Teil 2

Vom Bit zur Maschinensprache

9 Grundlagen der Komponierung informationeller Operatoren

Zusammenfassung

Die traditionelle Rechentechnik arbeitet fast ausschließlich mit *binär-statischer Codierung*, d.h. mit Codierung mittels zweier Zeichen, deren codierende Zustände (die materiellen Zustände, welche die Zeichen darstellen), statisch stabile Zustände des Trägermediums sind. Die Beschränkung auf zwei elementare Zeichen bedeutet wegen der Arbitarität der Codierung keine Einschränkung der Allgemeinheit. Ein Zustand heißt statisch stabil, wenn er sich nicht mit der Zeit verändert, sondern nur in einen anderen Zustand überspringen kann. Er heißt dynamisch stabil, wenn er ein stabil periodischer oder stabil repetierender Zustand ist, d.h. ein Zustand, in dem sich eine endliche Folge von Zuständen ständig wiederholt.

Die *elementaren* (d.h. nicht weiter dekomponierbaren) Operatoren eines informationellen Operators mit binär-statischer Codierung sind zwangsläufig die *elementaren booleschen* Operatoren. Unterhalb der elementaren informationellen Operatoren liegt der Bereich der kausalkontinuierlichen Prozesse. Den Übergang aus dem kontinuierlichen Bereich in den Bereich der informationellen, d.h. kausaldiskreten Prozesse leisten *Schwellenoperatoren*. Ohne sie ist auf dem Boden der klassischen Physik keine Informationsverarbeitung möglich. Doch sind sie nicht Bestandteil von traditionellen Computern, wohl aber von *Analog-digital-Konvertern*, die Computern in irgendeiner Form vorgeschaltet sein müssen. Bei Eingaben über die Tastatur durch den Nutzer fungiert dieser als Konverter. In Neurocomputern, die auf der Basis neuronaler Netze arbeiten, kommen Schwellenoperatoren in Form künstlicher Neuronen auch als *interne* Bausteinoperatoren zur Anwendung.

Um sämtliche booleschen Operationen komponieren und sämtliche booleschen Funktionen berechnen zu können, genügen einige wenige der insgesamt 16 elementaren booleschen Operatoren. Sehr häufig kommen der NOT-Operator (die Negation), der AND-Operator (die Konjunktion) und der OR-Operator (die Disjunktion) zum Einsatz. Jede Funktion mit endlicher Wertetafel kann im Prinzip (ohne Berücksichtigung des Aufwandsproblems) als *Kombinationsschaltung*, d.h. als zirkelfreies Netz elementarer boolescher Operatoren realisiert werden. Insbesondere kann sie auch als *disjunktive* Kombinationsschaltung realisiert werden. Jede als Kombinationsschaltung realisierbare Funktion ist eine rekursive Funktion.

Kombinationsschaltungen lassen sich zu zirkulären Netzen verbinden, wenn in jeden Zirkel (in jede Rückkopplungsschleife) ein Speicher mit vorgeschaltetem Tor eingebaut wird. Die Speicher erübrigen sich, wenn das Netz als Ganzes *Eigenzustände* besitzt, wenn es also als Ganzes einen Speicher darstellt. Das einfachste zirkuläre boolesche Netz, das sich als Speicher verwenden lässt, ist der *Flipflop* mit Eingangstoren. Er kann ein Bit speichern.

Informationelle Operatoren mit binär-statischer Codierung sind notwendigerweise Netze aus Kombinationsschaltungen und Speichern, wobei in jeder Verbindung (in jedem Operandenweg) zwischen zwei Kombinationsschaltungen ein Speicher mit Eingangstor liegen muss. Das Entsprechende gilt für Neurocomputer mit statischer Codierung, wobei an die Stelle der Kombinationsschaltungen zirkelfreie Netze aus Schwellenoperatoren treten.

Für eine universelle Komponierungsmethode informationeller Operatoren gilt die *Vollständigkeitsforderung*: Sämtliche im Prinzip möglichen informationellen Operatoren mit statischer Codierung müssen komponierbar sein.

9.1 Statische und dynamische Codierung

Mit diesem Kapitel beginnen wir die Behandlung der zentralen Frage des Buches: Wie lässt sich Informationsverarbeitung technisch verwirklichen oder - im Sinne unserer Definition der Informatik - wie lässt sich ein technisches System aufbauen, das zur aktiven, sprachlichen Modellierung fähig ist?

Wir wollen das Gebiet der Computertechnik nicht nur als wissbegierige Interessenten betreten, die wissen wollen, “was es alles gibt” und “was man alles machen kann”, sondern vor allem als Forscher und Erfinder, die sich überlegen, wie etwas funktioniert oder wie sich eine gewünschte Funktion realisieren lässt, z.B. das Addieren, das Gedächtnis oder das Schlussfolgern. In diesem Geiste werden wir die Gefilde der technischen Informationsverarbeitung erkunden und versuchen, ihre Ideen und Konzepte nachzuerfinden. Dabei werden wir vom Trägerprinzip ausgehen und konsequent die Methoden des hierarchischen Komponierens und der uniformen Systembeschreibung anwenden. Zunächst werden wir weniger Wert auf Systematik, dafür umso mehr Wert auf die gedankliche und logische Folgerichtigkeit der wesentlichen Ideen legen. In den Kapiteln 11 und 18 werden die Resultate unserer Gedankengänge in ihren systematischen und historischen Zusammenhang gestellt.

Im Sinne des Trägerprinzips geht die USB-Methode von der Tatsache aus, dass Informationsverarbeitung in einem stofflichen Träger abläuft. Demzufolge geht sie nicht, wie die Mathematik und die theoretische Informatik, von den Begriffen *Operation* und *Funktion* aus, die vom stofflichen Träger abstrahieren, sondern vom Begriff des *realen Operators*. Ebenso geht sie nicht von einem abstrakten Operandenbegriff, nicht von objektivierten *Idemen* aus, sondern von *Realemen*, von Zuständen des stofflichen Mediums, das die Operanden (die Zeichenketten) “*trägt*”, in das die Zeichen in Form codierender Zustände *eingeprägt* sind.

Auf diese Weise werden wir ein Begriffsgebäude der Informatik “konstruieren”, das auf einem *stofflich-physikalischen* Fundament steht, auf dem Fundament des *Naturnotwendigen*. Man kann ein solches Gebäude auch auf einem rein *logischen* Fundament aufbauen, auf dem Fundament des *Denknotwendigen*. Das hat den Vorteil, dass man weniger in philosophische, insbesondere erkenntnistheoretische

Fragen verwickelt wird. Der logische Weg wird beispielsweise in einem Buch mit dem Titel "Nichtphysikalische Grundlagen der Informationstechnik" [Wendt 89] beschritten. Die "kausale Informatik" des vorliegenden Buches stellt in gewissem Sinne ein Pendant oder eine Ergänzung des Buches von S.Wendt dar, indem sie die *nichtphysikalischen* Grundlagen *physikalisch* (kausal) untermauert.

Wenn wir mit dem Konstruieren, d.h. mit dem Komponieren von Operatorenhierarchien im Sinne des Trägerprinzips beginnen, müssen wir zuerst klären, welche Art von Zuständen sich als codierende Zustände eignen und aus welchen elementaren Operatoren Kompositoperatoren zu komponieren sind. Wir beginnen mit der ersten Frage und präzisieren sie:

Wie lassen sich Zeichen (elementare Zeichen und Kompositzeichen) *materialisieren, d.h. materiell codieren* (materiell instanzieren; vgl. Pfeil 3 in Bild 2.1), *sodass sie nicht verloren gehen, sondern solange erhalten bleiben ("gespeichert" sind), wie sie gebraucht werden*?

