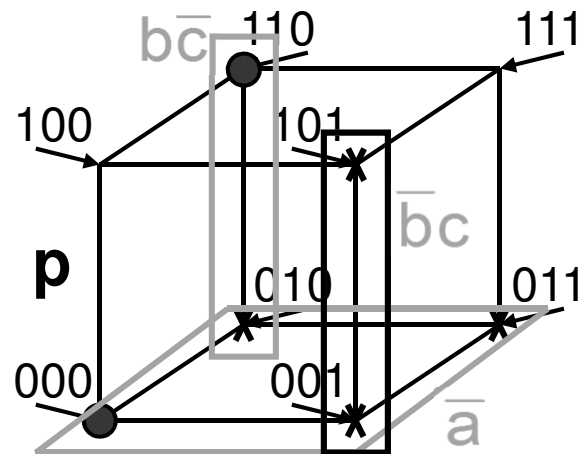
Primimplikanten
zu q:

\bar{a} , \bar{b}

Beispiel ff

Wir brauchen nun nur noch alle DF's zu betrachten, die man aus diesen Primimplikanten bilden kann, und eine billigste zu nehmen, die die Funktion realisiert.

Betrachten wir zunächst p :

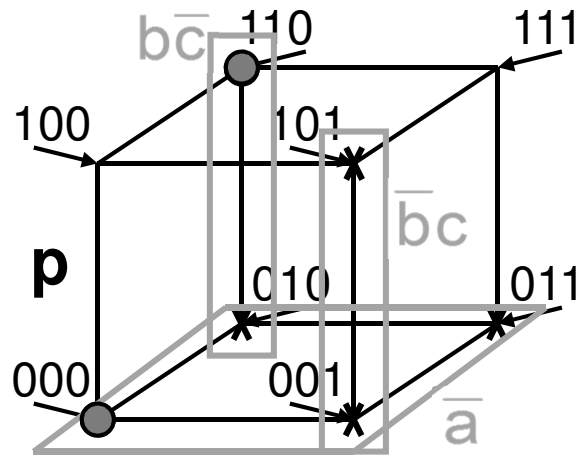


Es fällt auf, dass der Primimplikant $\bar{b}c$ nur Punkte der Redundanz überdeckt. Lässt man ihn in einer Realisierung g weg, so überdeckt

$$g \vee r = g \vee \bar{b}c \vee r$$

immer noch die gleichen Punkte. Wir nennen solche Primimplikanten **total redundant**.

Beispiel ff



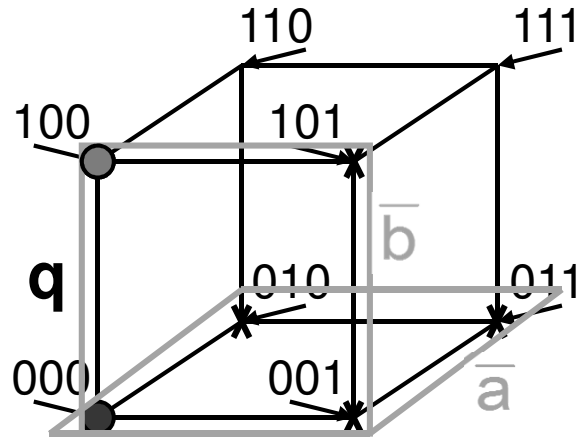
Ferner fällt auf, dass man auf \bar{a} und auch auf $b\bar{c}$ nicht verzichten kann, da sie Elemente des ON-Sets von p enthalten, die jeweils kein anderer Primimplikant enthält.

Man nennt solche Punkte auch **wesentliche Punkte**, und Primimplikanten, die wesentliche Punkte enthalten, **wesentliche Primimplikanten**.

Die billigste Lösung für p lautet also:

$$p = \bar{a} \vee b\bar{c}$$

Beispiel ff



Bei q fällt auf, dass man auf \bar{b} nicht verzichten kann.

Nimmt man aber \bar{b} , so braucht man \bar{a} nicht mehr, weil ja

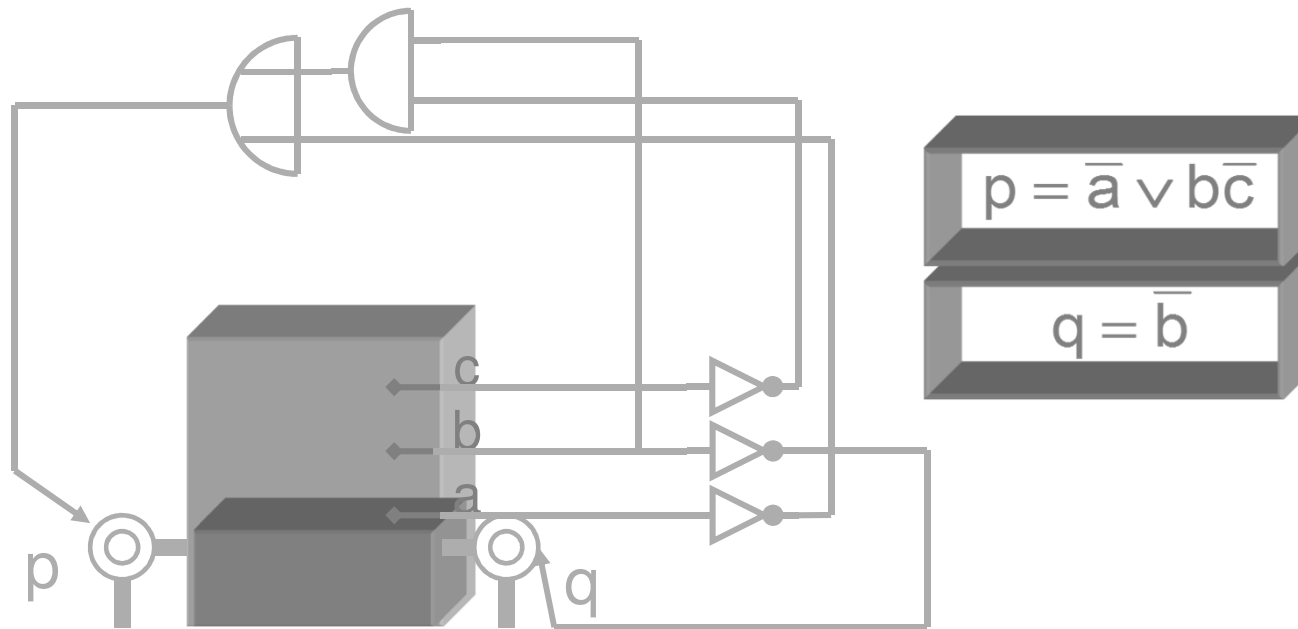
$$\bar{b} \vee \bar{a} \vee r = \bar{b} \vee r$$

Wenn die Hinzunahme eines Primimplikanten u einen anderen Primimplikanten v überflüssig macht, sagen wir auch **u dominiert v** .

Die billigste Lösung für q lautet also:

$$q = \bar{b}$$

Beispiel ff



```
ENTITY ctrl IS
  PORT (a,b,c: IN BIT; p,q: OUT BIT);
END ENTITY;
```

```
ARCHITECTURE implementation OF
ctrl IS
BEGIN
  p <= NOT a OR ( b AND NOT c);
  q <= NOT b;
END ARCHITECTURE;
```

Karnaugh Diagramme



Man kann die Funktionstafel auch als 2 dimensionale Tabelle hinschreiben. Je nachdem, wie man die Eingangsvektoren ordnet, erhält man maximale Unterwürfel aus Einsen oder Redundanzen (= Primimplikanten) durch spezielle Muster in dieser Tabelle.

Im Karnaugh Diagramm ordnet man die Eingangsvektoren so an, dass bis $n=4$ die Unterwürfel stets als zusammenhängende Gebiete sichtbar sind, denkt man sich die Tabelle auf den Torus geflochten.





Im Veitch Diagramm ordnet man die Eingaben als Binärdarstellungen der Größe nach. Dadurch sind manche Unterwürfel nicht zusammenhängend.

Karnaugh Diagramme ff

Veitch Diagramme

a	0	0	1	1
b	0	1	0	1
c	d	\bar{a}	a	
0	0			
0	1			
1	0			
1	1			

a	0	0	1	1
b	0	$\overline{1}$	0	1
c	d			
0	0			
0	1			
1	0			
1	1			






a	0	0	1	1
b	0	1	0	1
c	d	$\overline{b}\overline{d}$		
0	0			
0	1			$\overline{c}d$
1	0			
1	1			

a	0	0	1	1
b	0	1	0	1
c				
d				
0 0				
0 1				
1 0				
1 1				

Karnaugh Diagramme

a	0	0	1	1
b	0	1	1	0
c	d	\bar{a}	a	
0	0			
0	1			
1	1			
1	0			

a	0	0	1	1
b	0	1	1	0
c	\bar{b}	b	\bar{b}	
d	\bar{b}	b	\bar{b}	
0	0			
0	1			
1	1			
1	0			

a	0	0	1	1
b	0	1	1	0
c	d	$\overline{b\overline{d}}$		
0	0			
0	1			
1	1		$c\overline{d}$	
1	0			

a	0	0	1	1
b	0	1	1	0
c				
d				
0 0				
0 1				
1 1				
1 0				

Diagramme zur Pumpensteuerung

a	b	c	p	q	r
0	0	0	1	1	0
0	0	1	*	*	1
0	1	0	*	*	1
0	1	1	*	*	1
1	0	0	0	1	0
1	0	1	*	*	1
1	1	0	1	0	0
1	1	1	0	0	0

$$p = \bar{a} \vee b\bar{c}$$

$$q = \bar{b}$$

p	a	0	0	1	1
b	0	1	0	1	
c					
0		1	*	0	1
1		*	*	*	0

p	a	0	0	1	1
b	0	1	1	0	
c					
0		1	*	1	0
1		*	*	0	*

q	a	0	0	1	1
b	0	1	0	1	
c					
0		1	*	1	0
1		*	*	*	0

q	a	0	0	1	1
b	0	1	1	0	
c					
0		1	*	0	1
1		*	*	0	*

Problem: Bis 4, 5 Variablen sieht man noch was. Was tut man aber für $n > 5$? Benutze Synthesoftware!

