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 7

Programmierbare Logik und VLSI

7.1 Einführung

Wir knüpfen zunächst an die in den Kapiteln 1 und 2 angestellten Überlegungen an: Dort haben wir einerseits (in Form der Darstellungssätze 1.6, 1.10 und 1.15) Methoden kennengelernt, für beliebige Schaltfunktionen Schaltnetze zu entwerfen. Dabei haben wir uns insbesondere für *zweistufige* (SOP- oder POS-) Realisierungen interessiert (wie sie z. B. durch DNF und KNF ermöglicht werden), da diese kurze Signallaufzeiten aufweisen. Außerdem haben wir in Abschnitt 3.1 für disjunktive Darstellungen eine Vereinfachungsstrategie diskutiert, deren Ziel es war, eine gegebene Boolesche Funktion möglichst kostengünstig – und das hieß dort: mit möglichst wenig Und- bzw. Oder-Gattern – zu realisieren. Einen Nachteil dieser Strategie haben wir bisher verschwiegen: Die Bauteile, welche man zur Realisierung einer z. B. Karnaugh-optimierten Funktion benötigt, haben unterschiedliche Formate. Beispielsweise hat ein Inverter einen Ein- und einen Ausgang, ein Und- oder ein Oder-Gatter aber schon zwei Eingänge und einen Ausgang; häufig haben wir auch Gatter mit mehr als zwei Eingängen verwendet (vgl. z. B. die Carry-Bypass-Schaltung in Abschnitt 2.5). Gerade dies ist jedoch problematisch, wenn man eine solche Schaltung in moderner Halbleitertechnik realisieren will, denn die heute möglichen Packungsdichten (von mehr als 10^5 Bauelementen auf einem quadratischen Chip der Kantenlänge 4 mm) erfordern einen weitgehend automatisierten Herstellungsprozess, welcher durch den Wunsch, viele *unterschiedlich* formatierte Gatter auf einem Chip unterzubringen, deutlich erschwert wird. Darüberhinaus ist auch schon das Design solcher *VLSI-Chips* eine schwierige graphentheoretische Aufgabe, die eine Reihe von Nebenbedingungen zu berücksichtigen hat. Grundsätzlich hat man diese Probleme zwar heute im Griff: wie wir zu Beginn von Abschnitt 2.1 bereits erwähnten, ist man heute in der Lage, irgendeinen „special purpose"-Chip kurzfristig herzustellen, aber die Situation bleibt unbefriedigend.

Man kennt andererseits insbesondere aus der Schaltkreistheorie auch andere Optimierungsprinzipien, wie z. B. das folgende: Man entwerfe Schaltungen, welche primär mit *möglichst wenig Gattertypen* auskommen (und sekundär auch noch möglichst

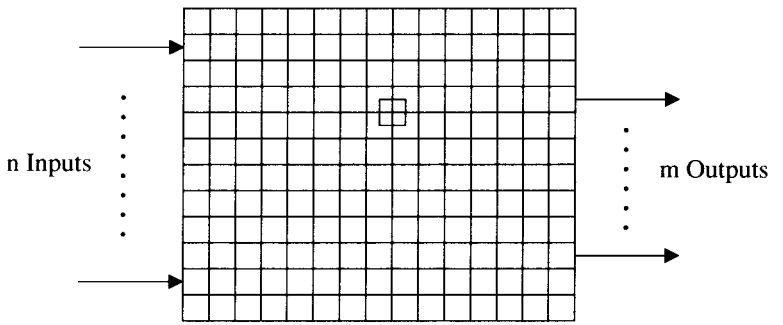


Abbildung 7.1: Prinzipaufbau eines PLAs.

wenig Gatter verwenden). Man darf dann natürlich nicht mehr erwarten, mit zweistufigen Schaltungen alle Probleme lösen zu können. Wie aus Satz 1.17 folgt, sind im Prinzip alle Booleschen und damit auch alle Schaltfunktionen allein mit Nand- oder mit Nor-Gattern realisierbar, aber der Beweis dieses Satzes zeigt, dass z. B. ein Oder-Gatter nur durch drei Nand-Gatter ersetzbar ist, was bereits andeutet, dass es für solche universelle Modulen keine „kurzen", übersichtlichen Normalformen gibt. (Die absolute Minimierung der Gatter-Zahl ohne Rücksicht auf die Stufenzahl ist Gegenstand der Theorie der Schaltkreiskomplexität, vgl. die bibliographischen Hinweise zu Kapitel 1.)

In diesem Kapitel werden wir eine neue Technik studieren, welche auf folgender Idee beruht: Man entwerfe für verschiedene Schaltfunktionen einen universell verwendbaren *Einheitsbaustein* mit möglichst homogener Netzstruktur, der für *unterschiedliche* Anwendungen eingesetzt werden kann (vergleichbar einem Stück Holz, aus welchem verschiedenartigste Figuren durch Schnitzen gefertigt werden können). Naturgemäß wird ein solcher Baustein etwas aufwendiger sein als eine Schaltung, welche nur im Hinblick auf *eine* Anwendung entworfen wird, dafür darf man andererseits einen übersichtlichen Aufbau sowie eine hohe Wartungsfreundlichkeit erwarten, was insbesondere für die Herstellung, das Testen und den Betrieb eines solchen Bausteins von großer Bedeutung ist. Bereits in Abschnitt 2.2 haben wir mit dem Multiplexer MUX einen Baustein kennen gelernt, welcher universell alle n-stelligen Booleschen Funktionen realisieren kann. Dabei kam es im Wesentlichen auf die geometrisch richtige Verschaltungsreihenfolge von vier verschiedenen Daten (0, 1, x_n und $\overline{x}_n$) auf die 2^{n-1} Inputs des MUX an.

7.2 Aufbau eines PLAs

Der heute tatsächlich verwendete Typ eines Einheitsbausteins ist das *Programmierbare Logische Feld* (engl. Programmable Logic Array, kurz *PLA*), welches prinzipiell den in Abbildung 7.1 gezeigten Aufbau hat. Intern ist ein PLA gitterförmig verdrahtet, wobei jeder Kreuzungspunkt von zwei Drähten der Mittelpunkt eines einheitlich formatierten Bausteins ist. Ein solches „Kästchen" ist in Abbildung 7.1 eingezeichnet; in einer Ausschnittsvergrößerung sieht dieses wie in Abbildung 7.2 gezeigt aus.

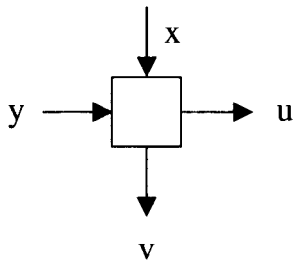


Abbildung 7.2: „Gitterpunkt" eines PLAs.

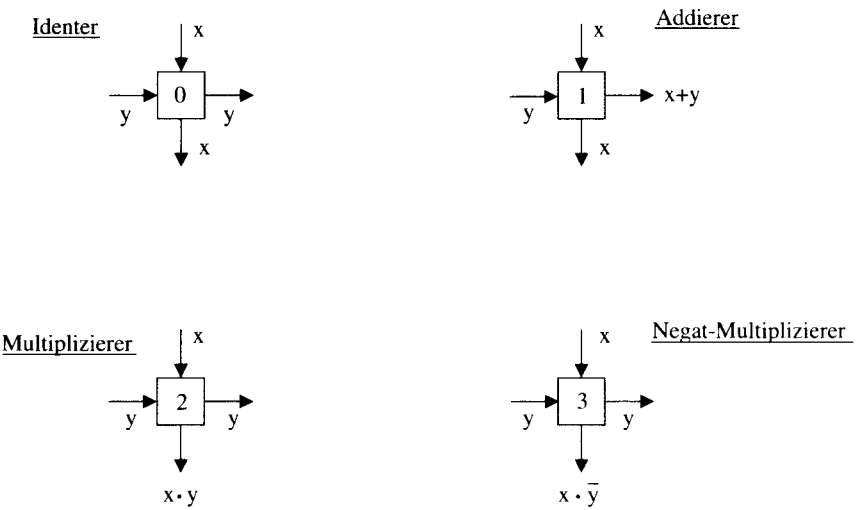


Abbildung 7.3: Bausteintypen eines PLAs.