Die Antwort lautet: Die Codierung muss durch stabile Zustände eines geeigneten Trägermediums erfolgen, genauer gesagt, durch Zustandsparameter, die ausreichend stabil sind. Wir nennen sie **codierende Zustandsparameter** und die Zustände selber **codierende Zustände**.

Es sind zwei Arten von Stabilität zu unterscheiden, statische und dynamische. *Ein (makroskopischer) Zustandsparameter* (z.B. Lage, Temperatur, Magnetisierung, Ladung) *heißt* ***statisch stabil****, wenn er sich nicht mit der Zeit verändert. Er heißt* ***dynamisch stabil****, wenn er sich periodisch oder repetierend verändert,* m.a.W. wenn eine bestimmte zeitliche Folge von Parameterwerten sich ständig wiederholt. Dynamisch stabil (bezogen auf einen angemessenen Zeitmaßstab) ist z.B. die Planetenbewegung, die Rotation eines Kreisels und die Schwingung eines Pendels oder eines elektrischen Schwingkreises. Ein dynamisch stabiler Zustand, d.h. ein Zustand mit mindestens einem dynamisch stabilen Parameter, ist zirkulärer Natur. Mikroskopisch (statistisch oder quantenmechanisch[1]) betrachtet gibt es keine statischen, sondern nur dynamische Materiezustände. *Codierung, die statisch stabile bzw. dynamisch stabile codierende Zustände verwendet, heißt* ***satische*** *bzw.* ***dynamische Codierung***.

In Kap.2.1 war von drei Evolutionen die Rede. Die dortigen Überlegungen lassen sich durch folgende Feststellung ergänzen: *Genetische und kulturelle Evolution beruhen auf* ***statischer****, die intellektuelle Evolution des Individuums offenbar auf* ***statischer und dynamischer*** *Codierung.* Das soll kurz erläutert werden.

Die *kulturelle Information,* also die Information, die im Verlauf der kulturellen Evolution gespeichert und weitergegeben wird, ist, soweit sie nicht mündlich überliefert wird, statisch codiert, zunächst in Bauwerken, Plastiken und Bildern, später

1 Alle physikalischen Überlegungen des Buches gehen von der klassischen Physik aus, d.h. es werden nur solche Prozesse betrachtet, in die relativ große Energien involviert sind, sodass das plancksche Wirkungsquantum vernachlässigt werden kann.

auch schriftlich, seit 500 Jahren vorzugsweise durch Bedrucken von Papier, neuerdings auch durch das "Beschreiben" von Datenträgern, z.B. von Kassettenbändern, Disketten oder CDs. Die codierenden Zustände sind Schwärzungs-, Magnetisierungs- oder andere Zustände des Datenträgers. Vorwiegend sind es Oberflächenzustände.

Die *genetische Information* ist bekanntlich in großen Molekülen, den DNS, codiert. Die codierenden Zustände sind statisch stabile Strukturen, die aus vier verschiedenen organischen Basen als Bausteinen komponiert sind. Jeder Baustein ist eine Ringverbindung aus Kohlenstoff-, Wasserstoff-, Sauerstoff- und Stickstoffatomen. Die Basen kann man als elementare Zeichen der genetischen "Schrift" auffassen und sagen, dass das genetische Alphabet 4 Buchstaben enthält.

Bei der Beschriftung der gängigen technischen Datenträger werden nur zwei Zeichen verwendet, es wird binär codiert. Das gilt generell für die interne Informationsdarstellung in Computern, m.a.W. für computerinterne Zeichenrealeme. Dieses
1 Vorgehen war in Kap.5.6 [5.20] begründet worden. Das menschliche Gehirn scheint eine kombinierte statisch-dynamische Codierung zu verwenden. Die Ergebnisse der Gehirnforschung legen die Annahme nahe, dass strukturelle Zustände des Zentralnervensystems (z.B. Verknüpfungsmuster zwischen den Neuronen) der statischen Codierung dienen, während Anregungszustände des Nervensystems der dynamischen Codierung dienen. Wenn der experimentelle Befund, dass ein Neuron nur einen einzigen stabilen Zustand besitzt, seinen Ruhezustand, dann ist statische Codierung durch Anregungszustände neuronaler Netze unmöglich.

Für unsere weiteren Überlegungen treffen wir folgende Festlegung: *Die Hardware, die in den folgenden Kapiteln schrittweise aufgebaut wird, soll mit* ***binärer***, ***statischer Codierung*** *arbeiten.* Die Beschränkung auf zwei elementare Zeichen bedeutet wegen der Arbitarität der Codierung keine Einschränkung der Allgemeinheit. Ob die Beschränkung auf statische Codierung eine Einschränkung bedeutet, werden wir später diskutieren.

9.2 Elementare informationelle Operatoren

9.2.1 Elementare boolesche Operatoren und die erste Grundidee des elektronischen Rechnens

Gemäß der vereinbarten Beschränkung auf binär-statische Codierung präzisiert sich unsere Aufgabe: *Es ist ein universelles informationelles System mit binär-statischer Codierung als Operatorenhierarchie zu entwerfen.* Das bedeutet, dass die Operanden *Bitketten* sind, auch *Binärwörter* genannt, und dass das System und seine Bausteinoperatoren Bitketten transformieren; darum nennen wir sie **Bitkettenoperatoren** oder **Binärwortoperatoren**.

Bei der Verwirklichung dieses Ziels werden wir uns folgender **Vollständigkeitsforderung** unterwerfen. *Keine Entwurfsentscheidung darf den Bereich der realisier-*

baren informationellen Operatoren mit binär-statischer Codierung einschränken, m.a.W. ***sämtliche im Prinzip möglichen informationellen Operatoren mit statischer Codierung müssen auf dem Wege, den wir gehen werden, komponierbar sein***. Wenn wir uns beim Systementwurf an diese Bedingung halten, werden wir am Ende sagen können: Das entworfene System ist "universell" in folgendem Sinne. Das System kann jede mittels statischer Codierung realisierbare Funktion berechnen und es gibt keinen informationellen Operator mit statischer Codierung, der mehr oder etwas anderes zu leisten vermag, als das entworfene System. Auch die Turingmaschine - sie "arbeitet" mit statischer Codierung - kann nicht mehr leisten. *Eine Funktion, die von einem Operator berechnet werden kann, der mit statischer Codierung arbeitet, heißt* ***statisch berechenbar***.

Der erste Schritt zur Lösung der Aufgabe ist die Festlegung der elementaren Operatoren. Der Allgemeingültigkeit halber, d.h. zum Zwecke der Vermeidung irgendwelcher Einschränkungen des Komponierens von Operatorenhierarchien, müssen die denkbar "elementarsten" Operatoren zugrunde gelegt werden, in die sich ein informationelles System dekomponieren lässt. Das bedeutet, dass ein elementarer Operator bei weiterer Dekomponierung den Charakter eines Operators, der Zeichen verarbeitet, verliert und dass die kausaldiskrete durch eine kausalkontinuierliche Beschreibung ersetzt werden muss.

Bevor wir die Suche nach den elementaren Operatoren informationeller Systeme beginnen, wollen wir, ausgehend vom Trägerprinzip, die gesuchten Operatoren genauer charakterisieren. Gesucht werden *reale Operatoren, die nicht weiter dekomponierbar sind und die ausschließlich den* ***Zuordnungsprozess*** *tragen*, d.h. den Prozess der Verarbeitung statisch codierter Operanden, die jedoch nicht die Operanden selber tragen (statisch codieren). Die gesuchten Operatoren besitzen keine stabilen Zustände, also kein Gedächtnis. Besäßen sie ein Gedächtnis, könnten sie in Zuordner und Speicher dekomponiert werden.