2.1.4 Mehrstufige Schaltkreise

Im Falle zweistufiger Schaltkreise kann man das Syntheseproblem, wenn auch mit sehr hohem Aufwand, lösen. Allerdings stößt man bei der Benutzung zweistufiger Schaltungen sehr schnell an Grenzen. Wir betrachten dazu

Aufgabe:

Entwerfe einen kombinatorischen Schaltkreis PGPC (Parity Generator Parity Checker) zur Berechnung folgender Schaltfunktion:

$parity: \mathbf{B}^n \rightarrow \mathbf{B}$, mit

$$parity(x) = \left(\sum_{i=0}^{n-1} x_i \right) \bmod 2$$

PGPC als zweistufiger Schaltkreis

Im Falle einer zweistufigen Lösung liegen die Dinge recht klar:

Beobachtung 1

Jeder Minterm x^ε der Parity Funktion ist schon ein Primimplikant.

Grund: Würde man in einem Minterm $x_1^{\varepsilon_1} \dots x_n^{\varepsilon_n}$ mit $\sum \varepsilon_i \bmod 2 = 1$ ein Literal weglassen, etwa ε_j , dann wäre schon

$$x_1^{\varepsilon_1} \dots x_{j-1}^{\varepsilon_{j-1}} x_{j+1}^{\varepsilon_{j+1}} \dots x_n^{\varepsilon_n} (\varepsilon_1, \dots, \varepsilon_{j-1}, 1, \varepsilon_{j+1}, \dots, \varepsilon_n) =$$

$$x_1^{\varepsilon_1} \dots x_{j-1}^{\varepsilon_{j-1}} x_{j+1}^{\varepsilon_{j+1}} \dots x_n^{\varepsilon_n} (\varepsilon_1, \dots, \varepsilon_{j-1}, 0, \varepsilon_{j+1}, \dots, \varepsilon_n)$$

$$\text{aber } \text{parity}(\varepsilon_1, \dots, \varepsilon_{j-1}, 1, \varepsilon_{j+1}, \dots, \varepsilon_n) \neq \text{parity}(\varepsilon_1, \dots, \varepsilon_{j-1}, 0, \varepsilon_{j+1}, \dots, \varepsilon_n)$$

PGPC als zweistufiger Schaltkreis

Beobachtung 2

Jeder Minterm x^ϵ der Parity Funktion ist schon ein wesentlicher Primimplikant.

Grund: Alle Minterme $x_1^{\epsilon_1} \dots x_n^{\epsilon_n}$ mit $(\sum_i \epsilon_i) \bmod 2 = 1$ werden genau auf einem Punkt $=1$. Diese Punkte sind paarweise verschieden.

Beobachtung 3

Die disjunktive Normalform ist kostenminimale 2-stufige Darstellung der Funktion *parity*.

Fazit:

Jede zweistufige Realisierung von *parity* kostet mehr als 2^{n-1} Gatter!

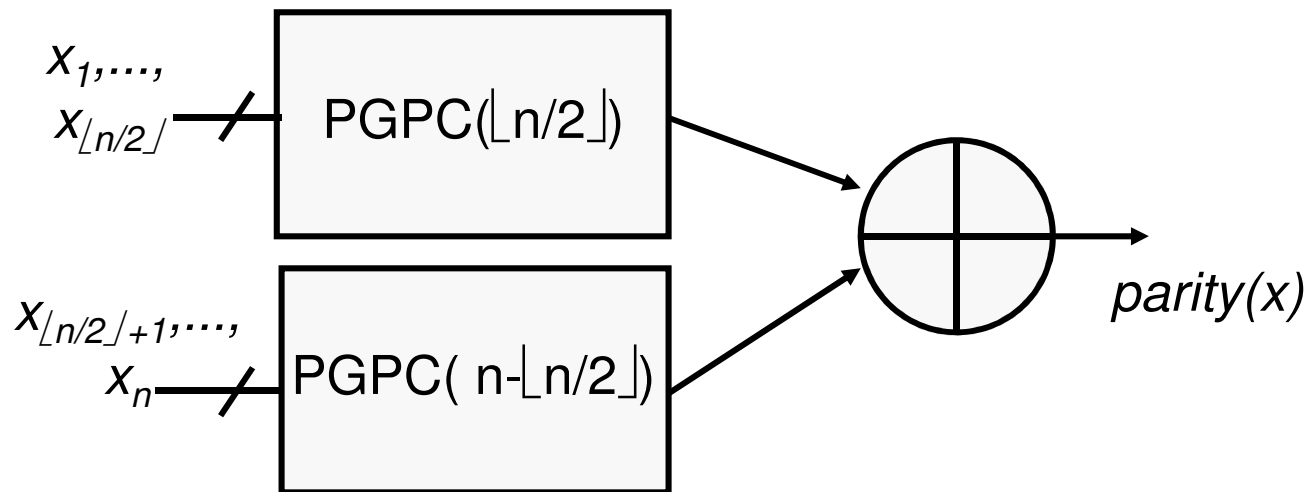
PGPC als mehrstufiger Schaltkreis

Im Falle einer mehrstufigen Lösung bietet sich aber eine denkbar einfache Lösung:

Beobachtung

Die Parity eines Vektors ergibt sich aus der Summe modulo 2 der Parity von zwei Teilvektoren.

Es ergibt sich also folgendes einfache Schema für n bit:



PGPC als mehrstufiger Schaltkreis

Wir brauchen demnach nur $n-1$ XOR Gatter:

Beweis durch Induktion nach n :

$n=2$: $\text{parity}(x_1, x_2) = \text{xor}(x_1, x_2)$. Also nur ein Gatter.

$n>2$: Dann brauchen wir $\lfloor n/2 \rfloor - 1 + (n - \lfloor n/2 \rfloor) - 1 + 1 = n - 1$
Gatter

Man beobachtet auch: dem Schaltkreis liegt ein Bildungsgesetz in Abhängigkeit von der Breite n des Eingabevektors zugrunde. Es wäre also nützlich, eine Definition vornehmen zu können, die generisch in dem Parameter n ist.

WüHDL sieht dazu bei der Definition von ENTITIES sogenannte GENERIC Parameter vor. Damit beschreibt man eine ganze Familie von Bausteinen:

Generische Definition in WüHDL

```
ENTITY pgpc IS
    GENERIC (n : POSITIVE);
    PORT (x: IN BIT_VECTOR(0 TO n-1);
          y: OUT BIT);
END ENTITY pgpc;

ARCHITECTURE structure OF pgpc IS
    COMPONENT pgpc --Benutzt sich selbst zur Rekursion
        GENERIC (n: POSITIVE );
        PORT (x: IN BIT_VECTOR(0 TO n-1);
              y: OUT BIT);
    END COMPONENT;
    COMPONENT xor2
        PORT (x,y: IN BIT; z: OUT BIT);
    END COMPONENT;
    --Zwischenergebnisse (most/least signifikant parity)
    SIGNAL msp, lsp: BIT;
```

Generische Definition in WüHDL -- ff

```
BEGIN --rekursive Definition
  verankerung2: IF n=2 GENERATE
    X0: xor2 PORT MAP (x=>x(0), y=>x(1), z=>y);
  END GENERATE;
  verankerung3: IF n=3 GENERATE
    X1: xor2 PORT MAP (x=>x(0), y=>x(1), z=>msp);
    X2: xor2 PORT MAP (x => x(2), y => msp, z=>y);
  END GENERATE;
  rekursion: IF n>3 GENERATE
    lsb: pgpc -- 1. Aufruf der Rekursion
      GENERIC MAP (n/2)
        PORT MAP (x=>x(n-n/2 TO n-1), y=>lsp);
    msb: pgpc -- 2. Aufruf der Rekursion
      GENERIC MAP (n-n/2)
        PORT MAP (x=>x(0 TO n-n/2-1), y=>msp);
    result: xor2 PORT MAP (x=>msp, y=>lsp, z=>y);
  END GENERATE;
END ARCHITECTURE;
```


Zahlen

Die wohl wichtigsten Objekte in Rechnern sind Zahlen. Die einfachste Art und Weise, Bitstrings als Zahlen aufzufassen, ist ihre Interpretation als nichtnegative ganze Zahl:

Definition

Wir nennen $u_n: B^n \mapsto N_0$

mit $u_n(a_0, \dots, a_{n-1}) := 2^{n-1} \sum_{i=0}^{n-1} a_i 2^{-i}$

Darstellung als vorzeichenlose (unsigned) ganze Zahl.

Das Bit a_0 hat das höchste Gewicht 2^{n-1} . Wir nennen es auch das **signifikanteste Bit**.

