Rechneraufbau und Rechnerstrukturen

Von
Walter Oberschelp,
Gottfried Vossen

10., überarbeitete und erweiterte Auflage

Oldenbourg Verlag München Wien

Prof. Dr. Gottfried Vossen lehrt seit 1993 Informatik am Institut für Wirtschaftsinformatik der Universität Münster. Er studierte, promovierte und habilitierte sich an der RWTH Aachen und war bzw. ist Gastprofessor u.a. an der University of California in San Diego, USA, an der Karlstad Universitet in Schweden, an der University of Waikato in Hamilton, Neuseeland sowie am Hasso-Plattner-Institut für Softwaresystemtechnik in Potsdam. Er ist europäischer Herausgeber der bei Elsevier erscheinenden Fachzeitschrift Information Systems.

Prof. Dr. Walter Oberschelp studierte Mathematik, Physik, Astronomie, Philosophie und Mathematische Logik. Nach seiner Habilitation in Hannover lehrte er als Visiting Associate Professor an der University of Illinois (USA). Nach seiner Rückkehr aus den USA übernahm er den Lehrstuhl für Angewandte Mathematik an der RWTH Aachen, den er bis zu seiner Emeritierung im Jahr 1998 inne hatte.

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
www.oldenbourg-wissenschaftsverlag.de

Lektorat: Margit Roth
Herstellung: Anna Grosser
Umschlagkonzeption: Kraxenberger Kommunikationshaus, München
Gedruckt auf säure- und chlorfreiem Papier
Druck: Oldenbourg Druckerei Vertriebs GmbH & Co. KG
Bindung: R. Oldenbourg Graphische Betriebe Binderei GmbH

ISBN 3-486-57849-9
ISBN 978-3-486-57849-2

Kapitel 2

Multiplexer und Addiernetze als spezifische Schaltnetze

In Kapitel 1 haben wir Möglichkeiten kennen gelernt, das „Innenleben" einer Black Box, deren Verhalten durch eine Boolesche Funktion beschreibbar ist, aus einfachen Grundgattern für die Operatoren $+$, $\cdot$ und $^-$ bzw. aus NAND- oder NOR-Gattern aufzubauen. Allgemeine Prinzipien zur Synthese von Schaltnetzen lieferten die verschiedenen Darstellungssätze, bei denen es sich allerdings um eher theoretische Hilfsmittel handelt. In diesem Kapitel werden wir uns zur *Synthese* von Schaltnetzen einen anderen Zugang verschaffen: Wir werden Entwurfsverfahren kennen lernen, mit deren Hilfe man in speziellen Situationen zu einfacheren Schaltnetzen gelangt als über die verschiedenen Normalformen. Daneben stellen wir Standard-Bausteine vor, mit welchen sich Boolesche Funktionen realisieren lassen. Von besonderer Bedeutung, auch für später anzustellende Überlegungen, sind dabei Addiernetze; für diese betrachten wir unter anderem das Problem der schnellen Berechnung eines Additionsübertrags und damit das Problem der Beschleunigung.

2.1 Vorüberlegungen zur Synthese von Schaltnetzen

Zum Entwurf von Schaltnetzen gibt es grundsätzlich zwei Techniken: Beim *Bottom-Up-Entwurf* werden komplexe Schaltungen aus elementaren Bausteinen sukzessive zusammengesetzt. Ein *Top-Down-Entwurf* beginnt mit einer Zerlegung der gesamten Entwurfsaufgabe in wohldefinierte Teilaufgaben, welche unter Umständen mehrfach weiter verfeinert werden; die gewünschte Schaltung ergibt sich aus einer Realisierung der Komponenten der feinsten Verfeinerungsstufe.

Die Idee des Bottom-Up-Entwurfs findet bereits dann Anwendung, wenn aus den in Kapitel 1 eingeführten Gattern komplexere Schaltungen aufgebaut werden. Diese können dann ihrerseits als neue „Grundbausteine" betrachtet und verwendet werden. Wenn man also beispielsweise von einer Normalformendarstellung ausgeht oder eine Schaltung aus NANDs bzw. NORs zusammensetzt, kann man die so realisierte Funktion als neuen Baustein bei der Realisierung einer komplexeren Funktion verwenden.

Tabelle 2.1: Klassifikation von Graphen mit 5 Punkten und Euler-Kreis.

Typ \ Ecke	1	2	3	4	5
a	2	2	2	2	2
b	4	2	2	2	2
c	4	4	2	2	2
d	4	4	4	4	4

Ein solches Zusammenfassen gewisser Module zu neuen bezeichnet man als *„Integration"*, und es sei darauf hingewiesen, dass heute nahezu jeder (auch hoch integrierte) Baustein als Chip kurzfristig herzustellen ist. Auf einem Chip können heute sogar mehrere Millionen Bauteile untergebracht werden, was generell als VLSI (Very Large Scale Integration) bezeichnet wird.

Als nächstes kommen wir, als Beispiel für den Bottom-Up-Entwurf, auf Beispiel 1.9 zurück und wollen für das dort geschilderte Problem spezielle Lösungsbausteine entwerfen: Gesucht ist ein Schaltnetz zur Realisierung der Booleschen Funktion e, welches also zu 10 gegebenen Inputs entscheidet, ob der dadurch codierte Graph einen Euler-Kreis besitzt oder nicht. Zur Lösung ziehen wir das aus der Graphentheorie bekannte „Euler-Kriterium" heran: Ein (zusammenhängender) Graph besitzt einen Euler-Kreis genau dann, wenn jeder Punkt des Graphen einen geraden Grad hat. (Dabei ist der Grad eines Punktes gleich der Anzahl der Kanten, die an ihm zusammen treffen.)

Da wir nur Graphen mit 5 Punkten betrachten, welche zusätzlich als zusammenhängend vorausgesetzt werden (insbesondere hat damit keine Ecke den Grad 0), kommen hier nur die Gradzahlen 2 und 4 in Frage. Daher können wir uns leicht eine Übersicht über alle Graphen mit 5 Punkten verschaffen, die einen Euler-Kreis besitzen. Wir geben dazu die Gradzahl jeder Ecke in Tabelle 2.1 an. In dieser Tabelle bedeutet z. B. Zeile 1, dass alle Ecken den Grad 2 haben; ein solcher Graph (vom Typ a) hat die in Abbildung 2.1 (a) gezeigte Gestalt. Entsprechend bedeutet Zeile 2, dass eine Ecke den Grad 4 hat und alle anderen den Grad 2 (vgl. Abbildung 2.1 (b)). Man beachte, dass es in diesen Fällen wie auch bei (c) und (d) jeweils mehrere Graphen gibt, welche diese Gestalt haben, denn mit jeder neuen Nummerierung der Ecken erhält man im Prinzip einen neuen Graphen. Für (c) und (d) erhält man analog zu oben die in Abbildung 2.1 (c) bzw. (d) gezeigten Gestalten.

Man sieht unmittelbar, dass in Tabelle 2.1 die Zeilen (4, 4, 4, 2, 2) und (4, 4, 4, 4, 2) nicht auftreten können. Wir erläutern dies nur für die erste dieser beiden (für die andere verläuft die Argumentation völlig analog): Haben in einem Graph mit 5 Punkten 3 Ecken den Grad 4, d. h. sie sind mit allen anderen Punkten verbunden, so liegt mindestens die in Abbildung 2.1 (e) gezeigte Situation vor: also haben die verbleibenden Ecken mindestens den Grad 3.

Für das zu entwerfende Schaltnetz bedeuten diese Betrachtungen: Einerseits reicht es zu testen, ob ein gegebener Input das Euler-Kriterium bzw. die spezifischen Folgerungen daraus für Graphen mit 5 Punkten erfüllt, andererseits können die oben genannten „unmöglichen" Fälle mit in den Entwurf einbezogen werden, da entsprechende Inputs nicht auftreten können. Wir brauchen uns also nicht darum zu kümmern,

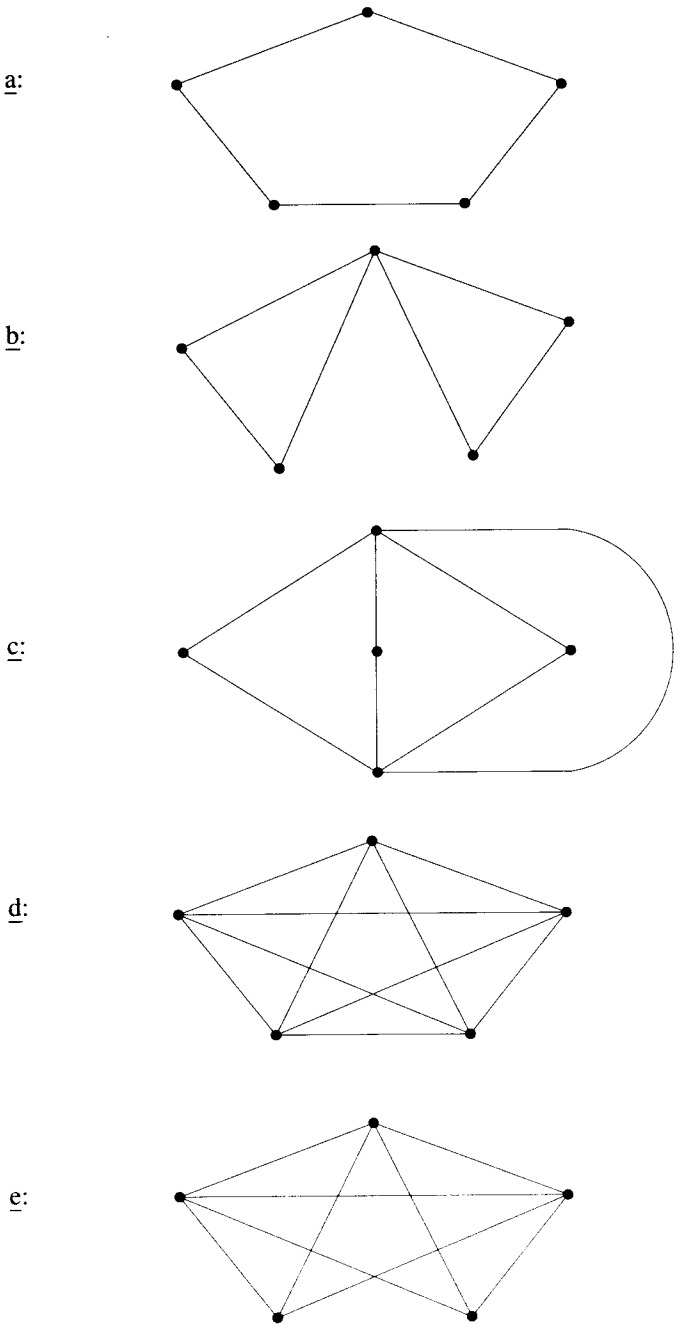

Abbildung 2.1: (a) (d) Graphen mit 5 Punkten, welche einen Euler-Kreis enthalten; (e) enthält keinen Euler-Kreis.