Zuordnungsprozesse in elementaren Operatoren nennen wir **Übergangsprozesse**. 2
Übergangsprozesse beginnen mit der Eingabe von Eingabeoperanden und enden mit der Ausgabe der zugeordneten Ausgabeoperanden. Aus der Sicht der kausaldiskreten Beschreibung verbindet ein Übergangsprozess zwei zeitlich benachbarte Ereignisse und überbrückt den nicht beschriebenen kausalkontinuierlichen Übergangsprozess. Der Übergangsprozess enthält keinen Zeitpunkt, der ein Ereignis der kausaldiskreten Beschreibung darstellt. Enthielte er einen solchen Zeitpunkt, könnte der Operator dekomponiert werden.

Die Schlussfolgerung unserer Überlegung lautet: *Die* ***elementaren Operatoren*** 3
informationeller Systeme mit statischer Codierung besitzen ***kein Gedächtnis***; sie können "sich nichts merken". Der Ausgabeoperand (das Resultat) einer Operation liegt höchstens solange am Ausgang, wie der Eingabeoperand am Eingang liegt. Danach "wird er vergessen". Das kann durch Vorschalten eines Speichers verhindert werden.

Nach dieser Vorüberlegung beginnen wir die Suche nach konkreten Operatoren, die als elementare Operatoren verwendbar sind. Im Weiteren wird das einzelne Bit als spezielle Bitkette und ein Operator, der einem Bit ein Bit zuordnet (*Einbitoperator*) als spezieller Bitkettenoperator angesehen. Der Einbitoperator ist der denkbar einfachste Bitkettenoperator. Doch ist er als alleiniger elementarer Operator für das Komponieren von Kompositoperatoren nicht ausreichend. Man kann mit seiner Hilfe zwar Operatoren komponieren, die einzelne Bits in Bitketten transformieren, aber keine Operatoren, die Bitketten in Bits oder in Bitketten transformieren. Dafür sind Bausteinoperatoren mit mindestens zwei Eingängen (bzw. Bausteinoperatoren mit vorgeschalteter Vereinung) erforderlich.. Die einfachsten Operatoren, die diese Bedingung erfüllen, überführen Bitpaare in Bits. Derartige Operatoren zusammen mit den Einbitoperatoren heißen **elementare boolesche Operatoren**. Die Funktionen, die sie realisieren, heißen **elementare boolesche Funktionen**. Mit dieser Bezeichnung wird der englische Mathematiker GEORGE BOOLE (1815-1869) geehrt, der bei der Analyse logischer Gesetzmäßigkeiten durch Abstraktion zur Definition eines Kalküls gelangte, der nach ihm **boolesche Algebra** genannt wird (siehe Kapitel 11.1).

4 *Die elementaren Operatoren informationeller Systeme mit statischer Codierung sind also die **elementaren booleschen Operatoren***. Netze aus elementaren booleschen Operatoren nennen wir **boolesche Netze**. Wir werden uns zunächst nur für starre, zirkelfreie Netze interessieren. Solche Netze werden wir im Weiteren **Kombinationsschaltungen** nennen. Sie enthalten keine Weichen und keine Rückkopplungsschleifen, sondern nur starre Flussknoten (Gabeln und Vereinungen) und evtl. starre Maschen.

Kombinationsschaltungen, die ein einziges Bit als Ausgabeoperand liefern, heißen **boolesche Operatoren**. Die Funktionen, die sie realisieren, heißen **boolesche Funktionen**. Kombinationsschaltungen sind Bitkettenoperatoren (Binärwortoperatoren, informationelle Operatoren) der ersten Komponierungsebene, sofern sie unmittelbar (ohne Zwischenschritt) aus den elementaren booleschen Operatoren komponiert sind. Sie können aber auch als Operatoren der zweiten Komponierungsschicht aufgefasst werden, denn sie lassen sich in boolesche Operatoren dekomponieren, sodass jede Stelle der Ausgabebitkette durch je einen booleschen Operator berechnet wird. Die Komponierung und Untersuchung von Kombinationsschaltungen wird in Kap.9.3 mit mathematischer Exaktheit ausgeführt. Dort wird auch verständlich, wie es zu der vielleicht etwas merkwürdig anmutenden Bezeichnung gekommen ist.

Die Idee, als elementare Operatoren der technischen Informationsverarbeitung elementare boolesche Operatoren zu verwenden, m.a.W. einen universellen Rechner aus elementaren booleschen Operatoren zu komponieren, ist inhaltlich identisch mit der **ersten Grundidee des elektronischen Rechnens**, die darin besteht, die boolesche Algebra hardwaremäßig zu realisieren. Denn mit dem Aufbau einer Operatorenhierarchie aus elementaren booleschen Operatoren wird de facto die boolesche

Algebra realisiert. Wir haben diese “Idee” soeben aus dem Prinzip der binär-statischen Codierung “abgeleitet”.

Die erste Grundidee des elektronischen Rechnens scheint im Widerspruch zu der Aussage aus Kap.8.2.4 [8.13] zu stehen, dass ohne Schwellenoperatoren keine Informationsverarbeitung möglich ist, weil nur sie den Übergang aus dem nichtsprachlichen, kontinuierlichen Bereich in den sprachlichen, diskreten Bereich bewerkstelligen können. Demzufolge müssten Schwellenoperatoren die elementaren Operatoren informationeller Operatoren sein. Wir wollen dem Widerspruch nachgehen, obwohl wir damit den Weg zum *traditionellen* Computer, den wir entwerfen wollen und den wir später **Prozessorcomputer** nennen werden, verlassen und den Weg zum sogenannten **Neurocomputer** betreten. Wir werden also kurzzeitig die *traditionelle Informatik* verlassen und einen Abstecher in die *Neuroinformatik* machen.

Wir wollen versuchen, Kompositoperatoren aus Schwellenoperatoren zu komponieren. Dazu müssen zunächst **mehrstellige Schwellenoperatoren** (Schwellenoperatoren mit mehreren Eingängen) eingeführt werden, um Netze aufbauen zu können. Ein Schwellenoperator mit n Eingängen berechnet eine Funktion

$$y = f(x_1,x_2,...x_n), \tag{9.1}$$

worin die x_i reelle Größen sind, während y nur zwei Werte annehmen kann, die in Bild 4.1 mit 0 und 1 bezeichnet sind. Ob y den Wert 0 oder 1 annimmt, hängt davon ab, ob eine Funktion $g(x)$, die das Schwellenverhalten explizit beschreibt und **Schwellenfunktion** genannt wird, einen **Schwellenwert** s überschreitet oder nicht. Im einfachsten Fall kann die Schwellenfunktion eine Stufe sein, wie sie in Bild 4.1 dargestellt ist. Für eine n-stellige stufenförmige Schwellenfunktion gilt:

$$\begin{aligned} &y = 0, \text{ wenn } g(x_1,x_2,...x_n) \le s \\ &y = 1, \text{ wenn } g(x_1,x_2,...x_n) > s. \end{aligned} \tag{9.2}$$

Als Funktion g wird häufig die Summe der x_i verwendet. Unter Benutzung von ein- und mehrstelligen Schwellenoperatoren lassen sich Kompositoperatoren aufbauen. Doch scheint es wenig sinnvoll zu sein, Schwellenoperatoren hintereinander zu schalten, denn der Nachfolgeoperator empfängt von seinem Vorgänger bereits einen diskreten Wert, eine 0 oder eine 1. Wenn er mehrere Vorgänger hat, empfängt er - über eine Vereinung - ein Paar oder ein Tupel von Nullen und Einsen. Ein innerer Operator überführt also Bitketten in Bits, er ist ein *boolescher Operator*.

Ein Netz, dessen innere Operatoren boolesche Operatoren und dessen Eingangsoperatoren Schwellenoperatoren sind, arbeitet intern als Bitkettenoperator, dessen Eingabeoperanden reelle Wertetupel sind, sodass er mit der kontinuierlichen Umwelt kommunizieren kann. Die Schwellenoperatoren übernehmen die Funktion eines Analog-digital-Konverters.