Wenn klar ist, dass n die Wortbreite ist, schreiben wir auch einfach u statt u_n .

Unsigned Numbers

Der Zahlenbereich einer vorzeichenlosen ganzen Zahl ist

$$u_n(\mathbf{B}^n) = [0: 2^n - 1]$$

Größere Zahlen kann man nicht direkt darstellen. Will man mit größeren Zahlen arbeiten, dann muss dies explizit durch Programmierung realisiert werden.

Anmerkungen:

Rechnerarithmetiker benutzen oft auch eine gespiegelte Notation der Bitstrings um Zahlen zu definieren. In diesem Fall entspricht das Gewicht eines Bits i direkt 2^i .

Den Zahlenbereich erhält man durch

$$u_n(0, \dots, 0) = 0 \quad \text{sowie}$$

$$u_n(1, \dots, 1) = 2^{n-1} \sum_{i=0}^{n-1} 1 \cdot 2^{-i} = 2^{n-1} (2 - 2^{-(n-1)}) = 2^n - 1$$

Decoder

Wir wollen nun nach und nach sehr häufig vorkommende Schaltfunktionen durch kombinatorische Schaltkreise realisieren. Eine spezielle Familie bilden Realisierungen folgender Schaltfunktion:

$$\begin{aligned} \text{decode}_m: \mathbf{B}^n &\mapsto \mathbf{B}^m \text{ mit} \\ \text{decode}_m(a_0, \dots, a_{n-1}) &= (y_0, \dots, y_{m-1}) \\ &\Leftrightarrow \forall i: y_i = (u_n(a) = i) \end{aligned}$$

Diese Funktion bildet einen Binärcode in einen 1 aus m Code, einen sog. „one hot code“, ab. Deswegen muss natürlich $m \leq 2^n$ sein. Ist $u_n(a) \geq m$, liefert die Funktion den 0-Vektor.

Realisierung des Decoders

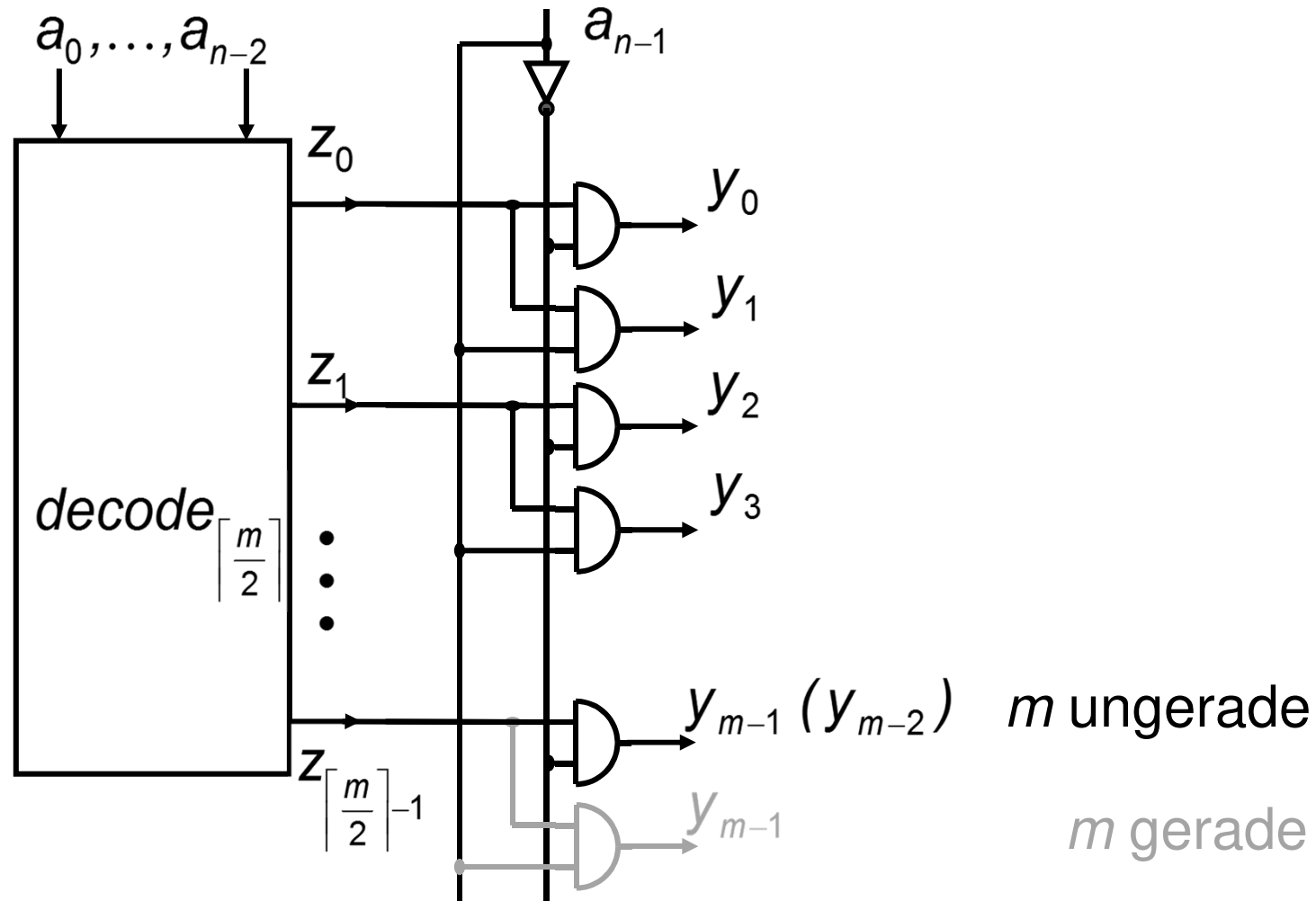
Idee: Wir setzen den Decoder rekursiv aus kleineren Decodern zusammen.

Nehmen wir an, dass wir schon einen Decoder für $\lceil m/2 \rceil$ Ausgänge haben, dann können wir diese Leitungen mit

$$u_n(a_0, \dots, a_{n-2})$$

nummerieren. Einen Decoder mit bis zu doppelt sovielen Ausgängen erhalten wir nun durch Aufspalten dieser Leitungen in eine mit gerader und eine mit ungerader Nummer in Abhängigkeit von a_{n-1} .

Realisierung des Decoders ff



Realisierung des Decoders ff

Für i gerade, ist

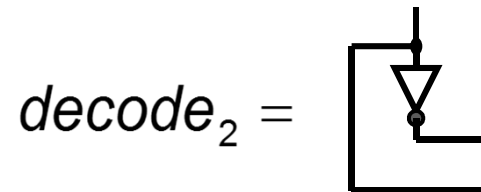
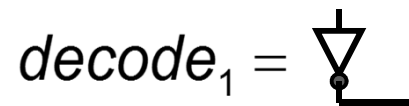
$$y_i \Leftrightarrow \bar{a}_{n-1} \cdot z_{\frac{i}{2}} = 1$$

$$\Leftrightarrow a_{n-1} = 0 \text{ und } u_{n-1}(a[0:n-2]) = \frac{i}{2}$$

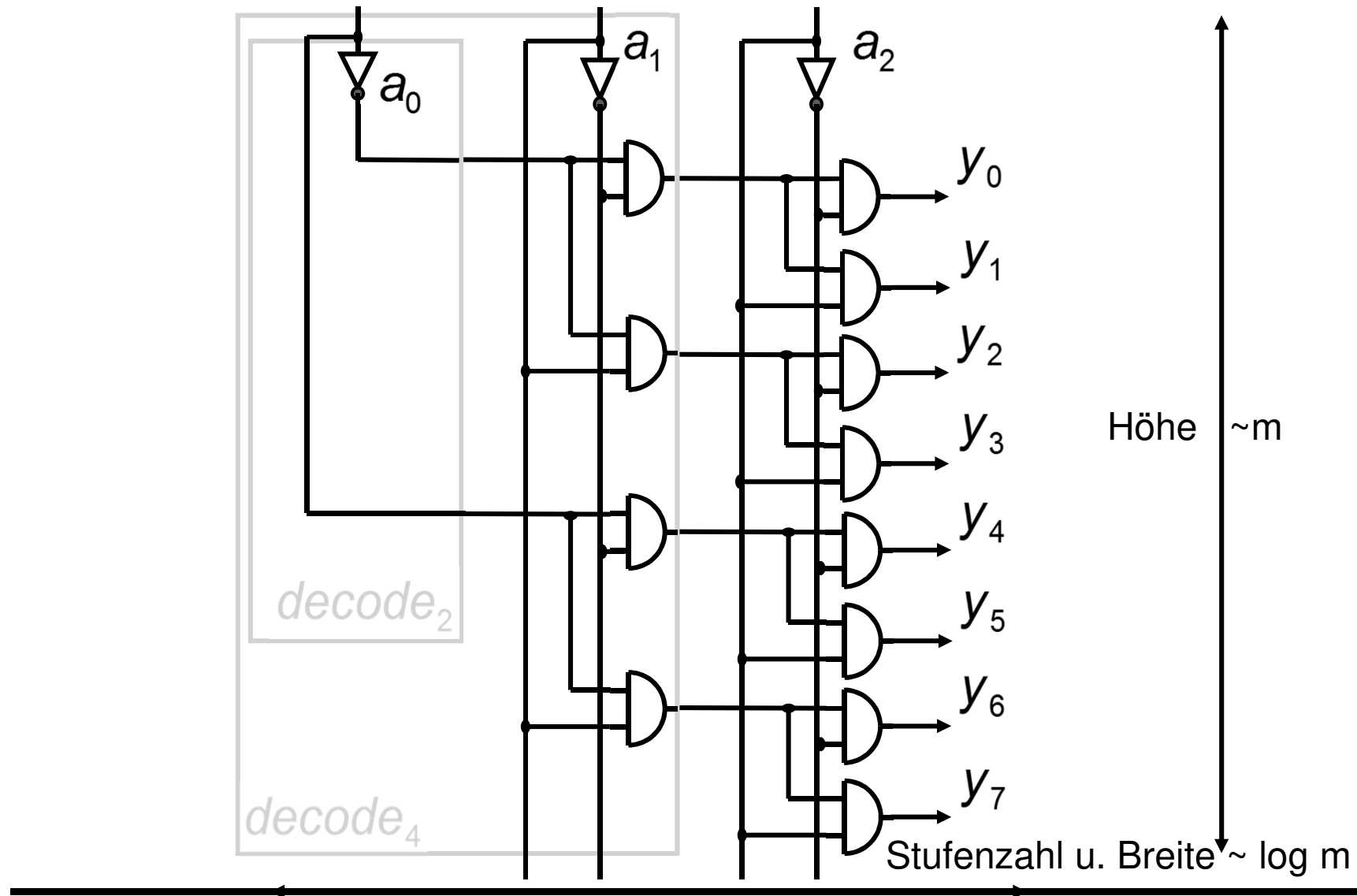
$$\Leftrightarrow i = 2 \cdot 2^{n-2} \sum_{i=0}^{n-2} a_i 2^{-i} + a_{n-1}$$