Wir wollen die Wirkungsweise eines PLAs an vier verschiedenen Typen solcher „Gitterpunkte" demonstrieren: es wird sich zeigen, dass diese ausreichen. Wir bezeichnen die Typen mit 0, 1, 2, 3 mit der in Abbildung 7.3 gezeigten Bedeutung. Mindestens einer der beiden Inputs wird an einen Ausgang unverändert weitergegeben, beim Identer sogar beide. Der Addierer liefert am „rechten" Ausgang die Summe (im Sinne von „Oder") seiner Inputs, die Multiplizierer am „unteren" Ausgang $x \cdot y$ bzw. $x \cdot \overline{y}$. Angemerkt sei, dass diese Bausteine auch leicht durch Gatter beschreibbar sind (vgl. Abbildung 7.4). Statt der in Abbildung 7.4 angegebenen Darstellungen verwenden wir jedoch jetzt nur noch Kästchen der in Abbildung 7.3 gezeigten Art, wobei wir durch eine Beschriftung jeweils angeben, um welchen Typ es sich handelt. Legen wir nun ein Format für ein aus diesen Bausteinen bestehendes Feld fest, so lassen sich damit sofort Schaltungen für eine Vielzahl von Funktionen angeben:

Beispiel 7.1 Wir wählen $n = 5$ Inputs an der linken Seite, $m = 5$ Outputs an der rechten Seite und $k = 4$ Spalten. Ein entsprechendes PLA sieht dann wie in Abbildung 7.5 gezeigt aus (die oberen Inputs und die unteren Outputs werden mit der Außenwelt nicht verbunden).

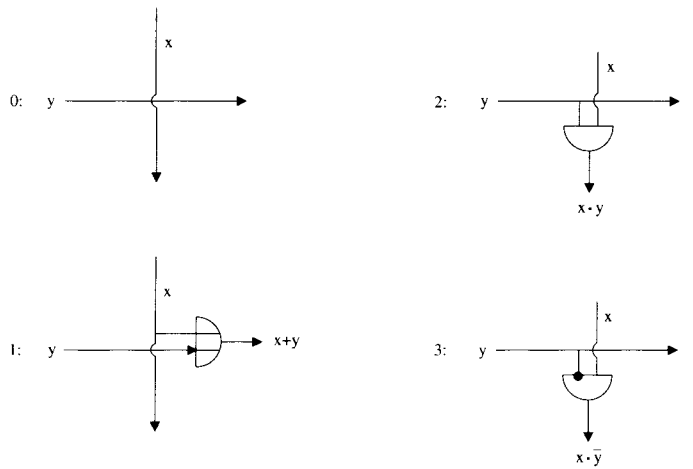


Abbildung 7.4: Realisierung der Bausteintypen eines PLAs.

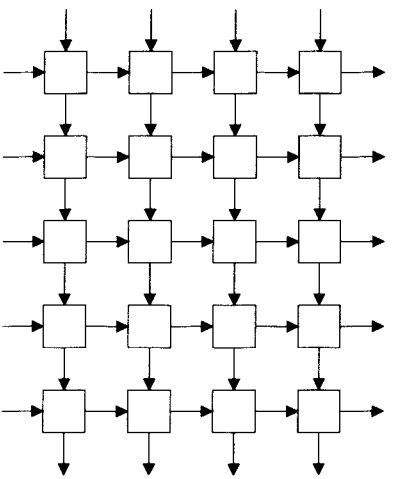


Abbildung 7.5: PLA-Schema zu Beispiel 7.1.

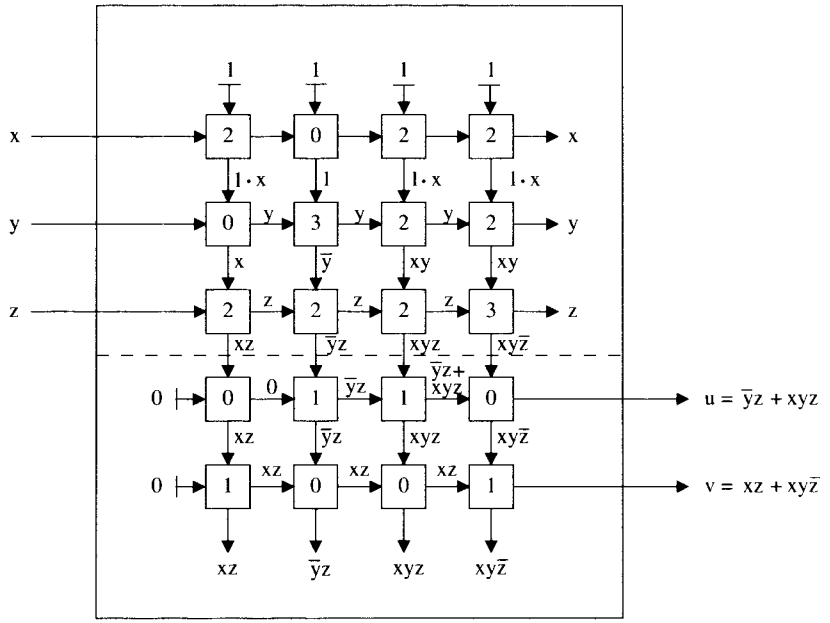


Abbildung 7.6: Interne Realisierung des PLA zu Beispiel 7.1.

Mit diesem PLA soll nun die Schaltfunktion

$$F : B^3 \to B^2, \text{ definiert durch}$$

$$F(x, y, z) := (\underbrace{\bar{y}z + xyz}_{u}, \underbrace{xz + xy\bar{z}}_{v})$$

realisiert werden. Dies bewerkstelligen wir wie folgt: Da nur drei der fünf vorhandenen Eingänge benötigt werden, „sperren" wir o. B. d. A. die unteren beiden durch Anlegen von Null. Wir „neutralisieren" die oberen Inputs der ersten Feld-Zeile durch Anlegen von Eins; von den Ausgängen brauchen wir in diesem Beispiel nur zwei. Da das Feld genau vier Spalten besitzt, können wir diese zur Erzeugung der vier Produktterme verwenden, welche wir für das Ergebnis benötigen; diese sind dann noch geeignet zu summieren und an die Ausgänge weiter zu leiten. Die eigentliche „Verschaltung" geschieht nun durch Eintragung von 0, 1, 2 oder 3 in die Kästchen des Feldes. Die Eintragung gibt dann jeweils an, um welchen Baustein-Typ es sich handelt. Der Leser möge verifizieren, dass die in Abbildung 7.6 gezeigte Realisierung obige Funktion F berechnet.