Damit ist der Widerspruch ausgeräumt. Man könnte fragen, wo sich der Konverter verbirgt, wenn ein Mensch Daten in einen PC, also in einen reinen Binärwortoperator

eingibt? Die Antwort lautet: Der Mensch selber übernimmt die Funktion des Konverters, beispielsweise, wenn er die Tastatur des PC betätigt. Denn dabei wird die stetige (analoge) Bewegung seines Armes und seiner Finger in den binären Interncode des getasteten Zeichens überführt. (Genau genommen ist die Armbewegung eine kurze analoge Zwischenetappe zwischen der Codierung in Zentralnervensystem und der Codierung im Computer.) Früher erfolgte das Konvertieren durch Stanzen von Löchern in Karten oder Papierstreifen, ein Relikt aus der Zeit der Hollerithmaschinen, das inzwischen fast ausgestorben ist.

Hier unterbrechen wir den Abstecher in die Neuroinformatik, um ihn in den Kapiteln 9.2.2 und 9.4 fortzusetzen.

Man könnte auf die Idee kommen, einen “universellen” Computer dadurch zu bauen, dass man für jede Funktion, die er berechnen soll, eine Kombinationsschaltung entwirft und herstellt. In Kap.9.3 [9.3] wird sich zeigen, dass dies im Prinzip möglich ist. Doch kann man unschwer erkennen, dass sich die Idee nicht verwirkli-
6 chen lässt. Nehmen wir an, wir wollten einen Computer bauen, der 3-stellige ganze positive Dezimalzahlen addiert. Die Eingabeoperandenpaare bestehen dann aus 6 Ziffern, die Ausgabeoperanden aus maximal 4 Ziffern. Bei ziffernweiser binärer Codierung durch 4-stellige Bitketten pro Ziffer (3-stellige Bitketten reichen nur für 8 der 10 arabischen Ziffern aus) beträgt die Länge des Eingabewortes 24 Bit, die des Ausgabewortes 16 Bit. Jedes der 16 Ausgabebits ist (im Prinzip) eine Funktion der 24 Eingabebits. Der Computer muss also 16 verschiedene 24-stellige boolesche Funktionen berechnen. Um uns zu überzeugen, dass unser Ziel auf diesem Wege nicht zu erreichen ist, schätzen wir ab, wie viele 10-stellige boolesche Funktionen es gibt (wir begnügen uns mit 10 Stellen, um die Abschätzung zu vereinfachen).

Zunächst fragen wir nach der Anzahl der Argumentwerte, d.h. nach der Anzahl der möglichen Eingabebitketten des zu realisierenden booleschen Operators. Da jede Stelle zwei Werte annehmen kann, beträgt die gesuchte Anzahl 2^{10}, wofür sich in der Informatik die Schreibweise 2^10 eingebürgert hat in Anpassung an die PC-Tastatur. Eine 10-stellige boolesche Funktion ordnet jedem dieser Argumentwerte entweder eine 0 oder eine 1 zu. Für die Menge aller Argumentwerte gibt es also 2^(2^10) verschiedene Abbildungen auf die Menge {0,1}, also gibt es ebenso viele Funktionen. Allgemein gilt:

$$\text{Anzahl } n\text{-stelliger boolescher Funktionen} = 2\hat{}(2\hat{}n). \tag{9.3}$$

Hinsichtlich Dualzahlen ist unser Gefühl für Größenordnungen i.Allg. wenig entwickelt. Darum wollen wir die Anzahl der 10-stelligen booleschen Funktionen näherungsweise als Zehnerpotenz angeben. Dafür nutzen wir den Umstand, dass 2^{10} gleich 1024, also etwa gleich 10^3 ist. Wir schreiben $2^{10} \approx 10^3$.

Wenn man beide Seiten dieser Näherungsgleichung in die (n/10)-te Potenz erhebt, d.h. die Exponenten auf beiden Seiten mit (n/10) multipliziert, ergibt sich (9.4a); wenn man beide Seiten mit n/3 multipliziert, ergibt sich (9.4b).

$$2^n \approx 10^{(3/10)n} \tag{9.4a}$$

$$10^n \approx 2^{(10/3)n}. \qquad (9.4b)$$

Diese beiden Näherungsformeln können beim Übergang zwischen Dual- und Dezimalzahlen gute Dienste leisten. Die Anzahl der 10-stelligen booleschen Funktionen ist nach (9.3) gleich 2^(2^10) also etwa gleich 2^{1000}, was nach (9.4a) etwa gleich 10^{300} ist. Der Bau von 10^{300} realen Operatoren übersteigt die Möglichkeiten des Weltalls. Der Ausweg könnte in der Dekomponierung der booleschen Operatoren in elementare boolesche Operatoren gesucht werden, von denen es gemäß (9.3) nur 16 gibt. Das könnte zwar die Produktion verbilligen, das Aufwandsproblem würde jedoch nicht gelöst, weil die elementaren Operatoren in entsprechender Stückzahl hergestellt werden müssten. Der Ausweg aus dem Dilemma liegt im Bau *programmierbarer* Rechner, die durch Ausführung von Programmen Funktionen *berechnen*. Darauf wird später eingegangen.

Es könnte der Eindruck entstehen, dass unsere Überlegungen zur Komponierbarkeit *nichtprogrammierbarer* Computer aus Kombinationsschaltungen keinerlei praktische Bedeutung haben angesichts des hohen technischen Aufwandes. Dem ist jedoch nicht so. Denn in vielen Konsumgütern (Autos, Fotoapparaten, Waschmaschinen usw.) sind kleine "Computer" eingebaut, die Funktionen "berechnen", deren Wertetafeln ausreichend klein und zudem bekannt sind, z.B. in Form von Vorgaben, bei welcher Beleuchtungsstärke (d.h. bei welcher Ausgangsspannung des entsprechenden Sensors) welche Belichtungszeit einzustellen ist, oder bei welcher Temperatur die Heizung einer Waschmaschine abzuschalten ist. Diese Vorgaben werden als boolesche Funktionen codiert und als mikroelektronische Kombinationsschaltung "*implementiert*" (realisiert), häufig auf einem einzigen winzigen Siliziumplättchen, *Chip* genannt (siehe dazu Kap.12.3.4). Wenn die zu realisierende Funktion sehr komplex ist, kann es allerdings ökonomischer sein, einen programmierbaren Rechner einzusetzen. 7

Neben der Steuerung einfacher technischer Prozesse gibt es noch ein anderes weites Feld, wo Kombinationsschaltungen zum Einsatz kommen können, es ist der Bereich der "Information", hier im umgangssprachlichen Sinne von *Auskunft* verstanden. Auskunftstabellen wie Adressbücher, Telefonbücher oder Wörterbücher können als Wertetafeln von Funktionen (Abbildungen) aufgefasst, binär codiert und als Kombinationsschaltung realisiert werden. Die Eindeutigkeit geht nicht verloren, wenn die Auskunft mehrere Varianten enthält, wenn z.B. ein Wort in mehrere Wörter einer anderen Sprache übersetzt wird, denn die vollständige Auskunft stellt nach Umcodierung *eine* Bitkette dar. Selbstverständlich lassen sich auf diese Weise auch mathematische Tabellen (Logarithmentafeln, trigonometrische Tafeln u.ä.m.), also die *Wertetafeln* mathematischer Funktionen "in Hardware gießen", d.h. als Schaltung realisieren. Wenn die Funktion nur selten benutzt wird oder ihre Wertetafel sehr umfangreich ist, kann es ökonomischer sein, einen programmierbaren Rechner einzusetzen.