$$\Leftrightarrow i = 2^{n-1} \left(\sum_{i=0}^{n-2} a_i 2^{-i} + 2^{-(n-1)} a_{n-1} \right) = u_n(a)$$

Analog sieht man die Korrektheit für den Fall i ungerade. Es bleibt also nur noch, die trivialen Decoder für $m=1,2$ zu konstruieren, und wir haben eine Konstruktionsvorschrift für alle Decoder:



Beispiel: Konstruktion für $m=8$



Generische Definition in WüHDL

```
ENTITY decoder IS
```

```
    GENERIC ( n : POSITIVE );
```

```
    PORT (x: IN BIT_VECTOR(0 TO n-1);
```

```
          y: OUT BIT_VECTOR(0 TO 2**n-1) );
```

```
END ENTITY decoder;
```

```
ARCHITECTURE structure OF decoder IS
```

```
    COMPONENT decoder -- zur Rekursion
```

```
        GENERIC ( n: POSITIVE );
```

```
        PORT (x: IN BIT_VECTOR(0 TO n-1);
```

```
              y: OUT BIT_VECTOR(0 TO 2**n-1) );
```

```
    END COMPONENT;
```

```
    COMPONENT and2 PORT (x,y: IN BIT; z: OUT BIT);
```

```
    END COMPONENT;
```

```
    COMPONENT two_phases PORT(x:IN BIT;y,yz:OUT BIT);
```

```
    END COMPONENT;
```

```
--Zwischenergebnis
```

```
SIGNAL r:BIT_VECTOR(0 TO 2** (n-1)-1); SIGNAL l,lz:BIT;
```


Generische Definition in WüHDL -- ff

```
BEGIN --rekursive Definition
  verankerung2: IF n=1 GENERATE
    D1: two_phases PORT MAP (x=>x(0), y=>y(1), yz=>y(0));
  END GENERATE;
  rekursion: IF n>1 GENERATE
    RDEC: decoder -- Aufruf der Rekursion
      GENERIC MAP (n-1)
        PORT MAP (x=>x(0 TO n-2), y=>r);
    LBIT: two_phases
      PORT MAP (x=>x(n-1), y=>1, yz => 1z);
    ANDGATES: FOR i IN 0 TO 2**(n-1)-1
      GENERATE
        AND0: and2
          PORT MAP (x=>r(i), y=>1z, z=>y(2*i));
        AND1: and2
          PORT MAP (x=>r(i), y=>1, z=>y(2*i+1));
      END GENERATE;
    END GENERATE;
  END ARCHITECTURE;
```

Konstruktion eines Linksshifters

Als weiteres Beispiel soll uns ein Linksshifter dienen:

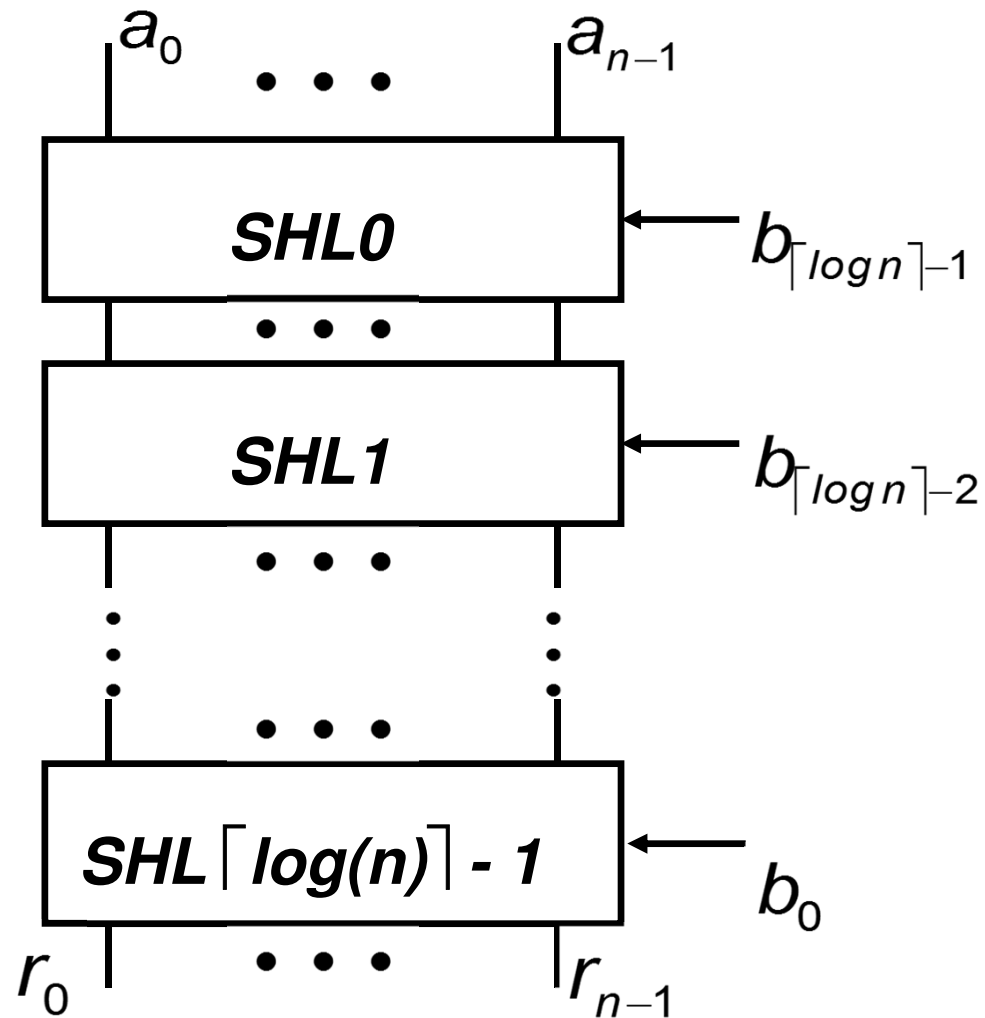
Ein Operand a habe Wortbreite n , in einem weiteren Operanden sei die Shiftweite i als unsigned Zahl kodiert, d.h. es sei

$$0 < u_{[\log n]}(b) < n \quad \text{und} \quad b = (b_0, \dots, b_{[\log n]-1})$$

Eine 1 auf Position $\log n - i$ bedeutet, dass wir um 2^{i-1} Stellen schieben müssen, und insgesamt um alle Zweierpotenzen, die auf den Positionen vorgegeben sind.