Nun ist unmittelbar klar, was das erwähnte „Sperren" mit 0 bzw. 1 bedeutet: Liegt am oberen Eingang eines Bausteins vom Typ 2 eine 1 an, so erscheint an *beiden* Ausgängen das links anliegende Eingangssignal; ist der Baustein vom Typ 3, erscheint am unteren Ausgang das Komplement des links anliegenden Inputs. Entsprechend bewirkt eine 0 am linken Eingang eines Bausteins vom Typ 1 ein „Auffächern" des oberen Inputs auf beide Outputs. □

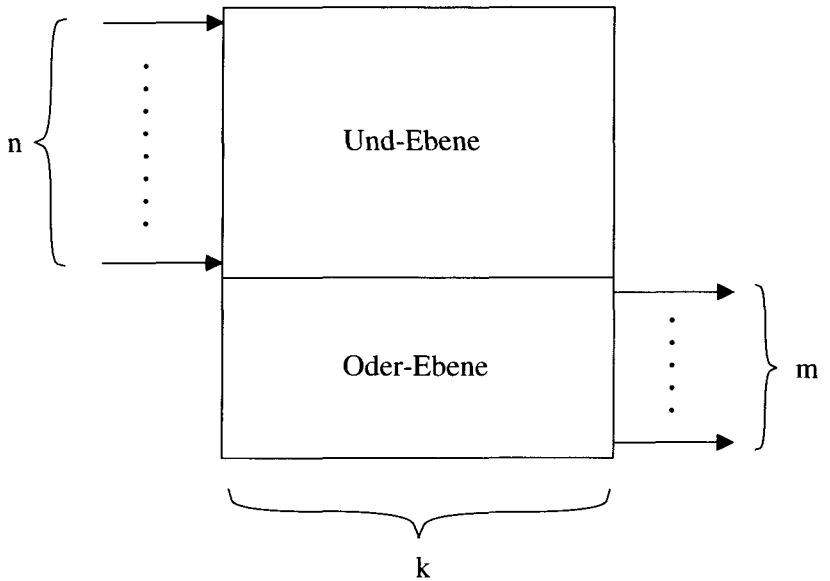


Abbildung 7.7: Logischer Aufbau eines PLAs.

Für die Funktion aus Beispiel 7.1 lässt die in Abbildung 7.6 angegebene Realisierung durch ein PLA eine typische Systematik erkennen: Oberhalb der gestrichelten Linie werden nur Bausteine der Typen 0, 2 oder 3 verwendet, darunter nur die Typen 0 oder 1. Ferner liegen die Inputs oberhalb dieser Linie an, die Outputs verlassen das Array darunter. Dies entspricht der generell verwendeten Trennung eines PLA in eine so genannte *Und-Ebene* (bestehend aus Identern und Multiplizierern) und eine *Oder-Ebene* (bestehend aus Identern und Addierern), so dass ein PLA grundsätzlich den in Abbildung 7.7 gezeigten logischen Aufbau hat. Wie aus Beispiel 7.1 hervorgeht, dient die Und-Ebene zur Erzeugung aller benötigten Produkt-Terme, die Oder-Ebene zur Erzeugung der entsprechenden Summen dieser Implikanten.

Formal verstehen wir unter einem PLA eine $(n + m) \times k$-Matrix, in welcher nur die Eintragungen 0, 1, 2 oder 3 nach dem beschriebenen Schema vorkommen. Dabei vereinbaren wir (vgl. Beispiel 6.1), dass in die Bausteine der obersten Zeile „von oben" Einsen gespeist werden, in die Bausteine der ersten Spalte „von links her" aber Nullen – sofern es sich hier um Mitglieder der Oder-Ebene handelt, d. h. nicht um Inputs. Wir werden von dieser Darstellungsform im Folgenden verschiedentlich Gebrauch machen.

Beispiel 7.1 (Fortsetzung): Das in Abbildung 7.6 angegebene PLA lässt sich kurz durch folgende Matrix beschreiben:

2	0	2	2
0	3	2	2
2	2	2	3
0	1	1	0
1	0	0	1

□

Dieser *normierte* Aufbau eines PLAs durch Trennung in Und- und Oder-Ebene bewirkt also einen Aufbau jeweils eines Produktes des Ergebnisses in einer eigenen Spalte und die Bildung der Summen schließlich in einer eigenen Zeile. Zur Realisierung einer beliebigen Schaltfunktion $F : B^r \to B^s$ benötigt man damit ein PLA mit (mindestens) r Zeilen in der Und-, s Zeilen in der Oder-Ebene. Ist F in disjunktiver Form gegeben und kommen in den einzelnen Summen insgesamt t verschiedene Produkt-Terme vor, muss das PLA (mindestens) t Spalten haben. Ist F speziell eine Boolesche Funktion, d. h. $s = 1$, so braucht die Oder-Ebene nur aus einer Zeile zu bestehen.

Zur Historie: Ein Beispiel für ein typisches PLA, welches in den USA in der zweiten Hälfte der 70er Jahre häufig verwendet wurde, war der Typ DM 7575 der Firma National Semiconductor. Dieses hatte 14 Inputs, 8 Outputs und 96 Spalten, d. h. es bestand aus einer Und-Ebene der Größe $14 \times 96 = 1344$, einer Oder-Ebene der Größe $8 \times 96 = 768$ und damit aus insgesamt 2112 Bausteinen. Mit diesem PLA ließen sich also theoretisch $(2^8)^{2^{14}} = 2^{131072}$ Schaltfunktionen der Form $F : B^{14} \to B^8$ realisieren. *Eine* vierzehn-stellige Boolesche Funktion kann aber bis zu 16384 Minterme besitzen, und nur 96 von ihnen waren in den Spalten des DM 7575 gleichzeitig generierbar. (Diese und die folgende Überlegung betrifft offensichtlich nur solche Schaltfunktionen, welche in einer Minterm-Darstellung vorliegen.) Dies bedeutete scheinbar eine starke Einschränkung; andererseits waren auf diese Weise immer noch $(2^8)^{96} = 2^{768}$ Funktionen schaltbar, und man konnte dadurch eine Fülle von Anwendungen abdecken.

Wir fassen die bisherigen Überlegungen zusammen:

Satz 7.1 *(PLA-Satz)* Durch geeignete Eintragung in die PLA-Matrix kann *jede* Schaltfunktion der gewünschten Dimensionierung realisiert werden.

Beispiel 7.1 zeigt exemplarisch, was dabei mit „Eintragungen in die PLA-Matrix" gemeint ist. In Bezug auf das DM 7575-PLA bedeutet dieser Satz: Alle Schaltfunktionen mit bis zu 14 Inputs, bis zu 8 Outputs, welche gegebenenfalls nach geeigneter Minimierung durch eine disjunktive Form mit maximal 96 Implikanten darstellbar sind, lassen sich durch entsprechende „Programmierung" dieses PLA realisieren. Wie diese Programmierung tatsächlich erfolgen kann, werden wir weiter unten erläutern.

Wir bemerken noch, dass eine Optimierung der gegebenen Schaltfunktion z. B. nach Karnaugh oder Quine-McCluskey in vielen Fällen nicht erforderlich ist:

Beispiel 7.1 (Fortsetzung): Für $u = \overline{y}z + xyz$ und $v = xz + xy\overline{z}$ gilt offensichtlich:

$$\begin{aligned} u &= (x + \overline{x})\overline{y}z + xyz = x\overline{y}z + \overline{x}\overline{y}z + xyz \\ v &= x(y + \overline{y})z + xy\overline{z} = xyz + x\overline{y}z + xy\overline{z} \end{aligned}$$

Schon die Mintermdarstellung enthält also — wie die optimierte Darstellung — nur vier verschiedene Produktterme, welche sich in den vier Spalten des oben angegebenen PLA wie folgt realisieren lassen:

2	2	2	3
2	2	3	3
2	3	2	2
1	0	1	1
1	1	1	0

Die erste Spalte realisiert dabei xyz, die zweite $xy\overline{z}$, die dritte $x\overline{y}z$, die vierte $\overline{x}\overline{y}z$. □

Allgemein ist also eine Optimierung z. B. einer disjunktiven Normalform nicht erforderlich, solange die Anzahl der verschiedenen Summanden die Dimensionierung des PLA (genauer: die Anzahl seiner Spalten) nicht übersteigt. Es können ferner kleine Änderungen einer Schaltfunktion in der Mintermdarstellung leichter berücksichtigt werden. Deshalb ist für eine PLA-Realisierung eine (nicht-optimierte) Mintermdarstellung im Allgemeinen sogar vorzuziehen.