Die weite Verbreitung "*nichtprogrammierbarer* Computer" ist nicht der eigentliche Grund dafür, dass wir uns so ausführlich mit booleschen Funktionen beschäf-

tigen. Dieser liegt vielmehr darin, dass auch die Hardware der konventionellen *programmierbaren* Computer aus booleschen Operatoren aufgebaut ist. Damit sind wir gut motiviert, um zwei zentrale Probleme anzugehen, die Darstellung von Binärwortfunktionen durch elementare boolesche Funktionen (Kap.9.3), und die technische Realisierung dieser Funktionen (Kap.10.1). Zuvor wollen wir der im ersten Augenblick für wenig sinnvoll angesehenen Idee nachgehen, aus Schwellenoperatoren Netze aufzubauen.

9.2.2 Künstliche Neuronen

Im vorangehenden Kapitel wurde festgestellt, dass es sinnlos zu sein scheint, aus Schwellenoperatoren Netze aufzubauen, da die inneren Operatoren (die Operatoren ohne externe Eingänge) durch boolesche Operatoren ersetzt werden können, ohne dass sich das Verhalten des Netzes ändert. Das bedeutet nicht, dass sie ersetzt werden *müssen*. Auch Schwellenoperatoren können die Rolle innerer Operatoren übernehmen und zwar in Form sogenannter *künstlicher Neuronen* (s.u.). Hier liegt sogar die eigentliche Bedeutung von Schwellenoperatoren.

Netze aus künstlichen Neuronen, sogenannte *neuronale Netze*, sind dem biologischen Vorbild der neuronalen Strukturen im Gehirn nachgebildet. Sie zeigen booleschen Netzen gegenüber neue Eigenschaften, die sich daraus ergeben, dass die Eingabeoperanden *aller* Bausteinoperatoren, nicht nur derjenigen mit externen Eingängen, reelle Werte annehmen dürfen. Das bedeutet physikalisch, dass die codierenden Zustandsparameter sich auf dem Wege von einem Operator zum nächsten kontinuierlich ändern dürfen, denn der empfangende Operator akzeptiert jeden reellen Wert bzw. jedes reelle Wertetupel und ordnet ihm einen binären Wert zu. Ein derartiges Netz mit mehreren (vereinten) Ausgängen liefert eine Bitkette.

8 Freilich können auch auf den (vereinten) *Eingang* eines Netzes aus Schwellenoperatoren Bitketten gegeben werden. Dann haben wir es mit einem Bitkettenoperator (Binärwortoperator) zu tun. Da aber die internen Operanden auf dem Weg durch das Netz ihren Wert kontinuierlich ändern, arbeitet das Netz im Gegensatz zu einem booleschen Netz nicht rein digital, sondern sowohl analog als auch digital.

Welche Funktion ein neuronales Netz berechnet, hängt - ebenso wie im Falle boolescher Netze - von der Verbindungstopologie, der Struktur des Operandenflussgraphen ab, außerdem aber von der Veränderung der Operanden auf den internen Übergabewegen. Diese spezifische, aus booleschen Netzen unbekannte Abhängigkeit führt zu einer Eigenschaft, deren Bedeutung nicht sofort zu erkennen und kaum zu überschätzen ist. Sie lässt sich anhand eines Gedankenexperiments unschwer verstehen.

Gegeben sei ein zirkelfreies Netz elektronisch arbeitender Schwellenoperatoren, das eine bestimmte Binärwortfunktion berechnet. Als codierende Zustandsparameter sollen die Ein- und Ausgangsspannungen der Schwellenoperatoren dienen, die durch elektrische Leitungen miteinander verbunden sind. In die Leitungen fügen wir variable Widerstände ein, die es gestatten, die Spannungsabfälle in den Leitungen

und damit die Operandenwerte kontinuierlich zu variieren. Es stellt sich die Frage, wie die Verhaltensweise des Netzes auf Veränderungen der Widerstände reagiert. Folgende globale Antwort ist leicht zu geben. Solange die Veränderungen nicht dazu führen, dass irgendeine Schwellenfunktion *g* (siehe (9.2)) den Schwellenwert *s* durchschreitet (sei es von oben her oder von unten her), ändert sich nichts. Sobald jedoch eine Schwelle infolge Widerstandsänderung über- oder unterschritten wird, ändert sich die Funktionsweise *sprunghaft*, d.h. die Binärwortfunktion, die das Netz realisiert, geht in eine andere über. Wir haben es mit einem *digitalen* Operator zu tun, dessen Eingabe-Ausgabeverhalten mittels *analoger* Parameter gesteuert werden kann, entweder durch Veränderung der Widerstände oder durch Veränderung der Schwellen.

Wenn das schon merkwürdig erscheint, so ist der Effekt, der damit zu erreichen ist, frappierend. Er besteht in der Fähigkeit zum Lernen. Dem Netz kann eine gewünschte Verhaltensweise (eine bestimmte Funktion) “beigebracht” werden und zwar dadurch, dass die Widerstände solange in kleinen Schritten verändert werden, bis das Netz die gewünschte Funktion realisiert. Das Verändern kann in mehr oder weniger zielstrebigem *Probieren* bestehen.

Diese kurzen Andeutungen sind vielleicht wenig überzeugend, und dem Nichteingeweihten mag es unerfindlich erscheinen, dass es möglich und überhaupt sinnvoll ist, die Verhaltensweise eines Operators mit *digitalem* Ein-Ausgabeverhalten durch *analoge* Steuerung zu verändern. Tatsächlich wäre wohl kaum irgendjemand durch reines Nachdenken auf diese Idee gekommen, wenn es die Natur nicht vorgemacht hätte. Die Idee zum Bau von lernfähigen Netzen aus Schwellenoperatoren ist nicht am grünen Tisch entstanden, sondern der Struktur des Gehirns abgeschaut. Der Schwellenoperator zusammen mit den variablen Widerständen in seinen Eingängen stellt ein stark vereinfachtes Modell des biologischen *Neurons* mit seinen *Synapsen* dar, des Bausteins der “grauen Materie” des Gehirns.

In diesem sehr groben Modell entsprechen die variablen Widerstände den *Synapsen* eines Neurons, über die es mit anderen Neuronen verkoppelt ist. Die Funktion der Widerstände kann ebenso wie die der Synapsen als *Wichtung* der Eingabeleitungen bzw. der vorgeschalteten Neuronen aufgefasst werden. Je niedriger der Widerstand (je höher der Leitwert) ist, umso größer ist die Wirkung (das Gewicht) des betreffenden Vorgängerneurons, umso stärker ist die Kopplung. Darum ist es sinnfälliger, die Wirkung eines Neurons auf ein anderes Neuron nicht durch einen *Widerstand* zu charakterisieren, sondern durch einen *Leitwert* oder abstrakter durch ein *Gewicht*.

Für Schwellenoperatoren mit Eingabegewichten hat sich die Bezeichnung **künstliches Neuron** eingebürgert. Durch Vernetzung künstlicher Neuronen entsteht ein künstliches neuronales Netz. In der technischen Literatur wird das Adjektiv *künstlich* meistens unterdrückt und von Neuronen und **neuronalen Netzen** gesprochen. Doch können dadurch falsche Vorstellungen entstehen. Von den unendlich komplizierteren natürlichen Neuronen ist lediglich ihr Schwellenverhalten übernommen und von den

Verbindungen zwischen Neuronen über Synapsen lediglich die Variabilität ihrer Leitfähigkeit. Wir haben es mit einer "gewaltigen" (gewalttätigen) Idealisierung zu tun. Sie ist aus wissenschaftlicher Sicht legal, ähnlich wie Newtons Idealisierung der Erde zu einem Massepunkt. Aus suggestiver Sicht wäre es jedoch günstiger, das Wort *künstlich* hinzuzusetzen oder die Idealisierung auf andere Weise zu unterstreichen, indem man künstliche neuronale Netze beispielsweise als *neuromorphe* Netze bezeichnet. Dennoch schließen wir uns dem eingebürgerten Sprachgebrauch an und sprechen von neuronalen Netzen, auch wenn *künstliche* neuronale Netze gemeint sind.