Wir erhalten also einen Shifter, indem wir für jede Zweierpotenz einen Shifter **SHLi** bereitstellen, der schiebt, wenn diese mit 1 besetzt ist, und dann diese Bausteine hintereinander schalten:

Linksshifter -- Grundstruktur



Grundstruktur ff

Berechnet nun der Baustein $SHLi$ die Funktion

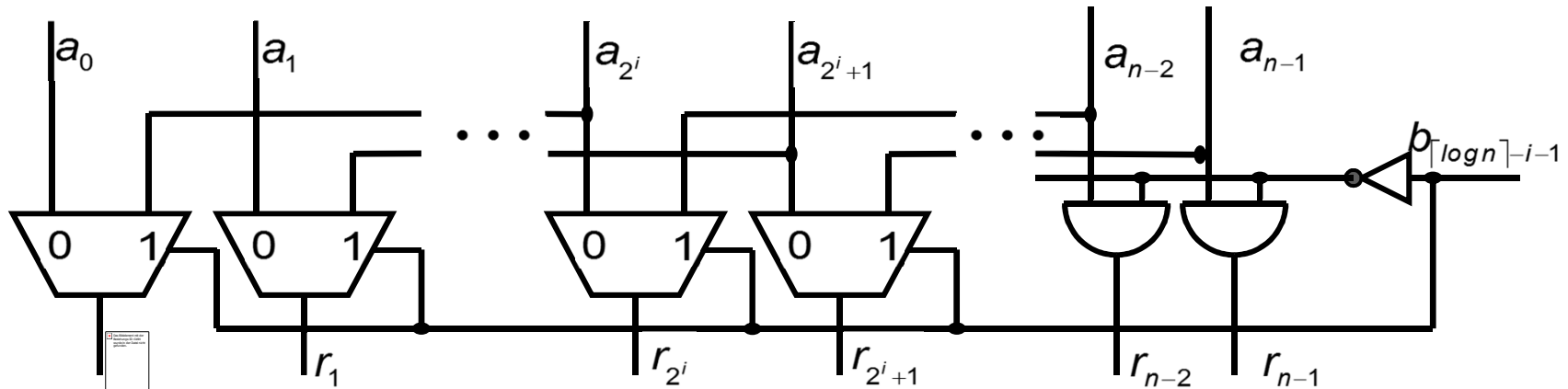
$$shl_i(a, s) = SHL(a, s \cdot 2^i), \quad s \in \mathbf{B}$$

dann berechnet die Schaltung die Funktion

$$\begin{aligned} & shl_{\lceil \log n \rceil - 1}(shl_{\lceil \log n \rceil - 2}(\dots(shl_0(a, b_{\lceil \log n \rceil - 1}), \dots), b_1), b_0) \\ &= SHL(SHL(\dots(SHL(a, b_{\lceil \log n \rceil - 1} 2^0), \dots), b_1 2^{\lceil \log n \rceil - 2}), b_0 2^{\lceil \log n \rceil - 1}) \\ &= SHL(a, \sum_{i=0}^{\lceil \log n \rceil - 1} b_{\lceil \log n \rceil - i - 1} 2^i) \\ &= SHL(a, 2^{\lceil \log n \rceil - 1} \sum_{i=0}^{\lceil \log n \rceil - 1} b_i 2^{-i}) = SHL(a, u_{\lceil \log n \rceil}(b)) \end{aligned}$$

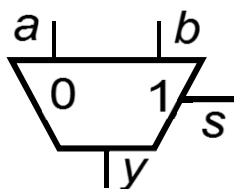
Shift um eine Zweierpotenz

Ein einzelner SHL_i Baustein hat folgende einfache Struktur:



Dabei verwenden wir diesmal einen (gerichteten)

Multiplexer

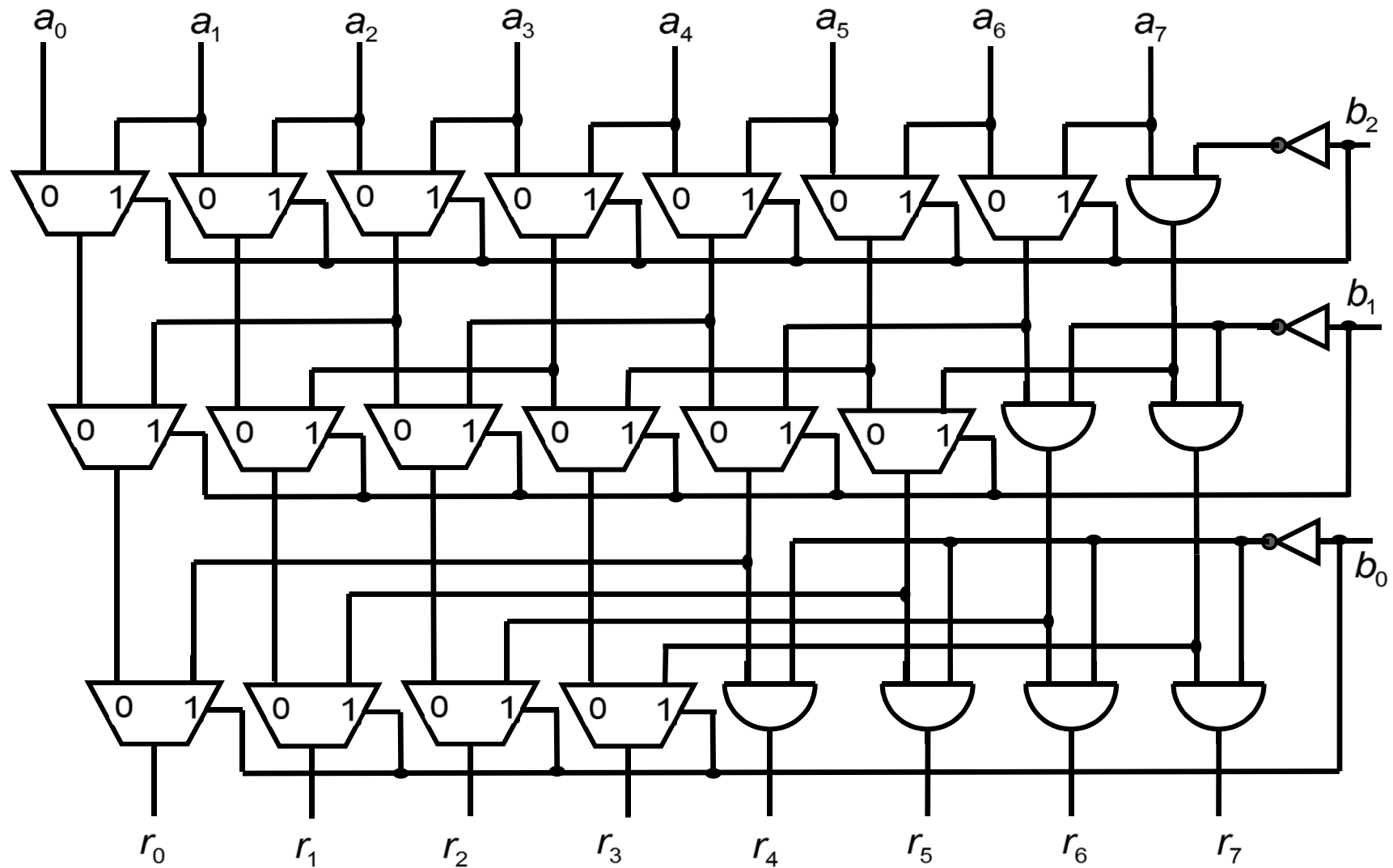


$$\text{Funktion: } y = a \cdot \bar{s} \vee b \cdot s$$

$$y = (\text{if } s \text{ then } b \text{ else } a) = \text{ite}(s, b, a)$$

Wir werden solche Multiplexer auch auf Wortbreite (vgl. Übungen) nutzen.

Beispiel: 8-Bit Linksshifter



Generische Definition in WüHDL

```
ENTITY lshifter IS
    GENERIC ( n,k : POSITIVE); -- k <= log n
    PORT (x: IN BIT_VECTOR(0 TO n-1);
          w: IN BIT_VECTOR(0 TO k-1);
          y: OUT BIT_VECTOR(0 TO n-1) );
END ENTITY lshifter;

ARCHITECTURE structure OF lshifter IS
    COMPONENT lshifter -- zur Rekursion
        GENERIC ( n,k: POSITIVE );
        PORT (x: IN BIT_VECTOR(0 TO n-1);
              w: IN BIT_VECTOR(0 TO k-1);
              y: OUT BIT_VECTOR(0 TO n-1) );
    END COMPONENT;
    COMPONENT MUX2 PORT (x0,x1,sel:IN BIT; y:OUT BIT);
    END COMPONENT;
    COMPONENT inverter PORT(x:IN BIT;y:OUT BIT);
    END COMPONENT;
```

Generische Definition in WüHDL -- ff

```
    COMPONENT and2 PORT (x,y:IN BIT;z:OUT BIT);
    END COMPONENT;
SIGNAL r: BIT_VECTOR(0 TO n-1);
SIGNAL wz : BIT;

BEGIN --rekursive Definition
    verankerung:IF k=1 GENERATE
        shiftbyone: FOR i IN 0 TO n-2
            GENERATE
                MUXES: mux2
                    PORT MAP (x0=>x(i), x1=>x(i+1),
                               sel=>w(0), y=>y(i));
            END GENERATE;
        fill:and2
            PORT MAP (x=>x(n-1), y=>wz, z=>y(n-1));
        Inv: inverter PORT MAP (x=>w(0), y=>wz);
    END GENERATE;
```


Generische Definition in WüHDL -- ff

```
rekursion: IF k>1 GENERATE
    shifter:lshifter GENERIC MAP (n=>n,k=>k-1)
        PORT MAP (x=>x,w=>w(1 TO k-1),y=>r);
    laststage: FOR i IN 0 TO n-1-2**(k-1)
        GENERATE
            MUXES: mux2
                PORT MAP (x0=>r(i),x1=>r(i+2**(k-1)),
                    sel=>w(0),y=>y(i));
            END GENERATE;
        lastfill: FOR i IN n-2**(k-1) TO n-1
            GENERATE
                ANDGATES: and2
                    PORT MAP (x=>r(i), y=>wz, z=>y(i));
                END GENERATE;
                Inv1: inverter PORT MAP (x=>w(0),y=>wz);
            END GENERATE;
        END ARCHITECTURE;
```