7.3 Programmierung von PLAs

Wie können nun die Eintragungen in eine PLA-Matrix für eine konkret gegebene Schaltfunktion vorgenommen werden? Anscheinend bieten sich zwei Möglichkeiten an: Einerseits lässt sich eine spezielle Eintragung *hardwaremäßig* durch eine geeignete Ätz-Maske erzeugen, welche drei verschiedene Ätz-Muster (entsprechend den Baustein-Typen 1, 2 und 3) ermöglicht („Hardware-Knipszange"). Die regelmäßige Geometrie eines PLAs erleichtert ein präzises Arbeiten dieser Art erheblich. Andererseits ist ein PLA dann nur noch für eine bestimmte Anwendung einsetzbar, denn physikalische Veränderungen wie Ätzungen können nicht mehr — oder nur mit großen Schwierigkeiten — rückgängig gemacht werden.

Eine andere, wesentlich flexiblere Programmierung eines PLAs besteht darin, die Eintragungen *softwaremäßig* vorzunehmen. Wir erläutern dieses Prinzip zunächst für einen einzelnen Baustein: Da wir oben vier verschiedene Typen benutzt haben, und da sich die Zahlen 0, 1, 2 und 3 durch zweistellige Dualzahlen darstellen lassen, versehen wir jeden Baustein der oben genannten Art noch mit zwei programmierenden *Zuleitungen*, über welche dann eingegeben oder *programmiert* werden kann, wie sich der Baustein verhalten soll. Ein Baustein bekommt damit das in Abbildung 7.8 angegebene Aussehen: x und y sind wie bisher die Datenleitungen, s und t Zuleitungen, die angeben, nach welcher Vorschrift u und v in Abhängigkeit von x und y zu bilden sind. Wir beschreiben das Verhalten dieses Bausteins in Tabelle 7.1. Daraus liest man sofort ab:

$$\begin{aligned} u &= y + \overline{s}t.x. \\ v &= \overline{s}x + s.x(t \dotplus y). \end{aligned}$$

Eine Schaltung für diesen programmierbaren Baustein könnten wir mit den uns bekannten Methoden leicht angeben. Versehen wir nun in einem PLA der Größe $M =$

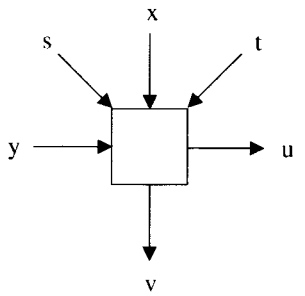


Abbildung 7.8: PLA-Baustein mit Zuleitungen.

Tabelle 7.1: Funktionale Beschreibung der Zuleitungen für ein PLA.

Baustein-Typ	s	t	v	u
0	0	0	x	y
1	0	1	x	$x+y$
2	1	0	$x \cdot y$	y
3	1	1	$x \cdot \overline{y}$	y

$(n + m) \times k$ die M Bausteine mit je zwei Zuleitungen, so kann offensichtlich jeder „Gitterpunkt" des PLAs durch entsprechende Ansteuerung zu einem bestimmten Verhalten veranlaßt werden. Die dazu benötigten $2M$ binären Informationen können dabei etwa in einem so genannten *Festwertspeicher* (engl. Read-Only-Memory, kurz ROM) abgelegt sein. Ein solcher Speicher ist dadurch gekennzeichnet, dass der in ihm einmal (z. B. durch Ätzen) abgelegte Inhalt nicht mehr veränderbar ist, also nicht mehr durch einen neuen Inhalt überschrieben werden kann. Ein ROM kann damit zur Programmierung eines PLAs verwendet werden, und durch Austausch des ROM lässt sich das PLA leicht umprogrammieren, d. h. dazu veranlassen, eine neue Schaltfunktion zu realisieren. Dadurch wird ein PLA also zu einem universell verwendbaren, „multi-purpose"-Baustein, wobei lediglich als gewisser Nachteil in Kauf zu nehmen ist, dass ein PLA-Programm, d. h. eine Bit-Folge der Länge $2M$, im Allgemeinen recht lang ist. Für den oben beschriebenen DM 7575-Baustein müßte ein entsprechendes ROM bereits $2 \cdot 2112 = 4224$ Bits speichern können. Mit der Kurzbeschreibung „1K" für $1024 = 2^{10}$ Bits benötigt man hier also ein ROM mit mehr als 4 K Bits Speicherkapazität. Dies kann heute jedoch nicht mehr als „Nachteil" angesehen werden, da die Preise für derartige Speicher derzeit jährlich um etwa 30% fallen.

Wir erwähnen noch eine andere, in der Literatur sehr häufig anzutreffende Darstellung der PLAs: Die strenge Trennung eines PLAs in Und- und Oder-Teil hat, wie oben erwähnt, zur Folge, dass man sich im Und-Teil auf die Baustein-Typen 0, 2 und 3, im Oder-Teil auf 0 und 1 beschränken kann. Im Und-Teil ist darüber hinaus Baustein 3 entbehrlich, wenn man die Anzahl der Inputs auf $2n$ verdoppelt und dann für jede Variable zusätzlich ihr Komplement in das Array hineinführt. In beiden Teilen des PLAs kommt man dann mit 2 Baustein-Typen aus (0/2 bzw. 0/1), von denen

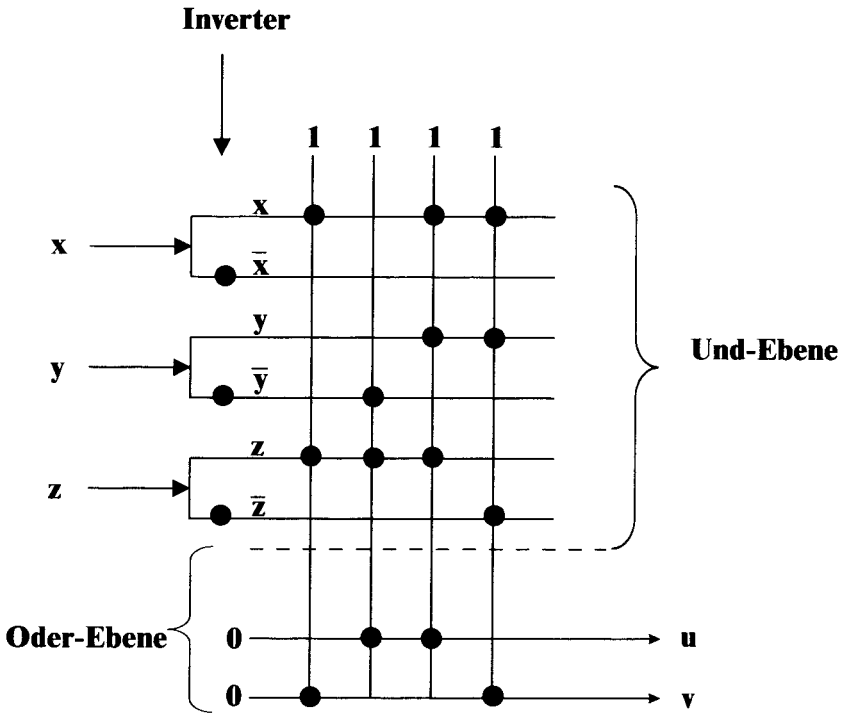


Abbildung 7.9: Alternative Darstellung des PLAs zu Beispiel 7.1.

dann in einem Gitter von Leitungen der von 0 verschiedene Typ durch einen Punkt gekennzeichnet wird. Wir nennen eine Darstellung von diesem Typ punkt-orientiert.