Neuronale Netze eröffnen einen dritten Weg zur Realisierung bestimmter Funktionen (neben dem Bau von Kombinationsschaltungen und dem Programmieren
11 programmierbarer Rechner): das **Lernen**. Wenn zur Realisierung einer Funktion die reine Hardwarelösung und auch die Formulierung eines Lösungsalgorithmus schwierig oder unmöglich ist, kann eventuell das *Erlernen* möglich sein. Das ist z.B. der Fall, wenn der Mensch dem Computer die Ausführung einer Operation übertragen will, die er selber zwar beherrscht, von der er aber nicht weiß, *wie* er dabei vorgeht, sodass er keine Operationsvorschrift angeben kann. Diese Sachlage ist charakteristisch für intuitive und assoziative Leistungen des Gehirns, z.B. für das *Erkennen*, etwa für das Erkennen eines Bekannten oder handschriftlicher Buchstaben. Von derartigen unverstandenen Operationen war in Kap.7.1 die Rede.

Damit wollen wir unseren Abstecher in die Neuroinformatik vorläufig abschließen, um ihn bei passender Gelegenheit fortzusetzen.

9.3* Ergänzung der formalen Methoden: Boolesche Algebra

Wir nehmen nun den Weg zum universellen Rechner wieder auf, den wir durch den Abstecher zu den neuronalen Netzen unterbrochen hatten. In Kap.8.5 hatten wie einen *universellen* Rechner dadurch gekennzeichnet, dass sich durch ihn jeder Kalkül realisieren lässt, d.h. dass sich die Operationen und Operanden jedes Kalküls in solche Operationen und Operanden abbilden lassen, die mit Hilfe des Computers komponierbar sind. Vorwegnehmend war erwähnt worden, dass die Realisierung eines Kalküls durch einen Computer in zwei Schritten erfolgt und zwar

1. in der Abbildung des zu realisierenden Kalküls in die boolesche Algebra und
2. in der Realisierung der booleschen Algebra als reale Operatorenhierarchie.

Wir waren zu dem Schluss gelangt, dass der erste Schritt infolge der Arbitrarität des Codierens und des Kalkülstransformationssatzes [8.39] stets möglich ist. Er hat natürlich nur dann Sinn, wenn der zweite Schritt bereits getan ist, wenn also die

boolesche Algebra realisiert ist, d.h. in Form eines Gerätes, Computer genannt, real existiert.

Der zweite Schritt bildet demnach das *technische* Fundament der Rechentechnik. Er basiert seinerseits auf der booleschen Algebra als einem der wichtigsten *theoretischen* Fundamente der Rechentechnik. Ihr wenden wir uns nun zu, werden allerdings keine vollständige Einführung in dieses Gebiet der Mathematik geben. Vielmehr verfolgen wir zwei spezielle Ziele. Es soll erstens nachgewiesen werden, dass sich für jede boolesche Funktion ein zirkelfreies Netz aus elementaren booleschen Operatoren angegeben lässt, das die Funktion realisiert, und zweitens, dass boolesche Funktionen rekursive Funktionen sind. Es wird eine Methode entwickelt, die es gestattet, die Wertetafel jeder beliebigen booleschen Funktion in einen booleschen Ausdruck aus elementaren booleschen Funktionen zu überführen.

Wir beginnen ganz formal ("semantikfrei") damit, die elementaren, also die ein- und zweistelligen booleschen Funktionen (Operationen) einzuführen, indem wir allen möglichen Wertetafeln *Namen* und *Symbole* zuordnen. Bild 9.1 gibt eine Übersicht über die Wertetafeln (Spalte 2) und die entsprechenden gängigen Namen (Spalte 4) und Symbole (Spalte 5). Dass in einigen Fällen mehrere Symbole verwendet werden, erschwert das Lesen einschlägiger Texte; es ist die Folge der historischen Entwicklung und der kontextlichen Bedingungen in unterschiedlichen Anwendungsgebieten. Dass die Wörter *Funktion* und *Operation* als Synonyme verwendet werden, ist legal, da nur von realisierbaren Funktionen und eindeutigen Operationen die Rede ist [8.16]. Die beiden Argumente sind nicht wie bisher mit x_1 und x_2, sondern der besseren Lesbarkeit halber mit *a* und *b* bezeichnet

In der Kopfzeile der Spalte 2 sind die vier möglichen Argumentwertepaare angegeben, die beiden Werte eines Paares jeweils untereinander. Sie stellt (um 90 Grad gedreht) die Argumentwertespalte der Funktionstafeln der 16 elementaren booleschen Funktionen dar. Darunter, in den 16 Zeilen der zweiten Spalte, folgen alle denkbaren vierstelligen Bitfolgen. Die Bitfolge einer Zeile stellt (um 90 Grad gedreht) die Funktionswertespalte der Funktionstafel einer der 16 elementaren booleschen Funktionen dar. Die Kopfzeile der Spalte 2 bildet also zusammen je einer der darunter folgenden Bitketten die Funktionstafel einer elementaren booleschen Funktion. In Spalte 1 ist der allgemeine Funktionsbezeichner *f* mit derjenigen Zahl indiziert, die sich ergibt, wenn die betreffende Bitfolge in Spalte 2 als Dualzahl interpretiert und in eine Dezimalzahl überführt wird.

Die Funktionen/Operationen sind durch Spalte 2 *formal definiert*. Mehr bedarf es nicht, um mit dem "Rechnen in boolescher Algebra" zu beginnen. Der *Kalkül* der booleschen Algebra ist durch die Spalten 2 und 5 im wesentlichen festgelegt, abgesehen von einigen notwendigen zusätzlichen Festlegungen wie Klammerungs- oder Vorrangregeln und die Gesamtheit der erlaubten Zeichen (Alphabet). In Spalte 6 sind boolesche Ausdrücke für die jeweiligen Funktionen angegeben. Wenn zwei verschiedene Ausdrücke dieselbe Funktion beschreiben, müssen sie einander gleich sein. In diesem Fall steht in Spalte 6 eine Gleichung.

1	2	3	4	5	6
a b	0101 0011	f=1; wenn:	Name der Funk- tion/ Operation	Symbol, Operationscode	mögliche Boolesche Ausdrücke $f_K=\bar{f}_{15-K}$
f_0	0000	nie	Konstante 0		$a \wedge \bar{a} = \overline{\bar{a} \vee a}$, dgl. für b
f_1	0001	sowohl a als auch b	Konjunktion	∧, &, AND	$a \wedge b = \overline{\bar{a} \vee \bar{b}}$
f_2	0010	nicht a, aber b			$\bar{a} \wedge b = \overline{a \vee \bar{b}}$
f_3	0011	b			$b = \bar{\bar{b}}$
f_4	0100	a, aber nicht b			$a \wedge \bar{b} = \overline{\bar{a} \vee b}$
f_5	0101	a			$a = \bar{\bar{a}}$
f_6	0110	entweder a oder b	Antivalenz, exklusives ODER	⊕, ⇎, XOR, EXOR	$(a \wedge \bar{b}) \vee (\bar{a} \wedge b)$
f_7	0111	a oder b	Disjunktion, inklusives ODER	∨, OR, ≥1	$a \vee b = \overline{\bar{a} \wedge \bar{b}}$
f_8	1000	weder a noch b	Pierce´scher Pfeil	↓, NOR	$\bar{a} \wedge \bar{b} = \overline{a \vee b}$
f_9	1001	a=b	Äquivalenz	⇔, EQ	$(\bar{a} \wedge \bar{b}) \vee (a \wedge b)$
f_{10}	1010	nicht a	Negation	¬, ~, ¯, NOT	$\bar{a}$
f_{11}	1011	nicht a, oder aber b	Implikation	$a \Rightarrow b$	$\bar{a} \vee b = \overline{a \wedge \bar{b}}$
f_{12}	1100	nicht b	wie f_{10}	wie f_{10}	$\bar{b}$
f_{13}	1101	nicht b, oder aber a		$b \Leftarrow a$	$a \vee \bar{b} = \overline{\bar{a} \wedge b}$
f_{14}	1110	nicht gleichzeitig a und b	Scheffer´scher Strich	| , NAND	$\overline{a \wedge b} = \bar{a} \vee \bar{b}$
f_{15}	1111	immer	Konstante 1		$a \vee \bar{a} = \overline{\bar{a} \wedge a}$