Addierer

Wir wollen zunächst die Addition von vorzeichenlosen Zahlen betrachten, d.h. wir wollen eine Funktionsscheibe **ADDU** entwickeln, die für zwei Worte a, b der Länge n als Eingänge und ein Wort s der Länge n als Ausgang, sowie ein Flag ov als weiteren Ausgang die folgende Funktion *addu* realisiert:

$addu: \mathbf{B}^{2n} \mapsto \mathbf{B}^{n+1}$ mit

$$addu(a, b) = (ov, s) \iff 2^n ov + u(s) = u(a) + u(b)$$

Da $u(s) \leq 2^n - 1$ zeigt das Flag ov offenbar an, dass die Summe nicht mehr in einem Maschinenwort dargestellt werden kann, da sie größer als $2^n - 1$ ist. Wir nennen dieses Flag auch die **Überlaufanzeige** (overflow flag)

Volladdierer und Halbaddierer

Wir betrachten die Funktion eines sogenannten Volladdierers

$$fa: \mathbf{B}^3 \mapsto \mathbf{B}^2 \text{ wobei } fa(x_1, x_2, x_3) = (c, s) \Leftrightarrow 2c + s = x_1 + x_2 + x_3$$

Man kann den Volladdierer direkt durch eine entsprechende Schaltung realisieren:

c	$0 \ 0 \ 1 \ 1$	x_1	s	$0 \ 0 \ 1 \ 1$	x_1
x_2	$0 \ 1 \ 0 \ 1$	x_2	x_2	$0 \ 1 \ 0 \ 1$	x_2
0	$0 \ 0 \ 0 \ 1$		0	$0 \ 1 \ 1 \ 0$	
1	$0 \ 1 \ 1 \ 1$		1	$1 \ 0 \ 0 \ 1$	
x_3	$c = x_3(x_1 \vee x_2) \vee x_1x_2$		x_3	$s = x_3(x_1 \equiv x_2) \vee \bar{x}_3(x_1 \oplus x_2)$	

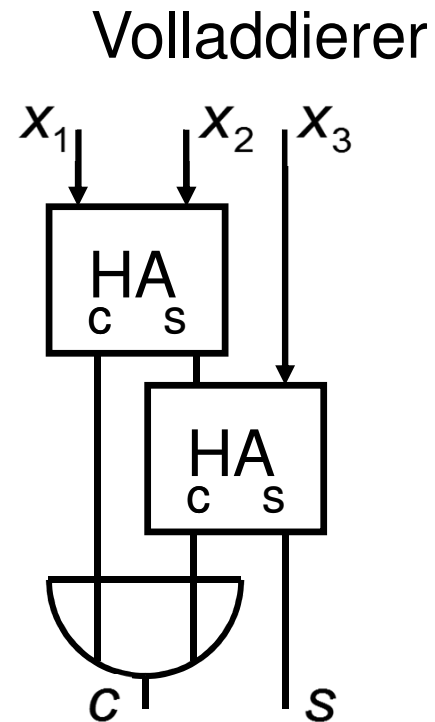
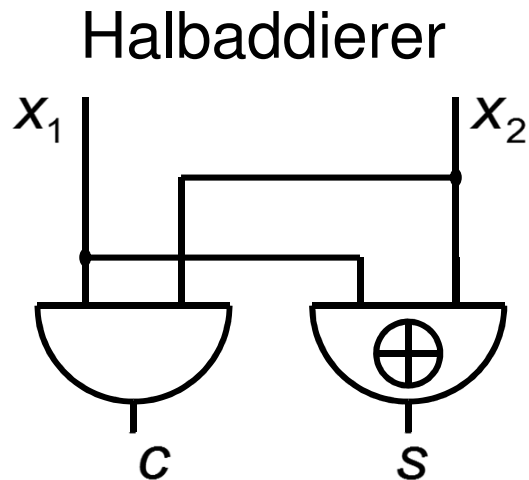
oder unter Zuhilfenahme zweier **Halbaddierer** (half adder), die nur die Summe zweier Bits als zweistellige Zahl kodieren können:

$$ha: \mathbf{B}^2 \mapsto \mathbf{B}^2 \text{ wobei } ha(x_1, x_2) = (c, s) \Leftrightarrow 2c + s = x_1 + x_2$$

c	$0 \ 1$	x_1	s	$0 \ 1$	x_1
0	$0 \ 0$		0	$0 \ 1$	
1	$0 \ 1$	$c = x_1x_2$	1	$1 \ 0$	$s = x_1 \oplus x_2$
x_2			x_2		

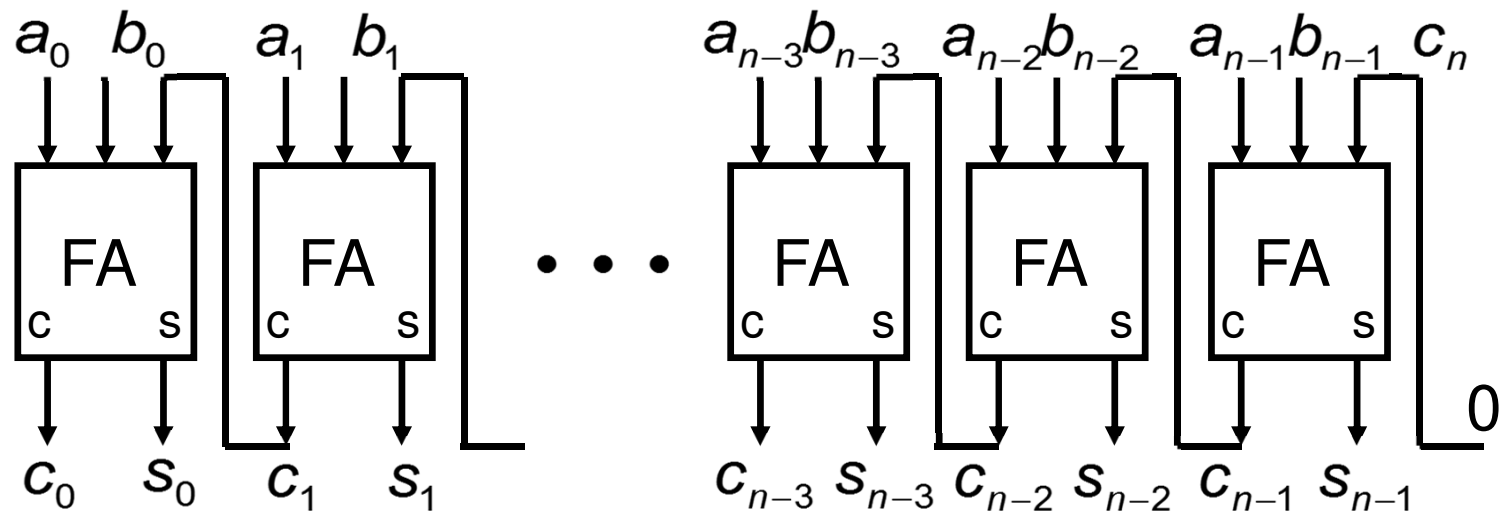
Volladdierer und Halbaddierer ff

Damit ergäbe sich z.B. folgende Schaltung:



Der ripple carry Addierer

Man erhält nun auf ganz einfache Weise einen Schaltkreis zur Addition, wenn man die uns allen bekannte Grundschulmethode auf das Binärsystem überträgt, und von den weniger signifikanten Stellen zur signifikanten Stelle hin die Ziffern jeweils mit Volladdierern addiert und einen entstehenden Übertrag weiterleitet:



Wir nennen diese Schaltung den **ripple carry Adder**.

Ripple carry Adder ff

Der Volladdierer auf Position $n - 1$ kann durch einen Halbaddierer ersetzt werden, weil ein Eingang (carry in) immer 0 ist, oder er kann dazu benutzt werden auch $u(a) + u(b) + 1$ zu berechnen (carry in = 1).

Allerdings ist die Verzögerungszeit dieses Addierers enorm hoch, da im schlimmsten Fall der Übertrag durch n Volladdiererstufen läuft:

Kann man schneller als seriell addieren?