Beispiel 7.1 (Fortsetzung): Das in Abbildung 7.6 angegebene PLA erhält mit dieser Konvention das in Abbildung 7.9 gezeigte Aussehen. (Die Punkte in den Input-Verzweigungen bedeuten Inverter — im Gegensatz zu den Punkten im Inneren des PLAs.) □

Abbildung 7.9 illustriert ein interessantes Problem von PLAs: Die Universalität dieses Einheitsbausteins wird mit einer gewissen Redundanz erkauft. Speziell sind im Innern des in Abbildung 7.9 gezeigten PLAs in der Und-Ebene von 24 Gitterpunkten nur 10 „besetzt": in einer Chip-Realisierung würden nur an diesen 10 Stellen tatsächlich Schaltelemente angebracht.

Es gibt verschiedene Möglichkeiten, den Entwurf (Design) eines PLA für eine gegebene Schaltfunktion weiter zu optimieren. Einerseits können die aus Kapitel 3 bekannten Minimierungsverfahren dazu verwendet werden, die Anzahl der Spalten (also der Produktterme) zu reduzieren. Andererseits kann man durch eine so genannte *Faltung* die Anzahl der Zeilen der Und-Ebene verringern, wenn man Inputs findet, welche unterschiedliche Vertikal-Leitungen benötigen und dabei paarweise von links

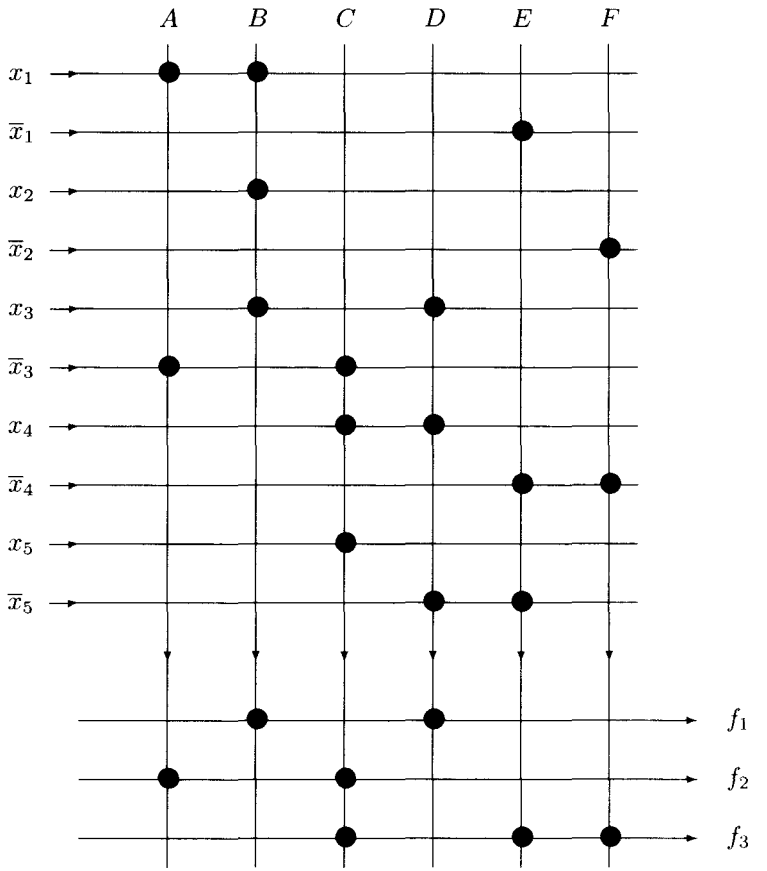


Abbildung 7.10: PLA für eine Funktion $F : B^5 \to B^3$.

bzw. von rechts her nicht kollidieren. So benutzt in dem in Abbildung 7.10 gezeigten Beispiel der Input x_1 die Drähte A und B, x_4 die sämtlich rechts davon liegenden Drähte C und D; ferner benutzt $\overline{x}_4$ die Leitungen E und F, x_5 den links von ihnen liegenden Draht C. Durch „Zusammenfächern" kann man hier Zeilen einsparen und überdies versuchen, durch kluge Anordnung der Vertikal-Leitungen dieses Verfahren zu optimieren. Bei großen Schaltungen kann der Rechenaufwand in der Entwurfsphase allerdings sehr groß werden, und dem Verfahren fehlt eine durchschlagende Akzeptanz, da offensichtlich durch diese Faltung nur eine Einsparung von höchstens 50% der Fläche der Und-Ebene möglich ist.

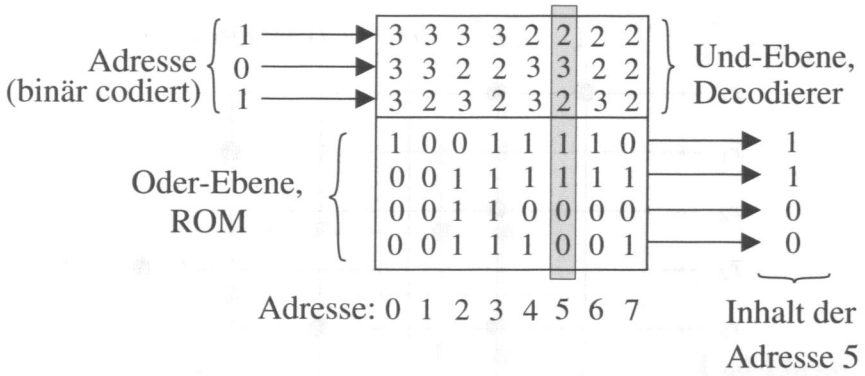


Abbildung 7.11: Beispiel eines Read-Only Memory (ROM).

7.4 Anwendungen von PLAs: ROMs und Mikroprogrammierung

Wir kommen nun auf einige *Anwendungen* der PLAs zu sprechen: Wichtig ist zunächst das bereits erwähnte ROM. Betrachten wir einen solchen Festwertspeicher für 2^n Worte der Länge m. Dieser kann aufgefasst werden als eine $m \times 2^n$-Matrix, deren Spalten den Adressen von 0 bis $2^n - 1$ entsprechen. Will man den Inhalt einer Adresse lesen, so kann man wie folgt verfahren: Man fasse das ROM als Oder-Ebene eines PLAs auf und erweitere diese durch Hinzunahme einer Und-Ebene der Dimensionierung $n \times 2^n$ zu einem PLA der Größe $(n + m) \times 2^n$, also mit n Inputs und m Outputs. (Diese Und-Ebene bezeichnet man häufig als *Adressdecodierer.*) Den Inhalt des ROM fasse man (zunächst) als Programmierung der Oder-Ebene des PLA auf, die Und-Ebene programmiere man (durch 2 und 3) so, dass in jeder Spalte genau der Minterm erzeugt wird, welcher diese Spalte dual codiert. Man beachte, dass dies genau die Technik des in Kapitel 2 behandelten Decoders ist. Gibt man dann eine ROM-Adresse i in Dualdarstellung ein, so wird nur in der i-ten Spalte eine Eins an den Oder-Teil übergeben, so dass der unter dieser Adresse stehende Wert ausgegeben wird.