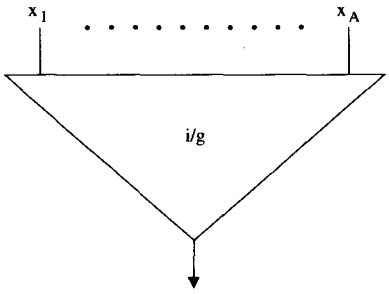

Abbildung 2.2: Symbol für Baustein zum Test einer Ecke auf geraden Grad.

welche Outputs hier herauskommen. Das in dieser „Don't Care"-Überlegung enthaltene Vereinfachungspotenzial bewirkt im konkreten Fall, dass zu einem Input bestehend aus 10 Bits lediglich zu testen ist, ob jede Ecke des Graphen Grad 2 oder 4 hat. Dazu entwerfen wir einen neuen Baustein mit der in Abbildung 2.2 gezeigten Bezeichnung, welcher eine 1 als Output liefern soll, wenn der Grad der Ecke i gleich 2 oder 4 ist. Für die Ecke 1 geben wir diesen Baustein explizit an:

Ecke 1 hat Grad 2 oder 4

$\Longleftrightarrow$ G enthält die Kanten (k_1 und k_2 und nicht $k_3 \ldots$ und nicht k_A)
oder (k_1 und $k_3 \ldots$)
oder (k_1 und $k_4 \ldots$)
oder (k_2 und $k_3 \ldots$)
oder (k_2 und $k_4 \ldots$)
oder (k_3 und $k_4 \ldots$)
oder (k_1 und k_2 und k_3 und $k_4 \ldots$)

$\Longleftrightarrow$ die Anzahl der Einsen im „Input-Bereich" $x_1, \ldots, x_4$ ist positiv und gerade

$$\Longleftrightarrow \overline{x_1 \dotplus x_2 \dotplus x_3 \dotplus x_4} \wedge (x_1 \vee x_2 \vee x_3 \vee x_4)$$

Für $\dotplus$ setzen wir bereits einen eigenen Baustein ein, welcher in Abbildung 2.3 gezeigt ist. Wie in Kapitel 1 bereits erwähnt, fassen wir auch jetzt identische Bausteine mit 2 Eingängen (unter Verwendung des Assoziativgesetzes; hier Satz 1.14 (d)) zu einem Bauteil mit mehreren Eingängen zusammen, so dass wir die in Abbildung 2.4 gezeigte Darstellung erhalten (zur Vereinfachung versehen wir alle Bauteile mit 10 Inputs). Mit Hilfe dieses neuen Bausteins erhalten wir nun sofort das gesuchte Schaltnetz, welches in Abbildung 2.5 gezeigt ist.

2.2 Multiplexer zur Realisierung Boolescher Funktionen

Als Alternative zu der im letzten Abschnitt beschriebenen Methode des Bottom-Up-Entwurfs von Schaltnetzen betrachten wir in diesem Abschnitt die Vorgehensweise des Top-Down-Entwurfs, und zwar am Beispiel des Multiplexers. Wir stellen damit

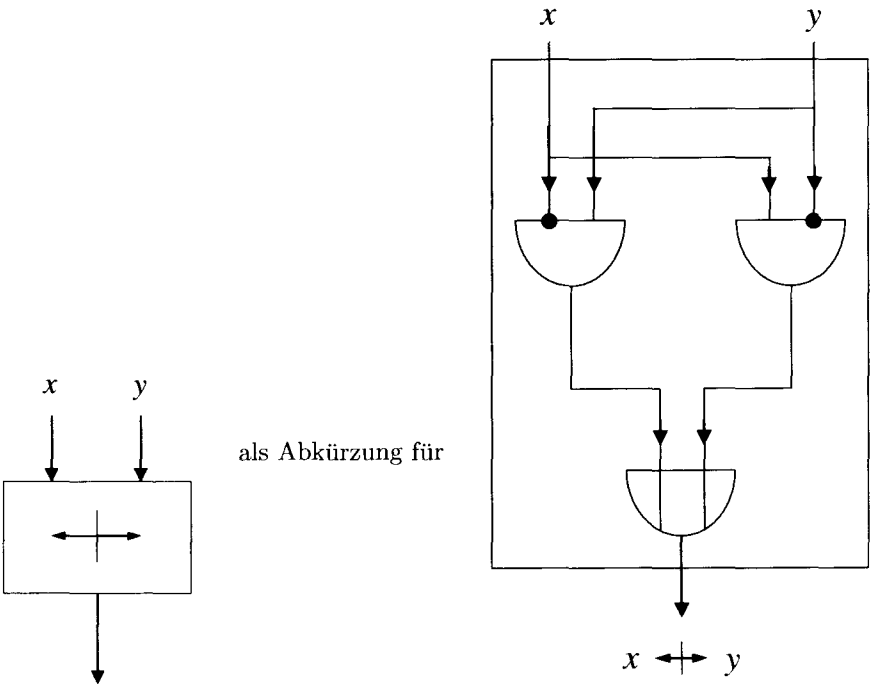

Abbildung 2.3: Baustein zur Realisierung von $\dashv\vdash$.

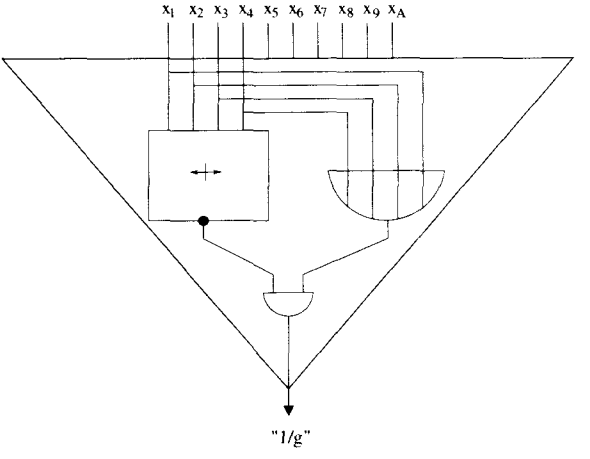

Abbildung 2.4: Baustein zum Test der Ecke 1 auf geraden Grad.

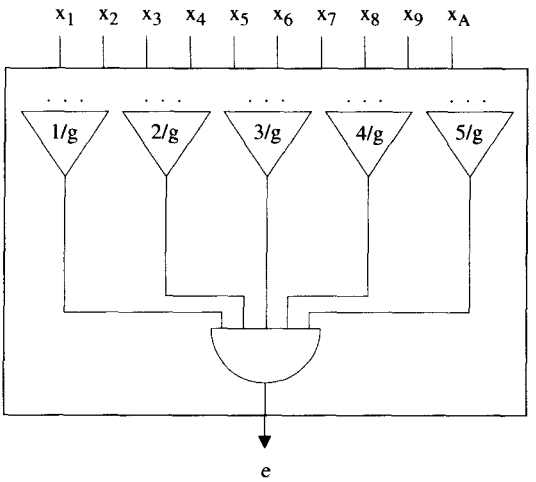

Abbildung 2.5: Schaltnetz zur Realisierung der Funktion e aus Beispiel 1.9.

gleichzeitig einen ersten Standard-Baustein zur Realisierung Boolescher Funktionen vor: weitere derartige Bausteine, die wir ebenfalls in diesem Abschnitt behandeln, sind der Demultiplexer, der Decoder und der Encoder.

Ein *Multiplexer*, abgekürzt MUX, ist ein häufig verwendetes Selektionsschaltnetz, welches als Input 2^d Daten-Inputs $x_0, \ldots, x_{2^d-1}$ sowie d Steuersignale $y_1, \ldots, y_d$ hat, und bei welchem an dem einzigen Output z genau einer der Daten-Inputs in Abhängigkeit von den Steuersignalen erscheint; es wird also ein Daten-Input als Output selektiert. Als erstes Beispiel zeigt Abbildung 2.6 einen MUX mit $d = 2$ Steuersignalen und $2^d = 4$ Daten-Inputs, dessen Funktionalität durch die folgende Tabelle beschrieben wird:

y_1	y_2	z
0	0	x_0
0	1	x_1
1	0	x_2
1	1	x_3

Abbildung 2.7 zeigt den allgemeinen Aufbau eines MUX. Man beachte, dass die Steuersignale für jede feste Belegung eine Dualzahl zwischen 0 und $2^d - 1$ darstellen: der Output eines MUX ist damit jeweils gleich dem Input $x_{(y_1 \ldots y_d)_2}$.

Wie die in Abbildung 2.6 angegebene Tabelle zeigt, lässt sich der Output z des in dieser Abbildung gezeigten MUX auch wie folgt beschreiben:

$$z = x_0\overline{y}_1\overline{y}_2 + x_1\overline{y}_1 y_2 + x_2 y_1\overline{y}_2 + x_3 y_1 y_2.$$

In dieser Summe tritt also pro Daten-Input x_i ein Summand auf. Dabei hat der Summand mit x_i, $0 \leq i \leq 3$, jeweils alle Steuersignale, und zwar negiert oder nicht negiert in Abhängigkeit von der Dualdarstellung von i. Ist z. B. $i = 2 = (10)_2$, so wird y_1 nicht negiert und y_2 negiert. Man beachte, dass diese Regel für jede Anzahl d von Steuersignalen gilt.

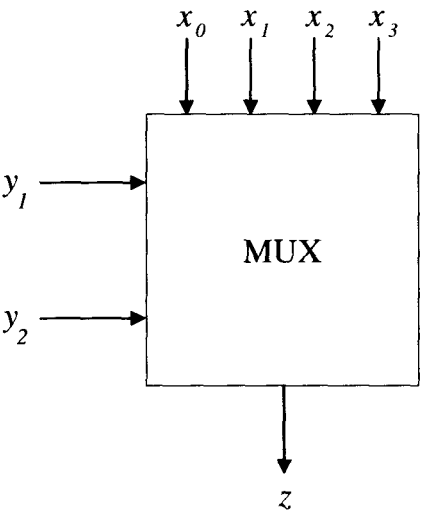

Abbildung 2.6: MUX für $d = 2$ (4 Daten-Inputs).