Bild 9.1 Elementare boolesche Funktionen

Um tatsächlich allein aufgrund der Spalten 2 und 5 zu rechnen und z.B. die Ausdrücke und Gleichungen (Formeln) der Spalte 6 abzuleiten, bedarf es einer ausgeprägten abstrakt-mathematischen Veranlagung, die den wenigsten Menschen in die Wiege gelegt ist. Die meisten müssen den mühsamen Weg von der Semantik, in der sie zu denken gewohnt sind, zur formalen Semantik des Kalküls gehen. Für diesen Weg soll Spalte 3 eine Hilfestellung geben. Bevor wir sie nutzen, wollen wir uns die Spalte 2 noch etwas genauer ansehen.

Die Tabelle enthält nicht nur die echt 2-stelligen, sondern auch die vier möglichen 1-stelligen und die beiden 0-stelligen Funktionen. Die 1-stelligen Funktionen hängen nur von einer einzigen Variablen ab; das sind zum einen die Funktionen f_5 und f_{10}, die nur von *a* abhängen (sie ändern ihren Wert *nicht*, wenn *b* seinen Wert ändert), und zum anderen die Funktionen f_3 und f_{12}, die nur von *b* abhängen. Die Funktionen f_3 und f_5 sind die Identitätsfunktion, z.B. $f(a)=a$. Die Funktionen f_{10} und f_{12} sind die Negation, z.B. $f(a)=\overline{a}$ (Überstreichung bedeutet Negation). Die Funktionen f_0 und f_{15} hängen weder von *a* noch von *b* ab, es sind die Konstanten 0 und 1. Die Konstante 0 ist die Negation der 1 und umgekehrt. Für die Bitfolgen in Spalte 2 bedeutet dies, dass jedem Funktionswert in der Wertetafel von f_0 der entsprechende negierte Funktionswert aus der Tafel von f_{15} stehen muss, wie es tatsächlich der Fall ist. Formal notieren wir $\overline{f}_0 = f_{15}$. Auf die gleiche Weise erkennt man durch bitweisen Vergleich der entsprechenden Bitfolgen, dass $f_1 = \overline{f}_{14}$ und allgemein $f_i = \overline{f}_{15-i}$ gilt.

Die booleschen Ausdrücke in Spalte 6 stellen Kompositoperatoren aus den drei Bausteinoperatoren Negation, Konjunktion und Disjunktion dar. Mit Hilfe dieser drei elementaren Operatoren lässt sich also *jede* elementare Operation beschreiben (komponieren). Wir wollen versuchen, aus den Wertetafeln die Ausdrücke in Spalte 6 abzulesen. Dazu drücken wir die Bedingung dafür, dass die jeweilige Funktion gleich 1 wird, verbal aus. Das Ergebnis ist in Spalte 3 in abgekürzter Form angegeben. Das Vorgehen soll anhand der Konjunktion demonstriert werden. Aus der Wertetafel folgt, dass f_1 genau dann gleich 1 ist, wenn sowohl *a* als auch *b* gleich 1 sind. Für die formale Notation in Spalte 6 ist das erste Symbol in Spalte 5 verwendet worden. Der Operationscode AND könnte ein “Aha!” auslösen: AND ist verständlich, es steht für “sowohl...als auch”.

Das Aha-Erlebnis verstärkt sich, wenn man die boolesche Algebra durch die 12
Aussagenlogik interpretiert, d.h. wenn man 0 und 1 als **falsch** und **wahr** und die Variablen als Platzhalter für Aussagen interpretiert, zum Beispiel: *a* für “Der Lichtschalter ist eingeschaltet”, *b* für “Die Glühbirne ist funktionstüchtig” und *c* für “Die Glühbirne leuchtet”; dann gilt $c = a \wedge b$. Das $\wedge$-Zeichen entspricht dem “und” in der verbalen Ausdrucksweise “Die Glühbirne leuchtet (genau dann), wenn sie funktionstüchtig ist *und* der Schalter eingeschaltet ist”.

Die angedeutete Interpretation der booleschen Algebra heißt **Aussagenkalkül**, **Aussagenlogik** oder **Aussagenalgebra**. In Kap.11.1 erfahren wir, dass die Aussagenalgebra der booleschen Algebra zeitlich vorausging. Der Weg von der Aussagenalgebra zur booleschen Algebra ist - wie die Entwicklung der Mathematik

überhaupt - der Weg der semantischen Objektivierung (vgl. Kap.5.4). Es ist der Weg der exakten Naturwissenschaften und der Weg jedes einzelnen Menschen, der sein Denken "mathematisiert". Wir gehen auf die Aussagenalgebra nicht weiter ein und wenden uns wieder der abstrakteren booleschen Algebra zu.

Das bisher zu Bild 9.1 Gesagte soll noch einmal anhand der Funktion f_1 zusammengefasst werden. Die Funktion ist durch ihre Wertetafel gemäß Spalte 2 definiert, sie heißt Konjunktion und der Ausdruck auf der linken Seite der Gleichung in Spalte 6 ist die in der booleschen Algebra übliche Notation. Auf der rechten Seite der Gleichung steht gemäß der Regel $f_i = \overline{f}_{15-i}$ der Ausdruck für $\overline{f}_{14}$. Der Operationscode von $\overline{f}_{14}$ ist NAND, das Negierte AND. Die Negation des negierten AND ist wieder das AND.

Die analoge Prozedur für die Disjunktion liefert die Gleichung in Spalte 6 der f_7-Zeile. Auf die gleiche Weise ergeben sich alle Gleichungen der Spalte 6. Sie stellen *Formeln* der booleschen Algebra dar. Die Formeln in den Spalten für f_1 und f_7 heißen **morgansche Regeln**. Mit dem Symbol "¬" für die Negation lauten sie:

$$a \vee b = \neg(\neg a \wedge \neg b) \tag{9.5}$$

$$a \wedge b = \neg(\neg a \vee \neg b) \tag{9.6a}$$

Formel (9.6a) soll noch einmal in anderer Notation angeschrieben werden, wobei Konjunktionszeichen unterdrückt und Negations- bzw. Disjunktionszeichen durch `NOT` bzw. `OR` ersetzt werden:

$$ab = \texttt{NOT}(\texttt{NOT}a \ \texttt{OR} \ \texttt{NOT}b). \tag{9.6b}$$

In dieser Form werden wir im Weiteren die Gleichung in Kap.12 anwenden.

Die Richtigkeit der Gleichungen (9.5) und (9.6) kann - wie die Richtigkeit *jeder* Gleichung der booleschen Algebra - durch Nachrechnen geprüft werden, indem man für jedes Argumentwertepaar den Wert der rechten und linken Seite der Gleichung bestimmt. Oft lassen sie sich mit externer Semantik belegen, sodass sie "verständlich" werden, die morgansche Regel z.B. folgendermaßen: "Ich gehe (dann und nur dann) spazieren, wenn ich gesund bin und die Sonne scheint", mit anderen Worten: "Ich gehe nicht spazieren, wenn ich nicht gesund bin oder wenn die Sonne nicht scheint".