Der conditional sum adder

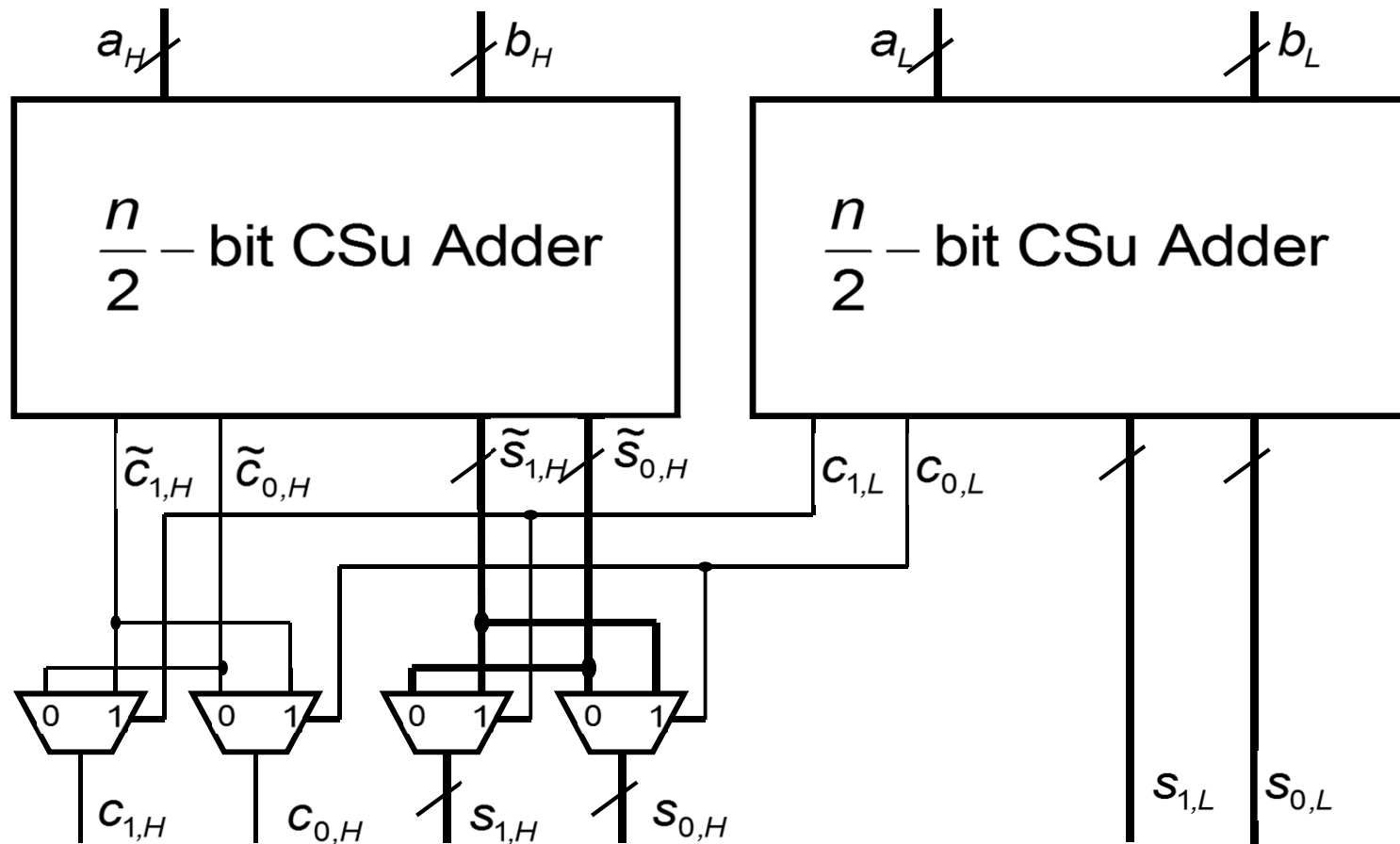
Diese Idee stammt von Sklansky und ist eines der Paradebeispiele dafür, dass man eine zunächst streng sequentiell anmutende Aufgabe schneller parallel lösen kann, wenn man etwas mehr tut:

Idee:

Berechne parallel die Summe und die Summe plus eins für die signifikantere Hälfte und die weniger signifikante Hälfte. Wähle dann in einem Schlag in Abhängigkeit vom Übertrag der weniger signifikanten Hälfte die richtigen Varianten der Summen aus.

Dies führt zu folgendem rekursiven Schema:

Conditional sum adder ff



Conditional sum adder ff

Dabei seien: $a_H := a[0 : \frac{n}{2} - 1]; \quad a_L := a[\frac{n}{2} : n - 1];$
 $b_H := b[0 : \frac{n}{2} - 1]; \quad b_L := b[\frac{n}{2} : n - 1];$

Für den einen Addierer gelte

$$u_{\frac{n}{2}}(s_{0,L}) + 2^{\frac{n}{2}} c_{0,L} = u_{\frac{n}{2}}(a_L) + u_{\frac{n}{2}}(b_L);$$
$$u_{\frac{n}{2}}(s_{1,L}) + 2^{\frac{n}{2}} c_{1,L} = u_{\frac{n}{2}}(a_L) + u_{\frac{n}{2}}(b_L) + 1;$$

Und für den identischen anderen Addierer ebenfalls

$$u_{\frac{n}{2}}(\tilde{s}_{0,H}) + 2^{\frac{n}{2}} \tilde{c}_{0,H} = u_{\frac{n}{2}}(a_H) + u_{\frac{n}{2}}(b_H);$$
$$u_{\frac{n}{2}}(\tilde{s}_{1,H}) + 2^{\frac{n}{2}} \tilde{c}_{1,H} = u_{\frac{n}{2}}(a_H) + u_{\frac{n}{2}}(b_H) + 1;$$

Conditional sum adder ff

Das Verfahren ist korrekt, denn

$$u_n(a) + u_n(b)$$

$$= 2^{\frac{n}{2}} (u_{\frac{n}{2}}(a_H) + u_{\frac{n}{2}}(b_H)) + u_{\frac{n}{2}}(a_L) + u_{\frac{n}{2}}(b_L)$$

$$= 2^{\frac{n}{2}} (u_{\frac{n}{2}}(a_H) + u_{\frac{n}{2}}(b_H)) + 2^{\frac{n}{2}} c_{0,L} + u_{\frac{n}{2}}(s_{0,L})$$

$$= \begin{cases} 2^{\frac{n}{2}} (u_{\frac{n}{2}}(a_H) + u_{\frac{n}{2}}(b_H)) + u_{\frac{n}{2}}(s_{0,L}) & \text{falls } c_{0,L} = 0 \\ 2^{\frac{n}{2}} (u_{\frac{n}{2}}(a_H) + u_{\frac{n}{2}}(b_H) + 1) + u_{\frac{n}{2}}(s_{0,L}) & \text{falls } c_{0,L} = 1 \end{cases}$$

$$= \begin{cases} 2^n c_{0,H} + u_n(\tilde{s}_{0,H}, s_{0,L}) & \text{falls } c_{0,L} = 0 \\ 2^n c_{1,H} + u_n(\tilde{s}_{1,H}, s_{0,L}) & \text{falls } c_{0,L} = 1 \end{cases} \quad \text{Ergebnis lt. Verdrahtung}$$

Conditional sum adder ff

Analog rechnet man nach, dass

$$u_n(a) + u_n(b) + 1 = \begin{cases} 2^n c_{0,H} + u_n(\tilde{s}_{0,H}, s_{1,L}) & \text{falls } c_{1,L} = 0 \\ 2^n c_{1,H} + u_n(\tilde{s}_{1,H}, s_{1,L}) & \text{falls } c_{1,L} = 1 \end{cases}$$

Werden die Teile rekursiv genauso realisiert, berechnet die Schaltung sowohl die Summe, als auch die Summe plus 1 parallel in Tiefe

$$\begin{aligned} T_{CSU}(n) &= T_{CSU}\left(\frac{n}{2}\right) + d(MUX) \\ &= \dots \\ &= T_{CSU}(1) + \log n \cdot d(MUX) \end{aligned}$$

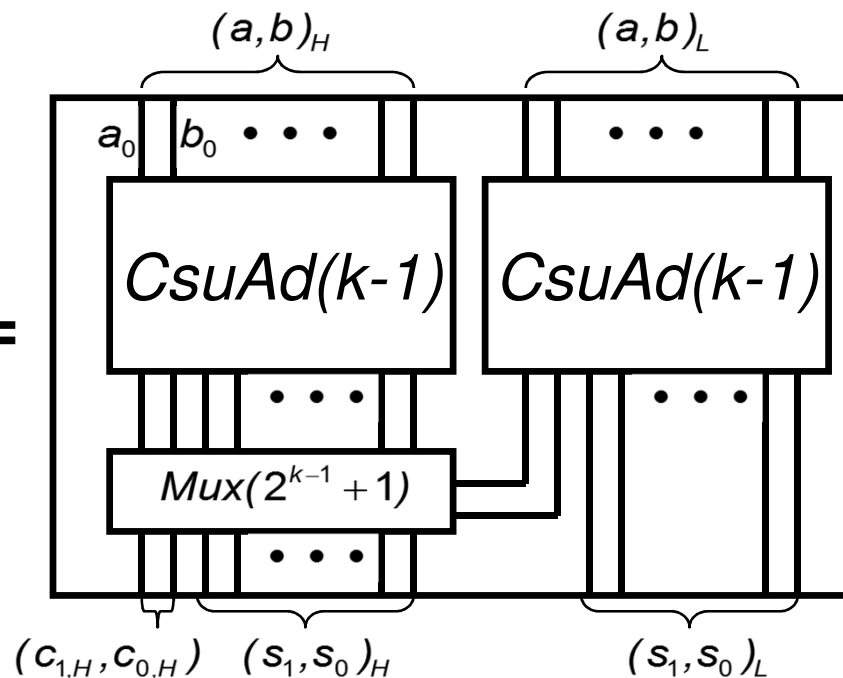
Die Tiefe wächst also nur logarithmisch mit n .