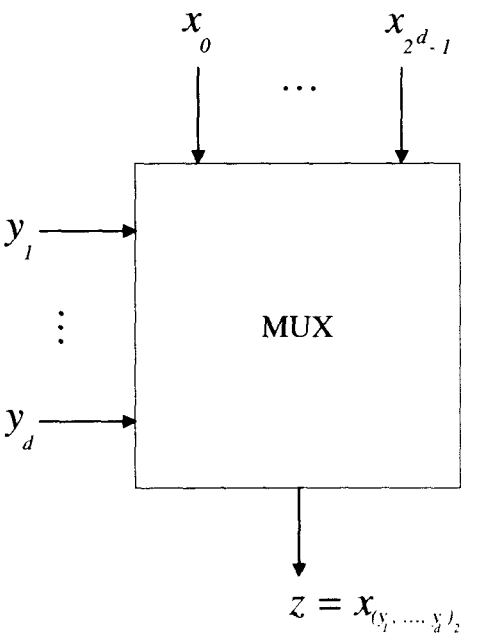

Abbildung 2.7: Allgemeiner Aufbau eines MUX.

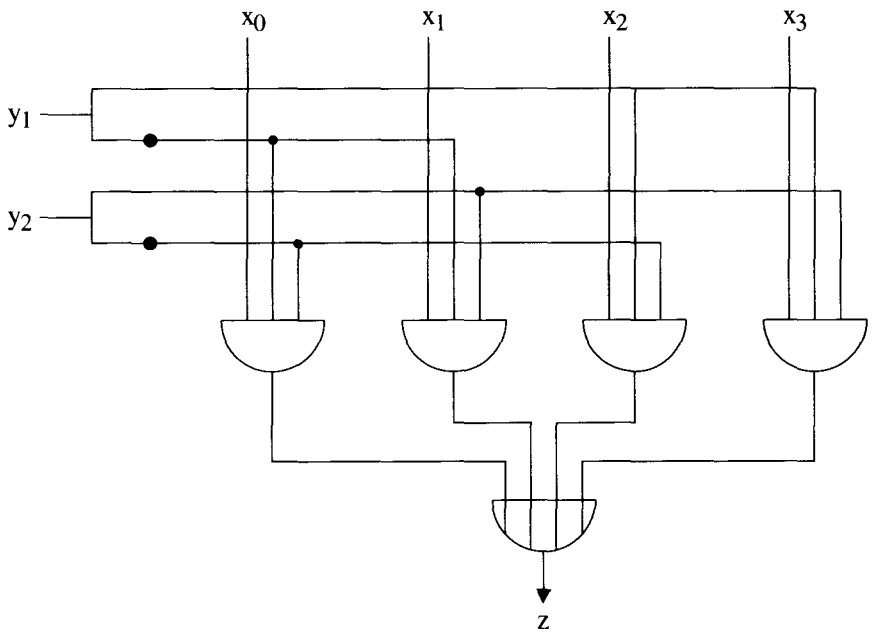

Abbildung 2.8: Realisierung des MUX aus Abb. 2.6 als dreistufiges Schaltnetz.

Aus der gerade angestellten Überlegung folgt unmittelbar, dass jeder MUX durch eine dreistufige Schaltung realisierbar ist: Auf der ersten Stufe werden die Negationen der Steuersignale berechnet. Die zweite Stufe besteht aus Und-Gattern für die einzelnen Summanden, wobei das i-te dieser Gatter das Produkt von x_i mit den entsprechenden negierten bzw. nicht negierten Kontrollsignalen bildet. Der Output des i-ten Gatters ist damit stets 0, es sei denn, die Steuersignale stellen die Binärcodierung von i dar; in diesem Fall ist der Output x_i. Die dritte Stufe besteht aus einem Oder-Gatter mit Inputs von jedem Und-Gatter; dieses Gatter berechnet die Summe, in welcher stets genau ein Summand ungleich 0 ist. Abbildung 2.8 zeigt ein solches Schaltnetz für den MUX aus Abbildung 2.6.

Dieser einfache Entwurf eines MUX ist allgemein nicht akzeptabel, da der an den einzelnen Gattern auftretende Fan-In sehr hoch werden kann: Das Oder-Gatter hat einen Fan-In von 2^d (im Beispiel = 4), die Und-Gatter haben einen Fan-In von $d+1$ (im Beispiel = 3). Wir werden als nächstes zeigen, wie sich durch eine Top-Down-Zerlegung des MUX-Entwurfs ein Schaltnetz angeben lässt, in welchem jedes Gatter einen Fan-In von höchstens 2 hat. Allerdings steigt die Anzahl der Stufen an.

Im Folgenden bezeichnen wir als *d-MUX* ein Schaltnetz für einen Multiplexer mit d Steuersignalen und 2^d Daten-Inputs. Der in Abbildung 2.6 gezeigte MUX ist damit ein 2-MUX. Die Idee der Zerlegung besteht darin, die Steuersignale in zwei Hälften zu spalten und einen $2d$-MUX aus $2^d + 1$ Kopien von d-MUXen zu konstruieren. Wir beginnen mit der Konstruktion eines 1-MUX, welcher in Abbildung 2.9 gezeigt ist. Man beachte, dass jedes der bei diesem 1-MUX verwendeten Gatter einen Fan-In $\leq$ 2 hat. Gemäß dem oben beschriebenen allgemeinen Vorgehen lässt sich sodann ein

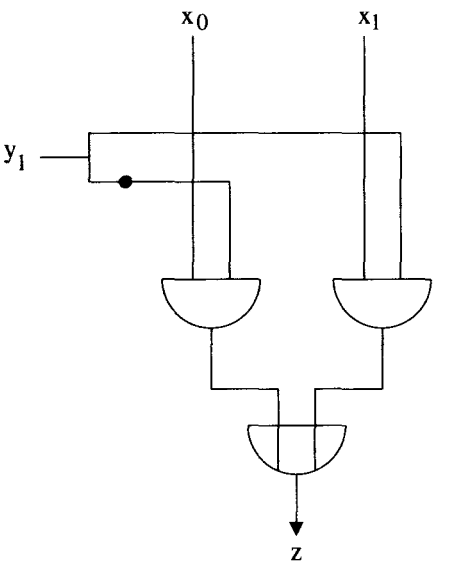

Abbildung 2.9: 1-MUX.

2-MUX aus 3 Kopien des 1-MUX konstruieren; dies ist in Abbildung 2.10 gezeigt. Die obere Zeile von Bausteinen des 2-MUX erhält die „obere Hälfte" der Steuersignale, und jeder 1-MUX dieser Zeile erhält 2^1 Daten-Inputs. Ist dann z. B. $y_2 = 1$, so werden in dieser Zeile x_1 und x_3 selektiert und an den 1-MUX der unteren Zeile weitergeleitet. Ist z. B. $y_1 = 0$, so wird dort x_1 selektiert.

Die allgemeine Konstruktion ist in Abbildung 2.11 gezeigt. Jeder d-MUX der oberen Zeile hat 2^d Daten-Inputs: Der erste hat die Inputs $x_0, \ldots, x_{2^d-1}$, der zweite die Inputs $x_{2^d}, \ldots, x_{2\times 2^d-1}$, der 2^d-te die Inputs $x_{(2^d-1)2^d}, \ldots, x_{2^{2d}-1}$. Man beachte, dass der Index des ersten x jeweils ein Vielfaches von 2^d ist. Die in der oberen Zeile anliegenden Steuersignale $y_{d+1}, \ldots, y_{2d}$ selektieren jeweils den k-ten Daten-Input mit

$$k = (y_{d+1} \ldots y_{2d})_2,$$

wobei der linkeste Input jeder Gruppe als 0 gezählt wird. Am d-MUX der unteren Zeile kommt daher der folgende Input an:

$$x_k, x_{2^d+k}, x_{2\times 2^d+k}, \ldots, x_{(2^d-1)2^d+k}.$$

Dieser MUX hat die Steuersignale $y_1, \ldots, y_d$ und selektiert daher den j-ten Input mit $j = (y_1 \ldots y_d)_2$ als Output, wobei wieder der linkeste Input als 0 gezählt wird. Der selektierte Output ist damit $x_{j\cdot 2^d+k}$.

Insgesamt wird also aus den 2^{2d} Daten-Inputs der i-te ausgewählt mit $i = j \cdot 2^d + k$. Dies gilt aufgrund folgender Überlegung: Wie oben erwähnt ist $j = (y_1 \ldots y_d)_2$. Eine Multiplikation von j mit 2^d entspricht einem Linksshift um d Stellen in der Dualdarstellung von j:

$$j \cdot 2^d = (y_1 \ldots y_d \underbrace{0 \ldots 0}_{d})_2.$$

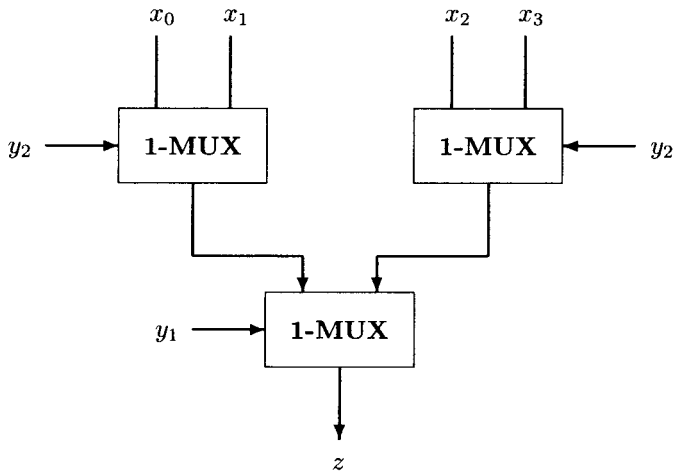

Abbildung 2.10: 2-MUX, konstruiert aus drei 1-MUXen.

Wegen $k = (y_{d+1} \dots y_{2d})_2$ folgt hieraus

$$j \cdot 2^d + k = (y_1 \dots y_d y_{d+1} \dots y_{2d})_2.$$

Der $2d$-MUX selektiert daher den Daten-Input x_i mit $i = (y_1 \dots y_{2d})_2$.