Beispiel 7.2 Für $n = 3, m = 4$, sieht eine vollständige Lösung wie in Abbildung 7.11 angegeben aus, wobei der ROM-Inhalt zufällig gewählt ist. Wählt man z. B. über die Inputs die Adresse 5 an durch Eingabe von 101, so wird in der fünften Spalte der Und-Ebene (und nur in dieser) eine Eins erzeugt und in den unteren Teil weitergegeben; der genaue Ablauf ist wie in Abbildung 7.12 gezeigt. Völlig analog ist der weitere Ablauf in der Oder-Ebene; der Leser mache sich klar, wieso schließlich genau der Inhalt 1100 der Adresse 5 an den Ausgängen erscheint. □

Aus logischer Sicht ist ein ROM damit ein spezielles PLA, welches sogar mit nur *einer* Zuleitung pro Gitterpunkt auskommt, da in seiner Und-Ebene keine 0-Eintragungen vorkommen können (vgl. Abbildung 7.13. Für die praktische Verwendung eines PLA als ROM in der gerade beschriebenen Weise darf man erwarten, dass Ein- und Ausgänge der in Abbildung 7.13 gezeigten Schaltung jeweils mit einem Register versehen sind (wir werden diese Register im Zusammenhang mit dem

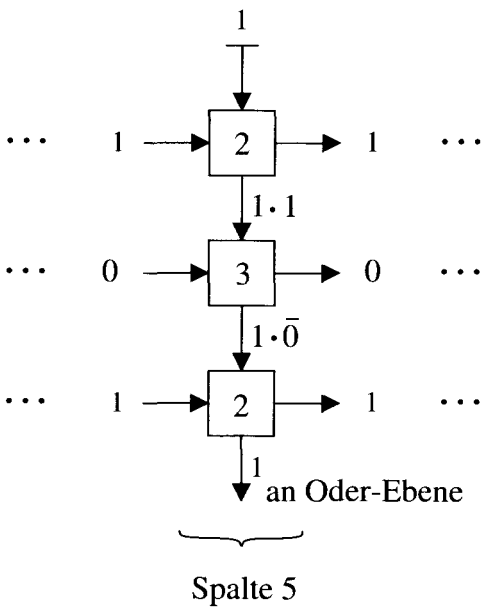


Abbildung 7.12: Auswahl einer Adresse (hier: 5) in einem ROM.

von Neumann-Rechnerkonzept auch als *Memory Address Register* für die Eingänge bzw. als *Memory Buffer Register* für die Ausgänge bezeichnen; man vergleiche hierzu Kapitel 8).

Eine weitere wichtige Anwendung der PLAs ist das von M. V. Wilkes schon 1951 vorgeschlagene Konzept der *Mikroprogrammierung*, auf welches wir jetzt zu sprechen kommen. Wir knüpfen dazu an Kapitel 5 an, in welchem wir Schaltungen mit Speicherelementen (Delays) kennen gelernt haben. Nicht nur Schaltnetze, sondern auch Schaltwerke können ein PLA in einfacher Weise benutzen, wenn man wenigstens einen Teil der Outputs über Delays wieder zu den Inputs zurückführt. Damit lässt sich dann auch ein (beliebiges) Schaltwerk durch ein um Delays erweitertes-PLA realisieren, prinzipiell wie in Abbildung 7.14 gezeigt. s der insgesamt $s+t$ Outputs werden über s Delays, d. h. ein s-stelliges Register, an s der $r+s$ Inputs zurück geleitet. Die übrigen t Outputs können als Steuerleitungen verwendet werden, welche Vorgänge anderwärts im Rechner anstoßen, die gemäß der momentanen Situation dort ausgelöst werden müssen. Alle überhaupt vorkommenden Delays sind also an einer Stelle der Schaltung übersichtlich untergebracht. Formal lässt sich dieses Schaltwerk wie in Kapitel 5 durch einen endlichen Automaten $A = (Q, \Sigma, \Delta, q_0, F, \delta)$ beschreiben, dessen Übergangsfunktion

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \to Q \times (\Delta \cup \{\epsilon\})$$

nun die spezielle Form

$$\delta : B^s \times (B^r \cup \{\epsilon\}) \to B^s \times (B^t \cup \{\epsilon\})$$

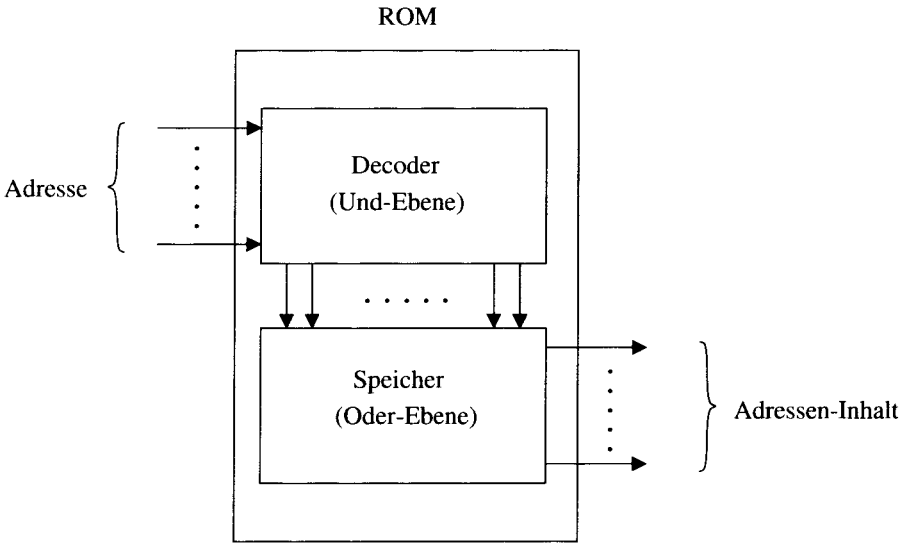


Abbildung 7.13: Anwendung eines PLA als ROM.

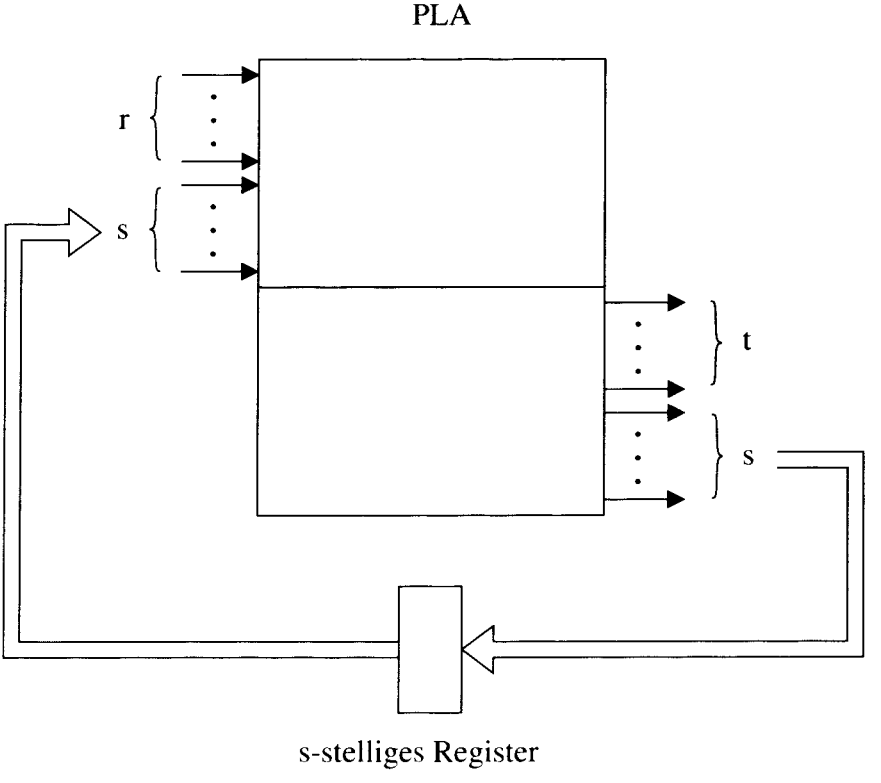


Abbildung 7.14: Realisierung eines Schaltwerks durch ein PLA.