Conditional sum adder ff

Man kann nach diesem rekursiven Bildungsgesetz nun auch leicht einen übersichtlichen Schaltplan definieren. Dazu ist es günstig, die Operanden- und Ergebnisleitungen wieder bitweise gemischt zu führen, d.h. in der Reihenfolge \dots, a_i, b_i, \dots sowie $c_{1,H}, c_{0,H}, s_{1,0}, s_{0,0}, \dots, s_{1,n-1}, s_{0,n-1}$

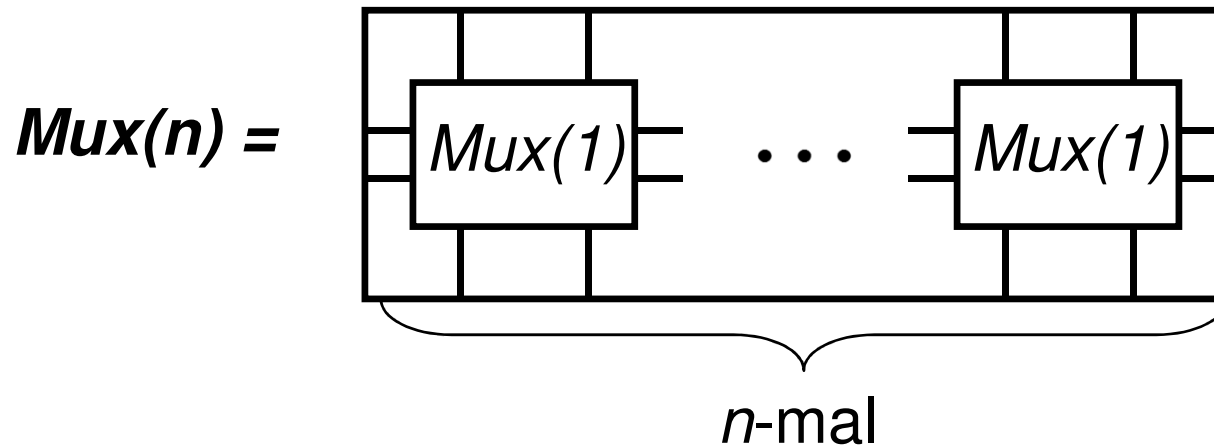
Rekursion für $k = \log n$:

CsuAd(k) =

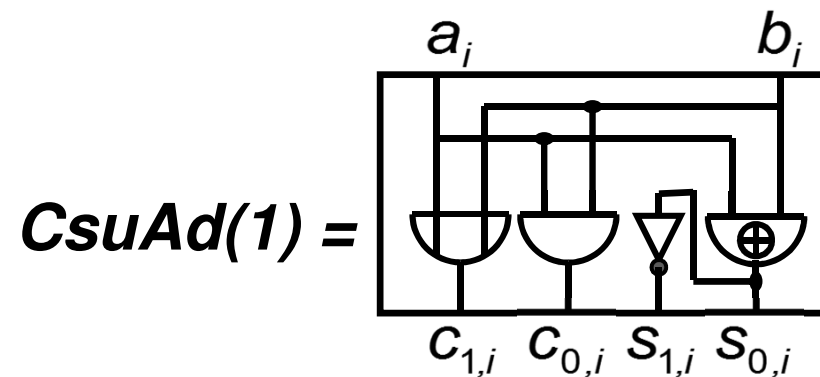
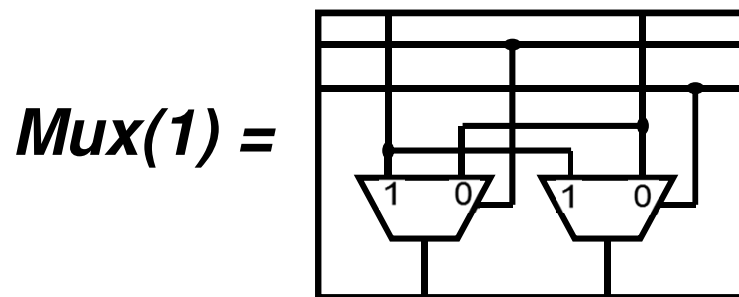


Conditional sum adder ff

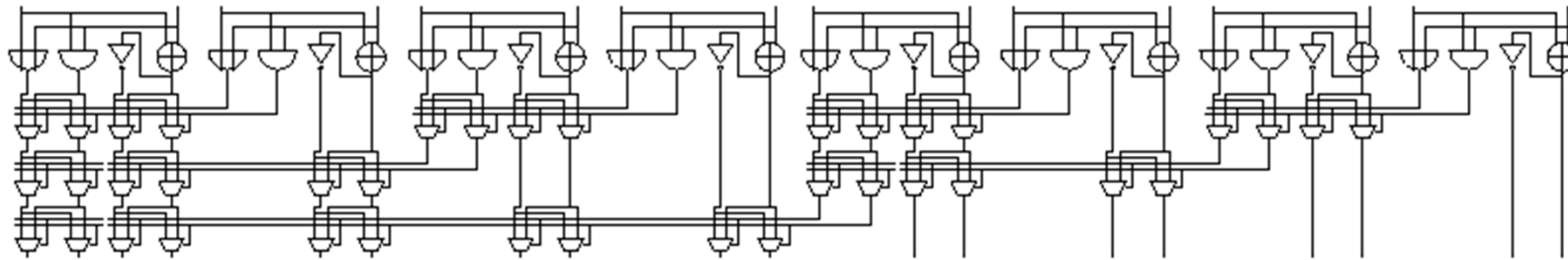
Der Auswahlbaustein wird durch einfaches Nebeneinander-
setzen realisiert:



Und für ein Bitpaar haben wir

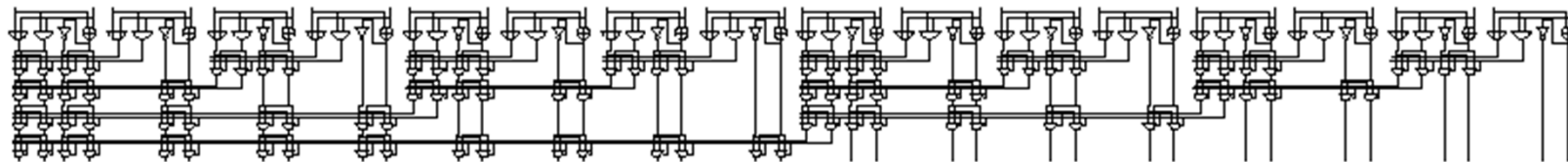


Beispiele:



8 bit Addierer

16 bit Addierer



Es gibt eine Fülle von Möglichkeiten, schnelle Addierer zu konstruieren. Wir stellen diese zurück und tun so, als hätten wir brauchbare Schaltungen.

Generische Definition in WüHDL

```
-- Definition des Grundbausteins fuer 1 Bit Breite
ENTITY simple_adder IS
    PORT ( a,b    :  IN BIT;
           s0,s1  :  OUT BIT;
           c0,c1  :  OUT BIT  );
END simple_adder;

ARCHITECTURE behavior OF simple_adder IS
BEGIN
    c0 <= a AND b;  s0 <= a XOR b;
    c1 <= a OR b;  s1 <= NOT(a XOR b);
END behavior;
```

Generische Definition in WüHDL -- ff

-- Definition des Auswahlbausteins

```
ENTITY auswahl IS
  GENERIC ( n: POSITIVE );
  PORT ( x0,x1: IN BIT_VECTOR(0 TO n-1);
         sel0,sel1: IN BIT;
         s0,s1: OUT BIT_VECTOR(0 TO n-1));
END auswahl;

ARCHITECTURE behavior OF auswahl IS
  BEGIN
    s0 <= x0 WHEN sel0 = '0' ELSE x1;
    s1 <= x0 WHEN sel1 = '0' ELSE x1;
  END behavior;
```


Generische Definition in WüHDL -- ff

```
-- CSU Addierer: Rekursive Definition der Struktur
ENTITY csu_adder IS
    GENERIC ( n : POSITIVE);
    PORT ( a, b      : IN  BIT_VECTOR(0 TO n-1);
           s0, s1    : OUT BIT_VECTOR(0 TO n-1);
           c0, c1    : OUT BIT );
END csu_adder;

ARCHITECTURE structure OF csu_adder IS
    COMPONENT csu_adder ...
    COMPONENT simple_adder ...
    COMPONENT auswahl ...
    SIGNAL msb0, msb1 : BIT_VECTOR(0 TO n-n/2-1);
    SIGNAL msbc0, msbc1 : BIT;
    SIGNAL carry0, carry1 : BIT;

BEGIN -- csuadder structure
    verankerung: IF n=1 GENERATE
        anker: simple_adder
            PORT MAP ( a => a(0), b => b(0), s0 => s0(0), s1 => s1(0),
                      c0 => c0, c1 => c1);
    END GENERATE;
END structure;
```

Generische Definition in WüHDL -- ff

rekursion:

```
IF n>=2 GENERATE
  adder_msb: csu_adder -- 1. Aufruf der Rekursion
  GENERIC MAP (n-n/2)
  PORT MAP (a => a(0 TO n-n/2-1), b => b(0 TO n-n/2-1),
            s1 => msb1, s0 => msb0,
            c1 => msbc1, c0 => msbc0    );
  adder_lsb: csu_adder -- 2. Aufruf der Rekursion
  GENERIC MAP (n/2)
  PORT MAP (a => a(n-n/2 TO n-1), b => b(n-n/2 TO n-1),
            s1 => s1(n - n/2 TO n-1), c1 => carry1,
            s0 => s0(n - n/2 TO n-1), c0 => carry0    );
  selektion: auswahl GENERIC MAP ( n - n/2 )
  PORT MAP (x0 => msb0, x1 => msb1,
            sel0 => carry0, sel1 => carry1,
            s0 => s0(0 TO n-n/2-1),
            s1 => s1(0 TO n-n/2-1)          );
  selektion_carry: auswahl GENERIC MAP (1)
  PORT MAP (x0(0) => msbc0, x1(0) => msbc1,
            sel0 => carry0, sel1 => carry1,
            s0(0) => c0, s1(0) => c1          );
END GENERATE;
```

END structure;