Wir wollen als nächstes die Gatter-Anzahlen des einfachen MUX und des aus obiger Konstruktion folgenden rekursiv aufgebauten d-MUX vergleichen. Beim d-MUX können die Inverter in der folgenden Überlegung ignoriert werden, da jedes der d Steuersignale einmal invertiert wird, so dass eine Addition von d am Ende der Rechnung dies korrigiert. Bezeichne dann $G(d)$ die Anzahl der Und- und Oder-Gatter eines d-MUX, so gilt:

$$\begin{aligned} G(1) &= 3 \\ G(2d) &= (2^d + 1) \cdot G(d) \end{aligned}$$

Durch vollständige Induktion (vgl. Aufgabe 2.1) kann man zeigen, dass diese Rekursionsgleichung folgende Lösung besitzt:

$$G(d) = 3 \cdot (2^d - 1).$$

Es gilt also z. B. $G(2) = 9$, $G(4) = 45$, $G(8) = 765$.

Zum Vergleich betrachten wir die Gatteranzahl bei einem einfachen MUX, welcher so modifiziert sei, dass alle Gatter ebenfalls einen Fan-In von höchstens 2 besitzen. Dazu wird jedes der 2^d Und-Gatter durch eine mehrstufige Schaltung (einen Baum) von d Und-Gattern und das finale Oder-Gatter mit 2^d Inputs durch $2^d - 1$ binäre Oder-Gatter ersetzt. Die Gesamtzahl der Gatter ergibt sich damit zu

$$2^d \cdot d + 2^d - 1 = 2^d \cdot (d + 1) - 1.$$

Für $d = 4$ hat ein einfacher MUX also 79 Gatter, für $d = 8$ bereits 2303.

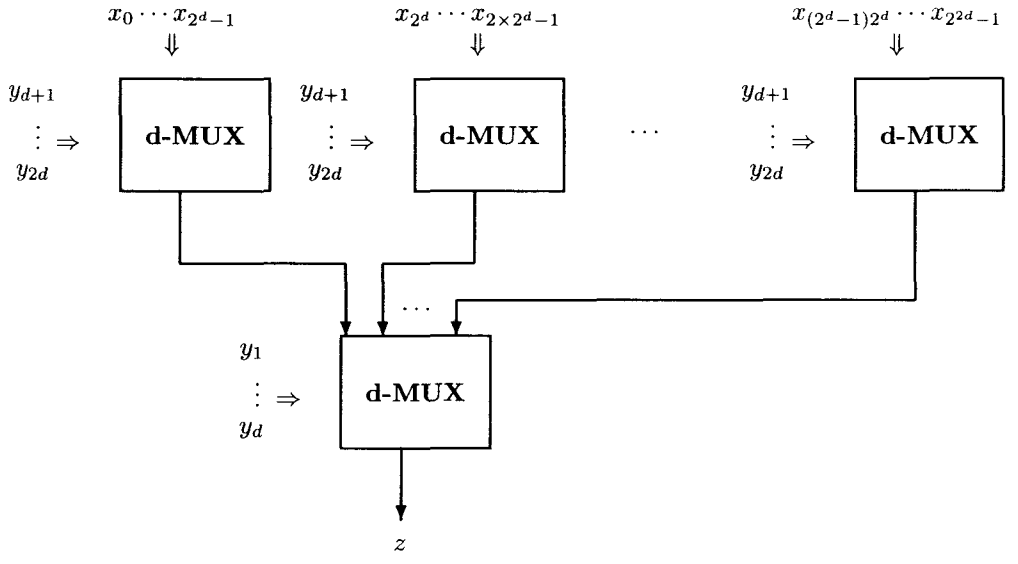

Abbildung 2.11: Top-Down-Multiplexer-Entwurf.

Wir erwähnen als nächstes eine wichtige Anwendung von Multiplexern: Sie können als Grundbausteine zur Realisierung beliebiger Boolescher Funktionen verwendet werden. Als Beispiel betrachten wir die durch folgende Funktionstafel gegebene Funktion f:

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Die Funktion f ist in alleiniger Abhängigkeit von x_1 und x_2 auch wie folgt darstellbar:

x_1	x_2	f
0	0	0
0	1	x_3
1	0	1
1	1	$\overline{x}_3$

Für jede Belegung von x_1 und x_2 entspricht der Wert von f also einem der Terme 0, 1, x_3 oder $\overline{x}_3$. Diese Beobachtung gilt für *jede* dreistellige Funktion. Dies legt folgende Realisierung nahe: Man verwende einen MUX mit x_1 und x_2 als Steuersignalen und

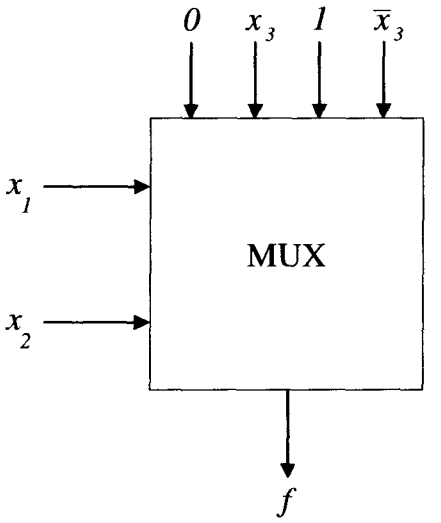

Abbildung 2.12: MUX zur Realisierung einer Booleschen Funktion.

den vier Daten-Inputs 0, 1, x_3, $\overline{x}_3$. Der Output dieses MUX, welcher in Abbildung 2.12 gezeigt ist, ist dann gerade der jeweilige Wert von f.

Allgemein ist dieses Vorgehen auf jede Stellenzahl anwendbar. Zur Realisierung aller Booleschen Funktionen der Form $f : B^3 \to B$ reicht *ein* MUX mit 4 Daten-Inputs, für Funktionen der Form $f : B^4 \to B$ ein solcher mit 8 Daten-Inputs (vgl. Aufgabe 2.7), allgemein für $f : B^n \to B$ ein MUX mit 2^{n-1} Daten-Inputs (und $n-1$ Steuersignalen). Dem individuellen Wertverlauf der Funktion f wird man durch die Schaltungs-Reihenfolge gerecht.

Wir zeigen noch eine einfachere Realisierung einer Booleschen Funktion durch einen MUX, welche die betreffende Funktion direkt in Hardware umsetzt und in der Literatur auch als *Hardware-Lookup* bezeichnet wird: Die gerade betrachtete Funktion f hat offensichtlich die Minterm-Darstellung

$$f(x_1, x_2, x_3) = m_3 + m_4 + m_5 + m_6.$$

Wir verwenden sodann einen 3-MUX zur Realisierung von f, bei welchem die drei Variablen als Steuersignale dienen und die acht Daten-Inputs mit 0 oder 1 fest beschickt werden in Abhängigkeit von ihrer Position: Wie aus Abbildung 2.13 zu ersehen ist, wird ein Daten-Input i mit 0 belegt, falls der Minterm m_i in der DNF von f fehlt, und mit 1 sonst. Zur Verifikation dieser Realisierung, die offensichtlich einfacher ist als die erste oben angegebene, dafür jedoch mehr Hardware erfordert, überlegen wir uns: Ist z. B. $x_1 = x_2 = x_3 = 0$, so wird der Input mit dem Index 0 zum Output durchgeschaltet: da dieser auf 0 gesetzt ist, folgt $f = 0$, also der richtige Wert. Analoge Überlegungen gelten für die anderen möglichen Werte der Steuer-Signale bzw. Variablen von f.

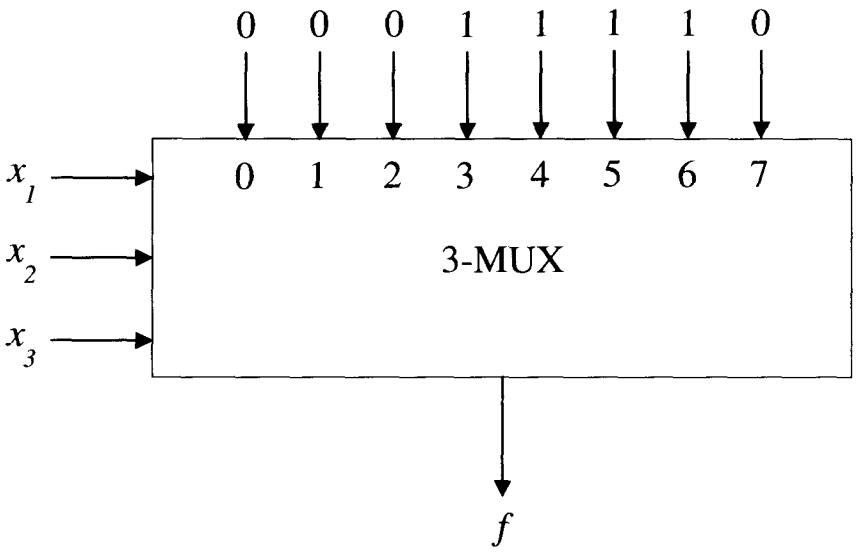

Abbildung 2.13: Alternative MUX-Realisierung einer Booleschen Funktion.

2.3 Demultiplexer, Decoder und Encoder

Wie eine genaue Betrachtung der eingangs angedeuteten formalen Beschreibung eines MUX als Boolesche Funktion zeigt, kann man jeden MUX unter Verwendung der aus Definition 1.4 bekannten Minterme wie folgt darstellen: Zu gegebenen Variablen $y_1, \ldots, y_d$ und einer Zahl i, $1 \leq i \leq 2^d - 1$ mit Dualdarstellung $(i_1 \ldots i_d)_2$ sei $m_i(y_1 \ldots y_d)$ definiert durch

$$m_i(y_1, \ldots, y_d) := y_1^{i_1} \cdot y_2^{i_2} \cdot \ldots \cdot y_d^{i_d}$$

mit

$$y_j^{i_j} := \begin{cases} y_j & \text{falls } i_j = 1 \\ \overline{y}_j & \text{falls } i_j = 0 \end{cases}$$

Ein *d-MUX* ist damit wie folgt beschreibbar:

- 2^d Daten-Inputs $x_0, \ldots, x_{2^d-1}$.
- d Steuersignale $y_1, \ldots, y_d$.
- 1 Output z mit $z = \sum_{i=0}^{2^d-1} x_i \cdot m_i(y_1, \ldots, y_d)$.