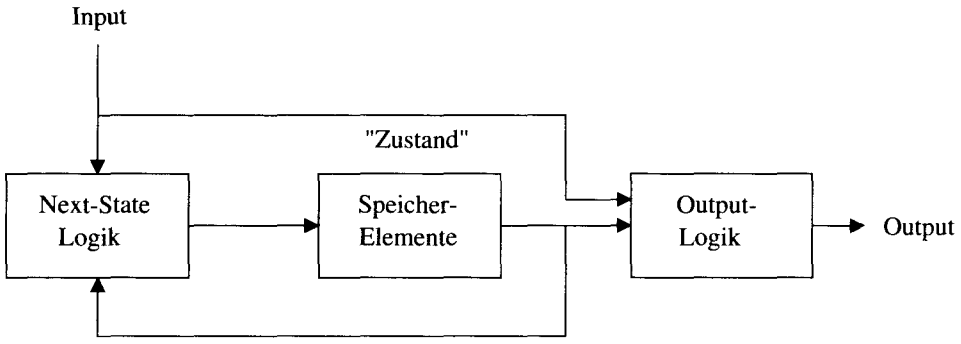


Abbildung 7.15: Prinzip eines sequentiellen Rechners.

annimmt. Die Berechnung des Folgezustands eines aktuellen Zustands erfolgt dabei durch die so genannte *Next-State-Logik*, die Berechnung des Outputs durch die *Output-Logik*. Dieses Vorgehen wird durch das in Abbildung 7.15 gezeigte „Prinzip-Schaltbild" des (klassischen) sequentiellen Rechners wiedergegeben.

Man beachte, dass unser Vorgehen so allgemein ist, dass auch umgekehrt *jeder beliebige* endliche Automat mit Ausgabe auf diese Weise realisiert werden kann. Damit ist gezeigt, dass der *abstrakte* Begriff des endlichen Automaten ein entscheidendes Hilfsmittel für das Verstehen der Arbeitsweise eines *konkreten* Rechners ist.

Beispiel 7.3 Realisierung eines dreistelligen Ringzählers im Gray-Code: $s = 3$ Outputs sind über ein dreistelliges Register an 3 Inputs rückzukoppeln; außerdem sei x ein weiterer (binärer) Input (d. h. $r = 1$), welcher etwa einen gewissen „Außenwelt"- Zustand angebe, und zwei weitere Outputs ($t = 2$) seien gemäß Tabelle 7.2 zu erzeugen. Daraus ist (z. B. nach Karnaugh) abzuleiten:

$$\begin{aligned} y_2 &= c_0 \\ y_1 &= x c_1 \bar{c}_2 + \bar{c}_0 c_1 \bar{c}_2 + c_0 \bar{c}_1 c_2 \\ C_2 &= \bar{c}_0 c_1 + c_0 c_2 \\ C_1 &= \bar{c}_0 c_1 + c_0 \bar{c}_2 \\ C_0 &= \bar{c}_1 \bar{c}_2 + c_1 c_2 \end{aligned}$$

Damit erhalten wir eine PLA-Realisierung (als Schaltwerk) in Matrix-Form wie in Abbildung 7.16 gezeigt. □

Es ist nun unmittelbar einzusehen, dass auch kompliziertere Schaltwerke wie z. B. das Von-Neumann-Addierwerk und also generell die entscheidenden Grundbausteine eines getakteten Rechners durch PLAs (gegebenenfalls mit Delays) realisierbar sind, wobei Next-State- *und* Output-Logik in *ein* PLA eingebaut werden können. PLAs mit Delays heißen auch „integrierte PLAs"; ein solches PLA, welches z. B. von der Firma HP in den 70er Jahren für die Verwendung in kleineren Taschenrechnern hergestellt wurde, bestand aus 8 Delays, 16 Ein- und 30 Ausgängen sowie 72 Spalten (d. h. $r = 16, s = 8, t = 30, n = 16 + 8 = 24, m = 30 + 8 = 38, k = 72$).

Tabelle 7.2: Funktionstafel des Ringzählers aus Beispiel 7.3.

c_2	c_1	c_0	x	C_2	C_1	C_0	y_1	y_2
0	0	0	0	0	0	1	0	0
0	0	1	0	0	1	1	0	1
0	1	1	0	0	1	0	0	1
0	1	0	0	1	1	0	1	0
1	1	0	0	1	1	1	0	0
1	1	1	0	1	0	1	0	1
1	0	1	0	1	0	0	1	1
1	0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0	0
0	0	1	1	0	1	1	0	1
0	1	1	1	0	1	0	1	1
0	1	0	1	1	1	0	1	0
1	1	0	1	1	1	1	0	0
1	1	1	1	1	0	1	0	1
1	0	1	1	1	0	0	1	1
1	0	0	1	0	0	0	0	0

Beispiel 7.3 zeigt eine für eine (endliche) Maschine typische Situation: In Abhängigkeit vom aktuellen Zustand und gegebenenfalls von (weiteren) externen Inputsignalen wird ein bestimmter Output erzeugt und ein neuer Zustand erreicht. Anwendung hierfür ist z. B. eine durch die Outputlogik erzeugte Werkzeugmaschinensteuerung, aber auch — besonders wichtig — ein Rechner selbst, und zwar in der Form, dass ein integriertes PLA als *Kontrolleinheit* fungieren kann, welche eine andere Funktionseinheit des Rechners (etwa die eigentliche „Recheneinheit") steuert.

Beispiel 7.4 Am Ende von Abschnitt 5.3 haben wir bereits erwähnt, dass z. B. ein Von-Neumann-Addierwerk auch zur Subtraktion und — im Rahmen eines größeren „Programms" — zur Multiplikation und Division von Dualzahlen verwendet werden kann. Damit besitzen wir ein universelles Bauteil für einen Rechner. Stellen wir uns nun vor, wir hätten ein solches Addierwerk geeignet durch ein PLA realisiert, so genügen zunächst zwei Steuerleitungen, um Addition oder Subtraktion anzustoßen oder ein Multiplikations- bzw. Divisionsprogramm zu starten:

Signal	s	t	PLA-Aktion
0	0	0	Addieren
1	0	1	Subtrahieren
2	1	0	Multiplizieren („Start")
3	1	1	Dividieren („Start")

Ein derart „von außen" gesteuerter Von-Neumann-Addierer könnte z. B. wie in Abbildung 7.17 gezeigt aussehen. □

Offen bleibt dabei nach wie vor, woher die Operanden bzw. die Steuersignale s und t kommen. Abweichend von der bisherigen Darstellung haben wir die Inputs und

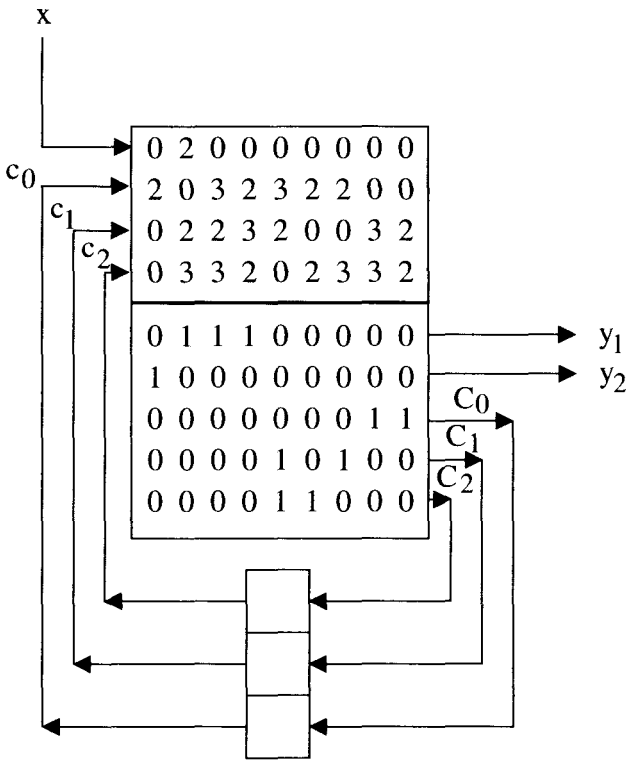


Abbildung 7.16: PLA zu Beispiel 7.3.

den Output nicht über das PLA laufen lassen. Wir nehmen stattdessen an, dass wir einen *Speicher* besitzen, welcher z. B. aus 10 achtstelligen Registern besteht, in dem der erste Operand auf Platz 2, der zweite auf Platz 3 steht und das Ergebnis auf Platz 1 abgelegt werden soll. Die „Steuerung", welche die Signale s und t generiert, muss dann also folgendes leisten:

1. Erkennen, welche Operation mit welchen Operanden auszuführen ist; sei dies etwa die Addition derjenigen Dualzahlen, welche auf den Plätzen 2 und 3 des Speichers stehen;
2. Transport dieser Zahlen in Akku bzw. Puffer;
3. Aktivierung des PLAs durch die Steuersignale $s = 0$ und $t = 0$ („Addition", siehe oben);
4. Speicherung des Ergebnisses auf Platz 1 des Speichers.