Aus dieser Darstellung lässt sich leicht die Darstellung eines *Demultiplexers*, im Folgenden abgekürzt als DeMUX, gewinnen. Informal hat ein DeMUX 1 Daten-Input sowie d Steuer-Signale wie ein MUX, die festlegen, auf welchen der 2^d Outputs der Input geschaltet wird. In der gerade für den MUX-Output z angegebenen Darstellung entfällt also die Summenbildung, und sämtliche x_i stimmen überein, d. h.

$$x_0 = x_1 = \ldots = x_{2^d-1}.$$

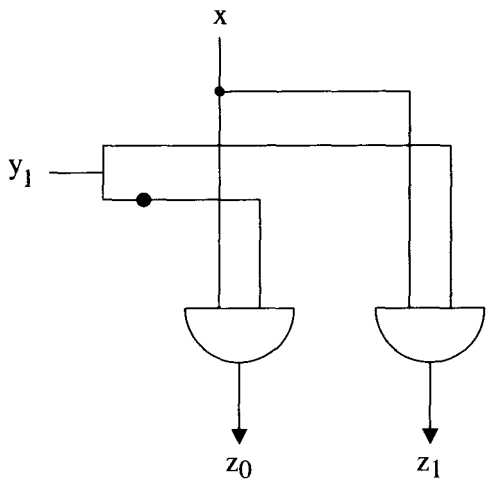

Abbildung 2.14: 1-DeMUX.

Bezeichnen wir den einen Input kurz mit x, so ist ein *d-DeMUX* wie folgt beschreibbar:

- 1 Daten-Input x,
- d Steuersignale $y_1, \ldots, y_d$,
- 2^d Outputs $z_0, \ldots, z_{2^d-1}$ mit $z_i = x \cdot m_i(y_1, \ldots, y_d)$ für $1 \leq i \leq 2^d - 1$.

Abbildung 2.14 zeigt einen 1-DeMUX (man vergleiche hierzu den 1-MUX in Abbildung 2.9); entsprechend zeigt Abbildung 2.15 einen 2-DeMUX (hierzu vergleiche man Abbildung 2.8). Der allgemeine Aufbau eines DeMUX der oben angegebenen Form ist in Abbildung 2.16 gezeigt; hierzu vergleiche man den MUX-Aufbau aus Abbildung 2.7. Wir bemerken zu diesem Baustein, dass ein DeMUX im Gegensatz zu einem MUX *nicht* universell ist, d. h. nicht zur Realisierung jeder Booleschen Funktion geeigneter Stellenzahl verwendet werden kann.

Wird bei einem Demultiplexer der Input x konstant mit „1" belegt, so folgt für den i-ten Output, $1 \leq i \leq 2^d - 1$:

$$z_i = 1 \cdot m_i(y_1, \ldots, y_d).$$

In diesem Fall wird also $z_i = 1$ genau dann, wenn $(y_1, \ldots, y_d)_2 = i$ ist. Der Demultiplexer fungiert jetzt als *Decoder*; x wird auch das *Aktivierungssignal* (*Enable Signal*) genannt, da x jetzt lediglich als Ein/Ausschalter wirkt. Aus den in den Abbildungen 2.14 bis 2.16 gezeigten DeMUXen erhält man also durch Vernachlässigung der jeweiligen x-Signale bereits Decoder; auf Grund ihrer engen Verwandschaft werden die Bezeichnungen „DeMUX" und „Decoder" häufig auch synonym verwendet.

Ein Decoder kann damit aufgefasst werden als ein Baustein mit d Inputs und 2^d Outputs, welcher genau einen der Outputs auf „1" setzt in Abhängigkeit von dem durch die Inputs binär codierten Index. Anders ausgedrückt wird durch die Inputs

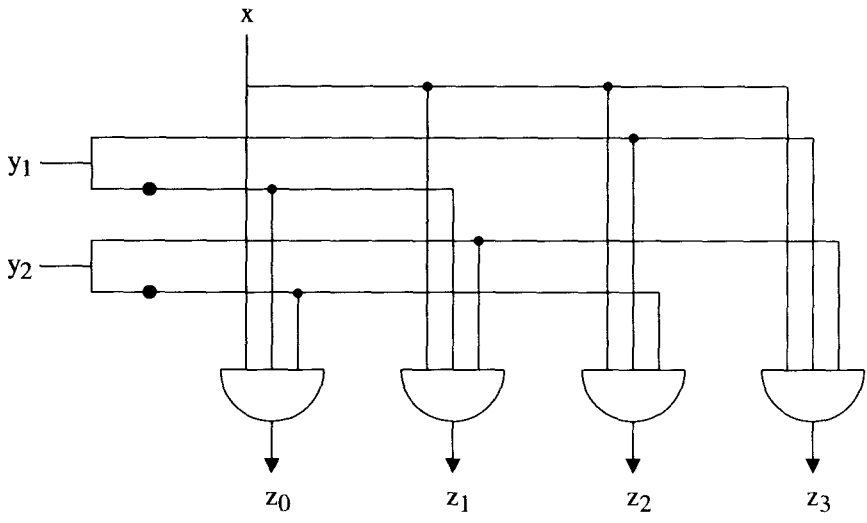

Abbildung 2.15: 2-DeMUX.

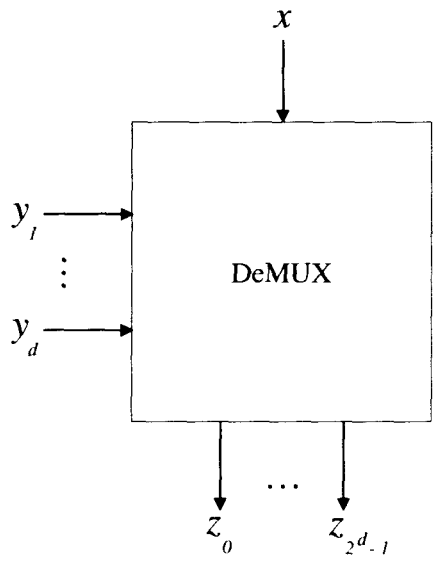

Abbildung 2.16: Allgemeiner Aufbau eines DeMUX.

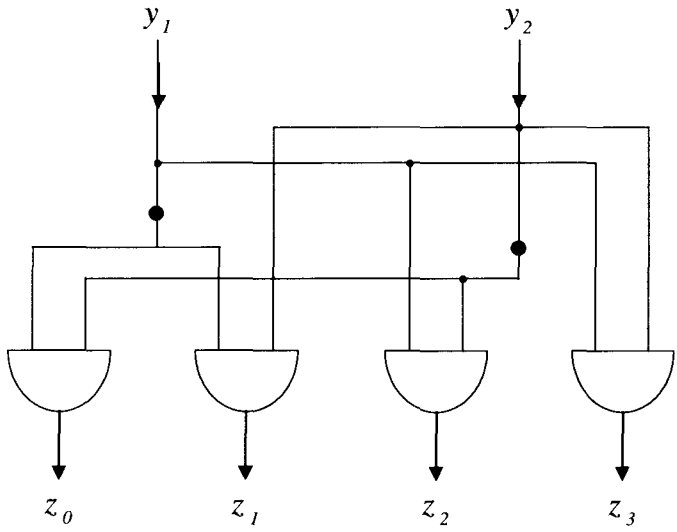

Abbildung 2.17: 2 × 4-Decoder.

ein bestimmter Output *adressiert*; wir werden hierauf in Kapitel 7 im Zusammenhang mit der Anwendung von PLAs zurückkommen.

Der Vollständigkeit halber zeigt Abbildung 2.17 eine Darstellung des 2-DeMUX aus Abbildung 2.15 ohne das (jetzt als konstant = 1 angenommene) x-Signal, also einen Decoder. Da dieser 2 Eingänge auf $2^2 = 4$ Ausgänge schaltet, wird er auch als 2 × 4-Decoder bezeichnet.

Als letzten Standardbaustein erwähnen wir den *Encoder*, der die umgekehrte Funktion eines Decoders hat. Ein Encoder hat 2^d Inputs und d Outputs; anstatt eine Adresse zur Aktivierung eines bestimmten Outputs zu verwenden, *erzeugt* ein Encoder die Adresse des aktuell aktiven Input-Signals. Ein Beispiel für einen 4 × 2-Encoder zeigt Abbildung 2.18; die Funktionstafel des in dieser Abbildung gezeigten Encoders lautet:

x_0	x_1	x_2	x_3	y_0	y_1
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

Man beachte, dass bei einem Encoder stets angenommen wird, dass genau ein Input aktiviert ist; die Schaltfunktion ist also partiell.

Wir demonstrieren abschließend noch einmal die Universalität von Multiplexern und Decodern durch die Angabe drei verschiedener Schaltungen für die Funktion

$$f(x_1, x_2, x_3, x_4) = \overline{x}_1\overline{x}_2\overline{x}_3\overline{x}_4 + \overline{x}_1 x_2 \overline{x}_3 x_4 + x_1 x_2 x_3 x_4 + x_1 \overline{x}_2 x_3 \overline{x}_4 :$$

1. Verwendung eines 4-MUX: In Analogie zu Abbildung 2.13 werden die x_i als

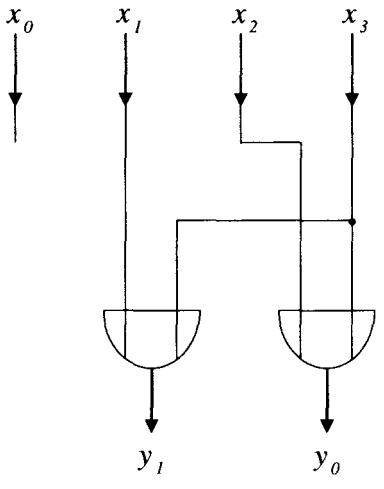

Abbildung 2.18: 4 × 2-Encoder.

Steuersignale verwendet; die Inputs, welche den Mintermen m_0, m_5, m_{10} und m_{15} entsprechen, werden auf 1 gesetzt, alle anderen auf 0.

2. Verwendung eines Decoders mit 4 Inputs $x_1, \ldots, x_4$ und 16 Outputs gemäß Abbildung 2.19: Die Outputs, welche den 4 Mintermen zu einschlägigen Indizes entsprechen, werden durch ein Oder-Gatter verknüpft.

3. Kombination von Decoder und MUX: Es gibt 4 Input-Kombinationen, für welche $f = 1$ gilt:

   (a) $x_1x_2 = 00$ und $x_3x_4 = 00$,

   (b) $x_1x_2 = 01$ und $x_3x_4 = 01$,

   (c) $x_1x_2 = 11$ und $x_3x_4 = 11$,

   (d) $x_1x_2 = 10$ und $x_3x_4 = 10$.

   Wir verteilen die Inputs auf einen 2 × 4-Decoder und einen 2-MUX gemäß Abbildung 2.20: Gilt z. B. $x_1x_2 = 00$, so setzt der Decoder z_0 auf 1: gilt außerdem $x_3x_4 = 00$, so wird dieses Signal an den Ausgang des MUX weitergegeben.

2.4 Addiernetze mit Halb- und Volladdierern

Wir wenden uns als nächstes der Entwicklung von Schaltnetzen für die *Addition* zu, welche sich als zentrale Bausteine in Rechnern erweisen werden. Auch bei der Addition kann man durch andere Verfahren als die aus den Darstellungssätzen resultierenden zu geeigneten Schaltnetzen kommen.

Dazu betrachten wir exemplarisch noch einmal die Addition von zwei 16-stelligen Dualzahlen (vgl. Beispiel 1.5): Eine Möglichkeit, ein Schaltnetz zur Lösung dieser

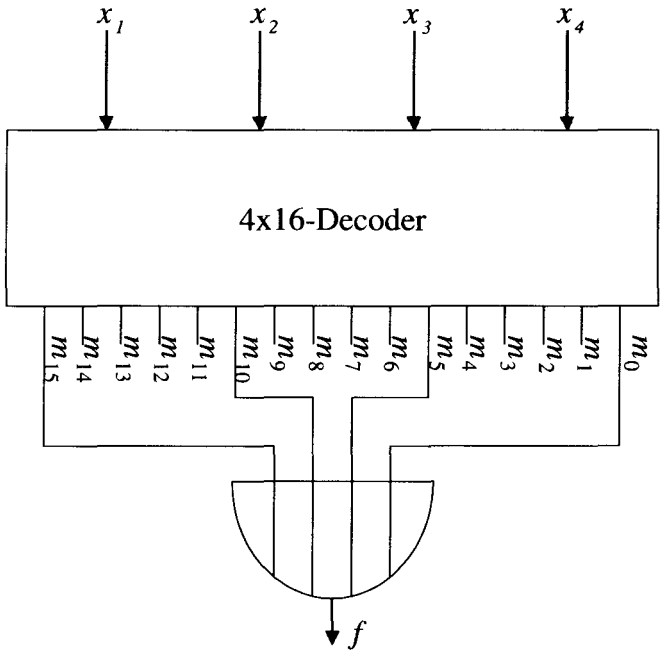

Abbildung 2.19: Realisierung einer Booleschen Funktion mittels Decoder.

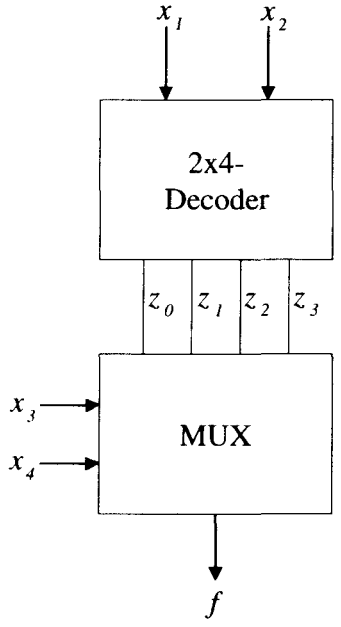

Abbildung 2.20: Realisierung einer Booleschen Funktion mittels Decoder und MUX.

Tabelle 2.2: Funktionstafel eines Halbaddierers.

x	y	U	R
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Aufgabe zu entwerfen, besteht sicherlich darin, für die Schaltfunktion

$$\mathcal{A} : B^{32} \to B^{17}$$

bzw. die entsprechenden 17 Booleschen Funktionen Tabellen aufzustellen und Satz 1.6 anzuwenden. Jede dieser 17 Funktionen hat $2^{32} \approx 4 \cdot 10^9$ Argumente-Tupel, und unter der Annahme, dass jede Funktion für etwa 50% der Argumente-Tupel den Wert 1 annimmt, ist dann mit ungefähr $2 \cdot 10^9 \cdot 17 = 3,4 \cdot 10^{10}$ einschlägigen Mintermen zu rechnen. Da man pro Minterm 15 Und-Gatter mit je 2 Eingängen benötigt, wären für eine entsprechende Schaltung allein rund $5,1 \cdot 10^{11}$ Und-Gatter erforderlich. Dies ist offensichtlich nicht realisierbar, und auch Und-Gatter mit 16 Eingängen würden keine spürbare Vereinfachung bringen. Völlig analoge Überlegungen zeigen, dass schon die Addition von zwei 8-stelligen Dualzahlen zu ähnlichen Problemen führt: Bei 65.536 Argumente-Tupeln wäre nun mit ca. $2,9 \cdot 10^5$ Mintermen zu rechnen. Wir müssen uns daher ein anderes Vorgehen überlegen und demonstrieren dies für die Addition von zwei 4-stelligen Dualzahlen: Ausgangspunkt ist die Betrachtung der Addition im Dualsystem. Bekanntlich kann man im Dualsystem genauso wie im Dezimalsystem kalkülmäßig addieren, indem man von hinten beginnend die einzelnen Stellen addiert und dabei eventuell auftretende Überträge berücksichtigt:

Beispiel 2.1 Addition von 183 und 197 im Dezimal- bzw. Dualsystem:

Dezimal-Addition:	Dual-Addition:
183	10110111
+197	+11000101
11	*1* *111*
380	101111100

□

Offensichtlich gilt in diesem Beispiel folgendes: Die letzte Stelle (ganz rechts) hat als einzige Stelle keinen Übertrag zu berücksichtigen, an allen anderen Stellen kann ein Übertrag auftreten: wir analysieren dies genauer:

(a) Wir betrachten zunächst die *letzte* Stelle: Eingabe sind hier zwei Dualziffern x und y, Ausgabe eine Resultatstelle R und ein Übertrag U für die nächste Stelle, und wir erhalten die in Tabelle 2.2 gezeigte Funktionstafel. Offensichtlich gilt:

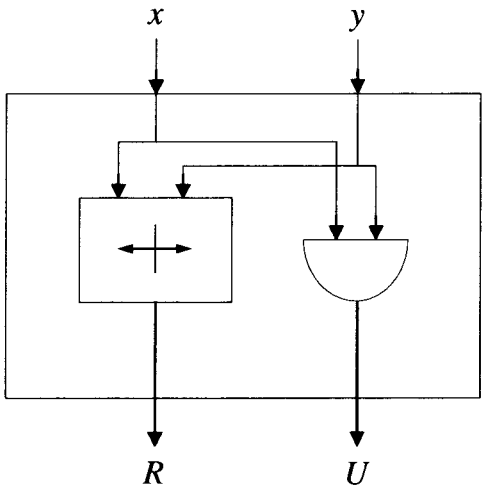

Abbildung 2.21: Halbaddierer.

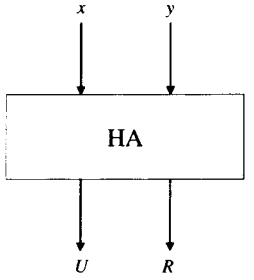

Abbildung 2.22: Kurzbezeichnung für den Baustein „Halbaddierer".

$R = \overline{x} \cdot y + x \cdot \overline{y} = x \leftrightarrow y$ und $U = x \cdot y$, so dass wir das in Abbildung 2.21 gezeigte Schaltnetz erhalten (unter Verwendung des oben bereits beschriebenen Bausteins für $\leftrightarrow$). Diesen Modul betrachten wir von nun an als neuen Baustein und nennen ihn *Halbaddierer* mit der in Abbildung 2.22 gezeigten Kurzbezeichnung. Es sei darauf hingewiesen, dass Halbaddierer in der Praxis heute keine Rolle mehr spielen, zumindest nicht als separate Bauteile. Wir verwenden ihn lediglich konzeptionell. Des Weiteren sei bemerkt, dass wir bei dem in Abbildung 2.21 gezeigten Schaltbild des Halbaddierers bereits von einer zeichnerischen Vereinfachung Gebrauch gemacht haben, welche immer dann Verwendung findet, wenn die Inputs einen hohen Fan-Out haben: Man zeichnet die Inputs dann als *Schienen*.

(b) Betrachten wir nun eine *beliebige andere* Stelle unserer Additionsaufgabe: Jede Stelle ungleich der letzten muss unter Umständen einen Übertrag berücksichtigen. Eingabe sind somit zwei Dualziffern x, y *und* ein dualer Übertrag u, Ausgabe wie oben ein Resultat R und ein Übertrag U für die nächste Stelle, so dass wir jetzt die in

Tabelle 2.3: Funktionstafel eines Volladdierers.

x	y	u	U	R
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

Tabelle 2.3 gezeigte Funktionstafel erhalten. Als Darstellungen für R und U erhalten wir nach kurzer Überlegung:

$$\begin{aligned} R &= x \leftrightarrow y \leftrightarrow u \\ U &= x \cdot y + x \cdot u + y \cdot u \\ &= x \cdot y + (x \leftrightarrow y) \cdot u \end{aligned}$$

Damit erhalten wir das in Abbildung 2.23 gezeigte Schaltnetz, bei welchem wir den Halbaddierer bereits als Baustein einsetzen. Dieses Schaltnetz nennen wir *Volladdierer* und bezeichnen es als Baustein wie in Abbildung 2.24 gezeigt.

Damit haben wir nun genügend Bausteine zusammen, um ein Schaltnetz für die Addition von zwei 4-stelligen Dualzahlen zu entwerfen (vgl. Abbildung 2.25), welches „lediglich" den bekannten Ziffernkalkül nachvollzieht; die zu addierenden Zahlen x und y seien dabei in Dual-Ziffernschreibweise $x_3x_2x_1x_0$ bzw. $y_3y_2y_1y_0$ so gegeben, dass die Indizes jeweils den Zweierpotenzen entsprechen.