Als nächster Schritt könnte nun die Ausführung eines weiteren Befehls folgen. Die Steuerung muss also für jeden Maschinenbefehl einen spezifischen Ablauf erzeugen. Dazu kann man ein eigenes ROM verwenden. In diesem wird z. B. für einen Befehl, welchen der Benutzer durch „`ADD X,Y`" programmiert mit der Wirkung „$X \leftarrow X +$

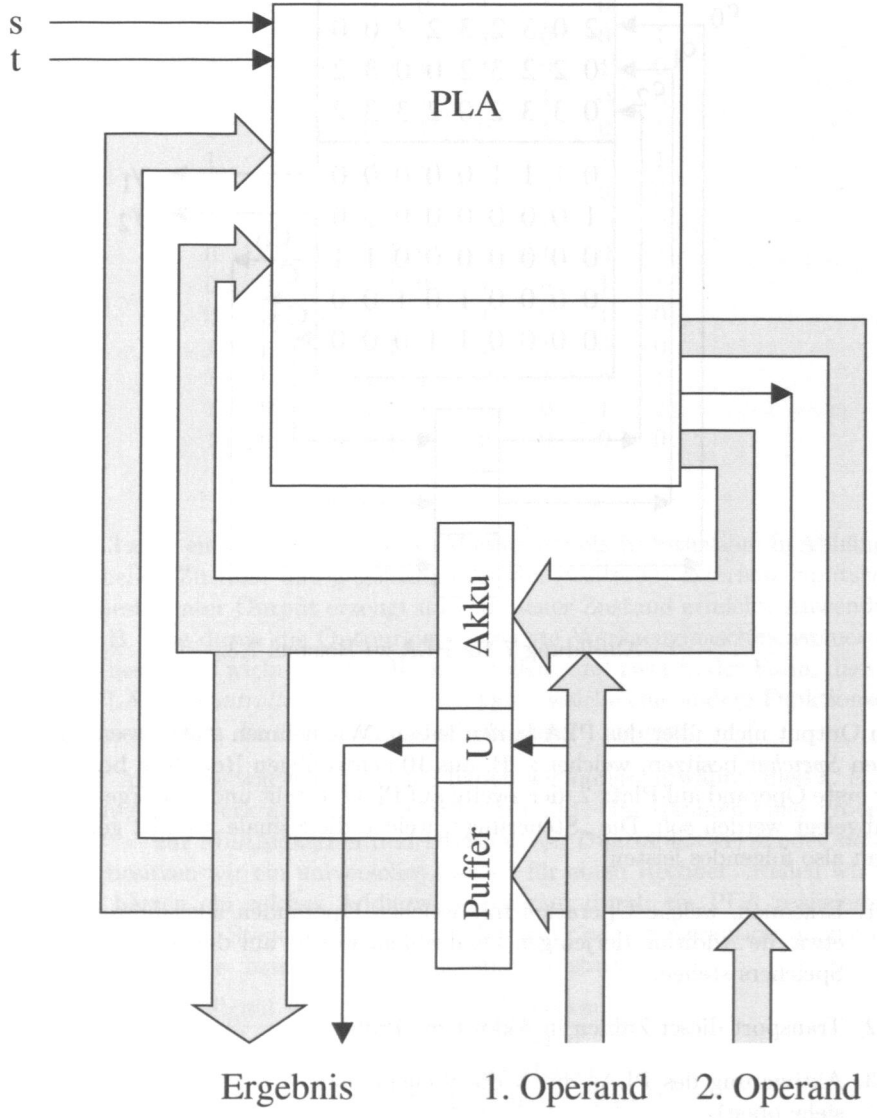


Abbildung 7.17: Prinzip eines Addierers bei Verwendung eines PLAs.

Y", die obige Schritt-Sequenz als eine Bit-Folge dargestellt. Sobald ein diesem ROM vorgeschalteter Decodierer (d. h. die zusätzliche Und-Ebene, welche das ROM zum PLA macht) erkennt, dass als nächstes dieser Befehl ausgeführt werden soll (d. h. eine entsprechende Spalte auswählt), wird der entsprechende ROM-Inhalt gelesen und dadurch der Ablauf obiger Schritt-Folge angestoßen.

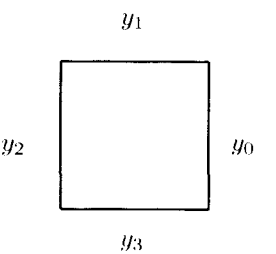
Damit kann die Folge dieser Schritte durch ein „Programm" realisiert werden, welches z. B. einen Addier-Befehl auf der Hardware-Ebene eines Rechners tatsächlich ausführt. Ein solches Programm nennt man *Mikroprogramm*, und das Erstellen eines Mikroprogramms, d. h. die Speicherung geeigneter Bit-Folgen in einem „Steuer-ROM", bezeichnet man als *Mikroprogrammierung*. Die Idee hierzu geht wie bereits erwähnt auf M. V. Wilkes zurück; wir werden darauf in Teil II zurück kommen.

7.5 Klassifikation von Logik-Designs

Eines der Hauptprobleme der Informationstechnologie ist seit altersher die *Realisierung* der logischen Hardware. Das Wirrwarr der ersten Computerschaltungen, welches beim Löten oder Stecken von Drahtverbindungen im Raum entstand, bedeutete für die Praxis — Produktion und Kontrolle — ein schweres Handicap. Die programmierbare Logik des PLA bedeutete deshalb einen gewaltigen Fortschritt, da normierte Bauteile (Chips) und eine konzeptionell einfache Beschriftungs- und Korrektur-Technik zusammen mit der logischen Technik der disjunktiven (Normal-)Formen sowohl das Design vereinfachte als auch die Produktion erleichterte. Das zweidimensionale Gitter bewährte sich dabei als eine besonders brauchbare Modell-Struktur. Hierbei bleibt die logische Oberstruktur des „Und"- und des „Oder"-Teils eines PLA fest.

Die Beschränkung auf eine zweidimensionale Gitterstruktur erscheint z.Zt. unverzichtbar. Man hat hierbei allerdings nicht viele Möglichkeiten zu einer Effizienz-Verbesserung. Die weiter oben erwähnte Faltung von PLAs ist ein erster Versuch, noch etwas mehr Chip-Fläche einzusparen.

Ein wichtiger Spezialfall von PLAs, bei denen ja grundsätzlich Und- *und* Oder-Ebene programmierbar sind, besteht darin, dass jeder Ausgang der Oder-Ebene eine feste (oder beschränkte) Anzahl von Implikanten-Bedingungen hat. Wir betrachten z. B. eine *4-Segment-Anzeige* eines Displays, welches zu Bitfolgen der Länge 4 eine korrespondierende Figur aus den Kanten oder Nicht-Kanten eines Quadrates wie folgt ausgeben soll: Indiziert man die Kanten des Quadrates in der angegebenen Weise,



so soll bei Eingabe der Dualzahl $x = (x_3x_2x_1x_0)_2$ die Kante y_i leuchten genau dann, wenn $x_i = 1$ ist. So leuchtet z. B. bei Eingabe von $x = (1111)_2$ das gesamte Quadrat.