Dieses Schaltnetz nennt man auch *asynchrones (Parallel-) Addiernetz*. Offensichtlich lässt es sich durch Hinzunahme weiterer Volladdierer beliebig auf Inputs mit mehr als vier Stellen erweitern. Für 16-stellige Dualzahlen z. B. reichen 15 Volladdierer und 1 Halbaddierer aus (anstelle der Betrachtung von 3,4 Milliarden Mintermen). Außerdem sei bemerkt, dass dieses Addiernetz auch allein aus Volladdierern aufgebaut werden kann; in diesem Fall muss dafür gesorgt werden, dass am u-Eingang des ersten (rechtesten) Volladdierers stets eine 0 anliegt. Da bei beiden Realisierungsvarianten der endgültige Übertrag durch das gesamte Schaltnetz „rieselt", bezeichnet man diesen Addierer auch als *Ripple-Carry-Adder*.

2.5 Beschleunigung der Übertragsberechnung

Am Beispiel der dualen Addition wollen wir nun auf das Problem der *Beschleunigung* von Schaltnetzen eingehen. Dazu überlegen wir uns, nach welcher Zeit wir beim 4-stelligen Ripple-Carry-Addiernetz mit dem (vollständigen) Ergebnis rechnen können: Unterstellen wir wie in Kapitel 1 eine Schaltzeit von 10 psec $=10^{-11}$ sec pro Gatter, so liefert ein Halbaddierer beide Outputs nach 30 psec, ein Volladdierer nach 70 psec.

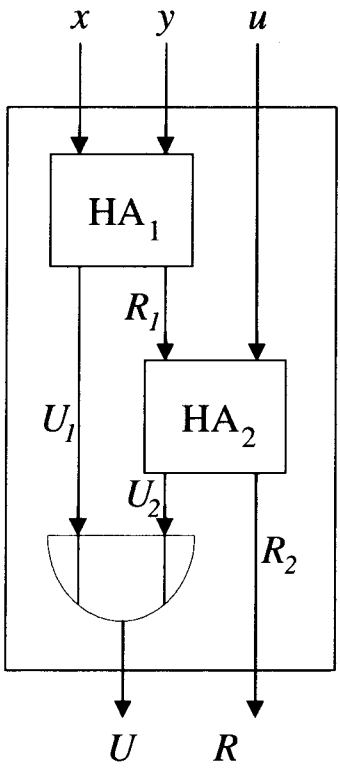

Abbildung 2.23: Volladdierer.

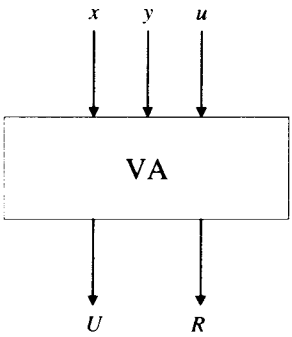

Abbildung 2.24: Kurzbezeichnung für den Baustein „Volladdierer".

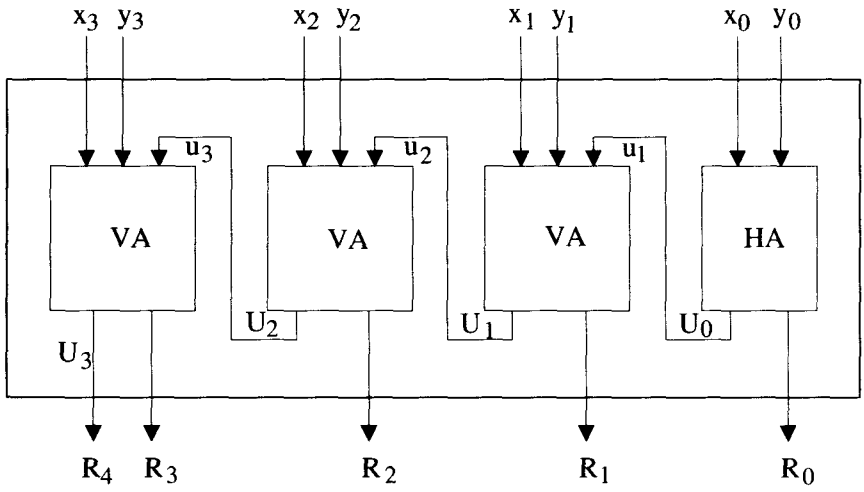

Abbildung 2.25: Addiernetz für zwei 4-stellige Dualzahlen.

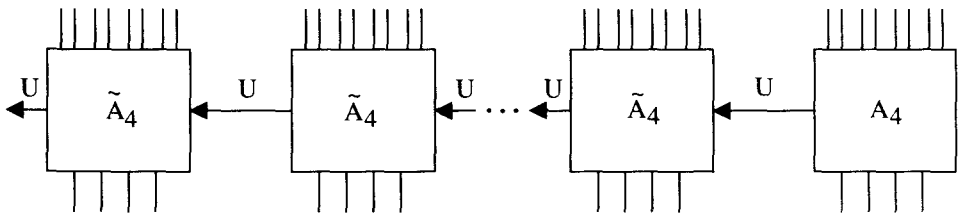

Abbildung 2.26: Prinzipschaltbild eines n-stelligen Addiernetzes.

Da nun bei obigem Addiernetz ein Übertrag (wie z. B. bei der Addition von 1111 und 0001) unter Umständen das ganze Netz durchlaufen kann, ist (im schlimmsten Fall) erst nach (70+70+70+30) psec = 240 psec mit einem „vollständigen" Ergebnis $R_4 \ldots R_0$ zu rechnen. Wir wollen nun daran gehen, diese Zeit zu verkürzen durch die Verwendung zusätzlicher Hardware, wobei wir alle Überlegungen exemplarisch für derartige Addiernetze durchführen.

Das eventuelle Durchlaufen des Übertrags durch die ganze Schaltung stört offensichtlich besonders bei großen Wortlängen wie z. B. $n = 32$. Eine Idee zur Beschleunigung eines solchen Addiernetzes ist daher die Verringerung der Anzahl der Schaltebenen durch Zusammenfassung von Bit-Gruppen der Größe $g = 4$, wie in Abbildung 2.26 skizziert. Dabei besteht A_4 aus drei Volladdierern und einem Halbaddierer. $\tilde{A}_4$ jeweils aus vier Volladdierern. Der zeitliche Engpaß liegt offenbar an den Übergangsstellen zwischen den einzelnen Modulen, d. h. bei U. Würde also U schneller zur Verfügung stehen, könnte im jeweils nächsten Modul bereits weiter gerechnet werden. Dies führt auf die Einführung einer zusätzlichen Schaltung zur schnellen Bestimmung von U (engl. *Carry-Look-Ahead* oder *Carry-Bypass*), welche wir gemäß folgender Formel für U (entspricht R_4 in dem in Abbildung 2.25 angegebenen 4-stelligen Addiernetz) entwickeln:

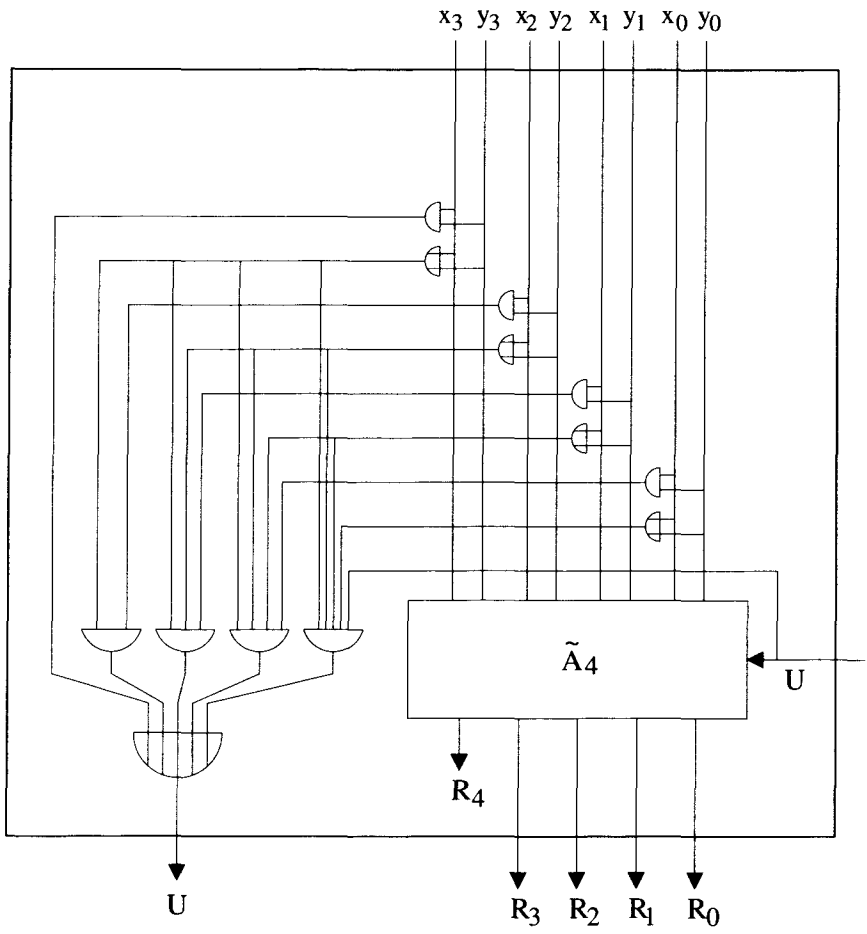

Abbildung 2.27: Carry-Bypass-Addiernetz.

$$
\begin{aligned}
U \;=\; & x_3y_3 \\
& + (x_3 + y_3)x_2y_2 \\
& + (x_3 + y_3)(x_2 + y_2)x_1y_1 \\
& + (x_3 + y_3)(x_2 + y_2)(x_1 + y_1)x_0y_0 \\
& + (x_3 + y_3)(x_2 + y_2)(x_1 + y_1)(x_0 + y_0)u
\end{aligned}
$$

Dabei entfällt die letzte Zeile für einen Baustein vom Typ A_4. Die Bausteine vom Typ $\tilde{A}_4$ erweitern wir somit wie in Abbildung 2.27 gezeigt (unter der Voraussetzung, dass uns auch für mehr als zwei Inputs schnelle Und- bzw. Oder-Gatter zur Verfügung stehen). Die dort gezeigte Schaltung ist dreistufig: Auf der ersten Stufe werden jeweils zwei Inputs multiplikativ und additiv verknüpft, auf der zweiten werden dann die benötigten Produkte gebildet und auf der dritten schließlich die Summe. Also liegt

U nach 30 psec vor, wenn man wieder pro Stufe von einer Schaltzeit von 10 psec ausgeht. Gegenüber der oben angegebenen Schaltung ist diese also um den Faktor 8 schneller, was den Übertrag betrifft. Es sei bemerkt, dass sich die Situation auch für Bit-Gruppen-Zusammenfassungen der Größe $g = 8$ nicht wesentlich ändert: Auch dann kommt man mit einer dreistufigen Zusatzschaltung aus, wenn man U schnell ermitteln will, nur wird diese „breiter" sein wegen der höheren Zahl von Inputs.

Die Konstruktion der Carry-Bypass-Schaltung in der gerade beschriebenen Form hat den offensichtlichen Nachteil, dass sie ein Oder- sowie ein Und-Gatter mit einem Fan-In von jeweils 5 (bzw. allgemein $n+1$ bei Stellenzahl n der zu addierenden Operanden) aufweist. Für höhere Stellenzahlen ist dies schwierig zu realisieren; allerdings kann man durch eine geeignete Modularisierung hier Abhilfe schaffen.

Für den Fall, dass mehr als 2 Summanden addiert werden sollen, kann man ein *Carry-Save-Addiernetz* verwenden (vgl. auch Abschnitt 6.3). Ein solches Netz ist ein im Allgemeinen mehrstufiges Addiernetz, welches in der ersten Stufe drei Summanden addiert, in jeder weiteren Stufe jeweils einen weiteren Summanden hinzuaddiert. Die pro Addition auftretenden Ergebnisbits bilden einen Summanden der nächsten Stufe, die auftretenden Carry-Bits bilden einen weiteren solchen Summanden (die Carry-Bits werden also nicht unmittelbar aufsummiert, sondern für die nächste Stufe erhalten). Wir illustrieren dies an zwei Beispielen von 4 Summanden X, Y, Z und W mit der Stellenzahl $n = 4$, für welche ein Ablauf z. B. wie folgt lauten kann:

X	0101		1111
Y	0011		1101
Z	+ 0100	bzw.	+ 0111
Summe	0010		0101
Übertrag	1010		11110

Man beachte, dass der Übertrag eventuell eine zusätzliche Stelle erfordert.

Im nächsten Schritt werden die berechnete Summe und der (separate) Übertrag zum nächsten Summanden addiert:

Summe	0010		0101
Übertrag	1010		11110
W	+ 0001	bzw.	+ 1111
neue Summe	1001		10100
neuer Übertrag	0100		11110

In jeder Stufe werden drei Summanden durch einen „Carry-Save-Addierbaustein" (CSA) auf zwei Summanden reduziert, von denen der eine sich aus den Summenausgängen und der andere sich aus den Übertragsausgängen der Volladdierer ergibt. In der darauf folgenden Stufe kann ein weiterer Summand hinzugenommen werden. Schließlich verbleiben zwei Summanden aus den letzten Summen- bzw. Übertragsausgängen. Diese können mit irgendeinem Addiernetz zur Addition von *zwei* Summanden, z. B. einem der oben vorgestellten, zur Endsumme verknüpft werden. Es ergibt sich damit insgesamt ein Addiernetz wie in Abbildung 2.28 gezeigt. Das Prinzip der Carry-Save-Addition ist für den oben geschilderten Fall von 4 Summanden in Abbildung 2.29 dargestellt.

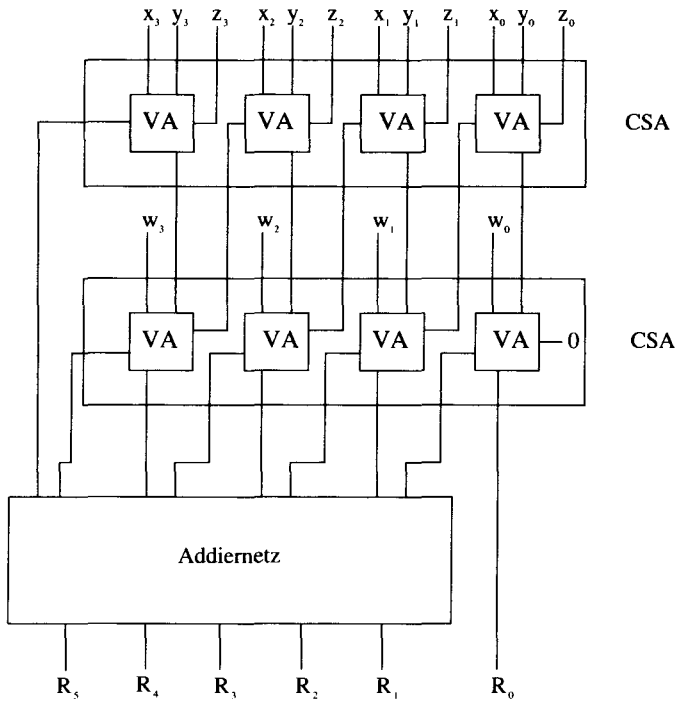

Abbildung 2.28: 4-stelliges Carry-Save-Addiernetz für 4 Summanden.

Wir können nun mit dieser Idee möglichst viele CSAs in *einer* Stufe parallel schalten. In Abbildung 2.30 ist eine solche Möglichkeit für die Addition von 8 Summanden gezeigt. Man schaltet von den jeweils noch verbliebenen Summanden (unter Auslassung von gegebenenfalls ein oder zwei Summanden) je drei zusammen und iteriert dieses Verfahren. Es entsteht eine baumartige Anordnung von CSAs; man spricht auch von einem *Adder-Tree* bzw. nach seinem Autor von einem *Wallace-Tree*. Es kann gezeigt werden, dass die Anzahl der CSA-Stufen eines Wallace-Trees oberhalb des finalen Addiernetzes für m Summanden durch $\log m$ — genommen zur Basis 3/2 — beschränkt ist. Da der Volladdierer eine feste Tiefe hat, ist die Gesamt-Tiefe des Wallace-Trees von logarithmischer Größenordnung. Obwohl sich von oben nach unten die erforderliche Stellenzahl der einzelnen CSA-Stufen erhöhen kann, nimmt deren Gesamtzahl pro Stufe ab. Es lässt sich zeigen, dass sich bei einer anfänglichen Stellenzahl $n \geq 3$ der Wallace-Tree nach unten hin schnell verjüngt.

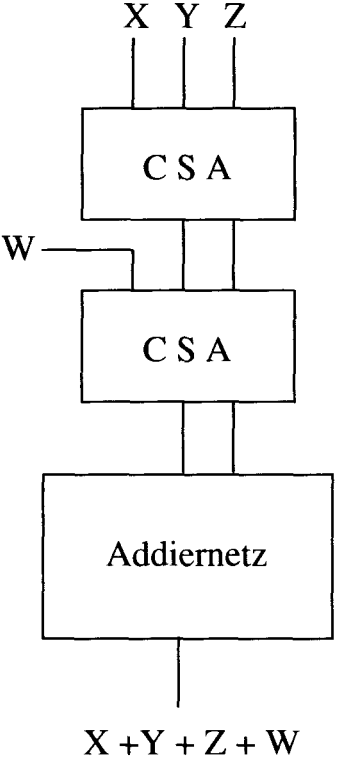

Abbildung 2.29: Prinzip der Carry-Save-Addition.

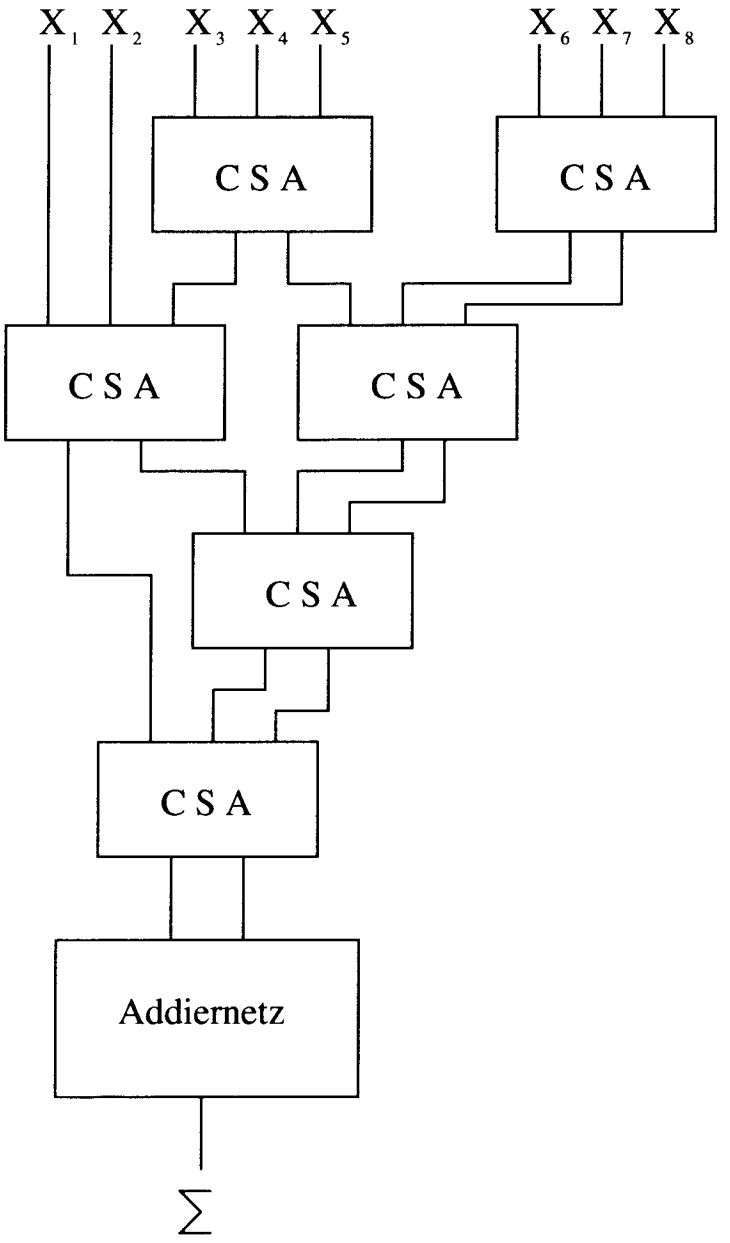

Abbildung 2.30: Carry-Save-Addierer für 8 Summanden (Wallace-Tree).