Geht man alle Zeilen der Tabelle von Bild 9.1 durch, erkennt man, dass Spalte 3 tatsächlich die Bedingungen dafür enthält, dass die jeweiligen Funktionen den Wert 1 annehmen. Die Bedingungen sind jedoch nicht vollständig (sie sind notwendig, aber nicht unbedingt hinreichend), sodass sie keine *exakte* Beschreibung oder gar Definition darstellen. Sie können nach folgender Vorschrift zu einer exakten Beschreibung der jeweiligen Funktion vervollständigt werden: "Schreibe die *vollständige* Bedingung dafür auf, dass f_i gleich 1 wird, notiere die Bedingung als booleschen Ausdruck mit Hilfe der Negation, Konjunktion und Disjunktion und setze diesen Ausdruck gleich f_i." Das Wort *vollständig* bedeutet, dass *sämtliche* Argumentwerte-

paare anzugeben sind, für die f_i=1 gilt. Nur dann ist gesichert, dass für die übrigen Wertepaare f_i=0 gilt, dass der resultierende Ausdruck also tatsächlich die Wertetafel von f_i exakt beschreibt.

Das Vorgehen soll am Beispiel der Funktion f_6, der *Antivalenz*, demonstriert werden. Die Funktion f_6 ist gleich 1, wenn entweder a=1 und b=0 ist, wenn also $a \wedge \overline{b}$=1 gilt, oder wenn a=0 und b=1 ist, also $\overline{a} \wedge b$=1 gilt. Die vollständige Bedingung dafür, dass f_6 gleich 1 und nicht gleich 0 ist, kann also als Disjunktion zweier Konjunktionen notiert werden. Der resultierende Ausdruck stellt die Komponierung der Funktion f_6 mittels Negation, Konjunktion und Disjunktion dar:

$$f_6 = (a \wedge \overline{b}) \vee (\overline{a} \wedge b). \tag{9.7}$$

Dieser Ausdruck ist in Bild 9.1 für f_6 angegeben. Die Klammern können fortgelassen werden, da in der booleschen Algebra vereinbart wird, dass die Konjunktion den Vorrang vor der Disjunktion hat (in Analogie zum Vorrang der Multiplikation vor der Addition). Auf die gleiche Weise gelangt man zu dem Ausdruck für die Funktion f_8, die *Äquivalenz*. Die Äquivalenz ist gleich der negierten Antivalenz und umgekehrt (die entsprechenden Formeln sind in Spalte 6 nicht angegeben). Wie unschwer zu erkennen ist, erfüllen sämtliche Ausdrücke in Spalte 6 die oben genannte Vollständigkeitsbedingung. Spalte 6 demonstriert, dass sich jede elementare boolesche Funktion als Disjunktion, eventuell als Disjunktion von Konjunktionen darstellen lässt.

Die Methode ist offenbar nicht auf 2-stellige Funktionen beschränkt. Für jede beliebige boolesche Funktion lässt sich aus ihrer Wertetafel der entsprechende Ausdruck ablesen, d.h. eine Disjunktion aus Konjunktionen, wobei allerdings die 2-stellige Konjunktion bzw. Disjunktion eventuell auf eine *n*-stellige erweitert werden muss und zwar im Sinne des umgangssprachlichen “und..und..und..” bzw. “oder..oder..oder..”. In dieser Verallgemeinerung liefert eine *Konjunktion* den Wert 1, wenn sämtliche Argumentwerte gleich 1 sind, und eine *Disjunktion* liefert den Wert 1, wenn wenigstens ein Argumentwert gleich 1 ist. Offensichtlich lassen sich mehrstellige Konjunktionen in mehrere 2-stellige Konjunktionen dekomponieren. Das Entsprechende gilt für Disjunktionen. Damit ergibt sich der **Satz**: *Jede n-stellige boolesche Funktion lässt sich als Disjunktion n-stelliger Konjunktionen darstellen. Die Darstellung heißt* ***kanonische disjunktive Normalform*** (***KDNF***). Jede Konjunktion der KDNF enthält jede Argumentvariable, entweder negiert oder nichtnegiert.

Das Aufstellen einer KDNF nach obiger Vorschrift soll anhand zweier 3-stelliger Funktionen vorgeführt werden, deren Wertetafeln in Bild 9.2 in einer Tafel zusammengefasst sind. Wenn man sich an die Notationsweise der booleschen Algebra gewöhnt hat, ist es nicht schwer, zu verifizieren, dass es sich um die Wertetafeln der beiden Operationen handelt, die auszuführen sind, wenn zwei Dualzahlen stellenweise addiert werden sollen. In jedem Berechnungsschritt, d.h. für jede Stelle muss aus den jeweiligen Summandenbits x_1 und x_2 und dem Übertragsbit z aus der vorherigen Stelle das jeweilige Resultatbit y und das neue Übertragsbit z' berechnet

13 werden. Ein Operator, der beide Funktionen berechnet, ist ein Dualstellenaddierer; er wird auch **Volladdierer** genannt.

x_1	x_2	z	z'	y
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

Bild 9.2 Wertetafeln des Volladdierers

Die Arbeitsweise des Volladdierers soll anhand der letzten Zeile der Wertetafel demonstriert werden. Es sind drei Einsen zu addieren. Das ergibt die Dezimalzahl 3 und die Dualzahl 11, d.h. sowohl das Resultatbit als auch das Übertragsbit hat den Wert 1. Die Bezeichnung des Übertragsbits mit *z* soll an den inneren Zustand des Automaten erinnern. Er entspricht ihm genau, wenn die einzelnen *Stellen* einer Summe *iterativ* mit Hilfe des Volladdierers ("*Stellen*addierers") berechnet werden, denn dann muss der Übertrag über einen Schrittverzögerer (Taktverzögerer) auf den Eingang des Volladders zurückgegeben werden, also über einen Speicher, der einen Eingabeoperanden um einen Takt verzögert ausgibt. Die gesamte Schaltung ist ein sogenannter **sequenzieller Addierer**. In Kap.9.5 werden wir sehen, dass sich auch der Schrittverzögerer aus elementaren booleschen Operatoren aufbauen lässt (siehe Bild 9.7), sodass der sequenzielle Addierer als *zirkuläres boolesches Netz* realisierbar ist.

Aus den Wertetafeln von Bild 9.2 lassen sich die Ausdrücke (9.8) für die beiden Funktionen ablesen. Der besseren Lesbarkeit halber ist eine **verkürzte Notation** verwendet worden, bei der die Negation durch Überstreichung notiert wird und die Konjunktionssymbole sowie die Klammern fortgelassen werden. Außerdem wird die Disjunktion ab jetzt stets durch `OR` bezeichnet.

$$z' = x_1x_2\bar{z} \text{ OR } x_1x_2z \text{ OR } x_1\bar{x}_2z \text{ OR } \bar{x}_1x_2z$$

$$y = x_1\bar{x}_2\bar{z} \text{ OR } \bar{x}_1x_2\bar{z} \text{ OR } \bar{x}_1\bar{x}_2z \text{ OR } x_1x_2z \tag{9.8}$$

Kanonische disjunktive Normalformen sind häufig redundant und lassen sich vereinfachen. So kann z.B. die erste Disjunktion in dem Ausdruck für z' in (9.8) durch x_1x_2 ersetzt werden, denn in beiden Argumenten der Disjunktion treten x_1 und x_2 nichtnegiert auf, während *z* einmal negiert und das andere mal nichtnegiert auftritt, sodass der Wert der Disjunktion nicht davon abhängt, welchen Wert *z* annimmt. In einer reduzierten (nicht mehr kanonischen) disjunktiven Normalform (abgekürzt DNF) müssen die Konjunktionen nicht mehr sämtliche Variablen enthalten. Es sind viele Reduktionsalgorithmen entwickelt worden, die bei der technischen Realisierung boolescher Funktionen zum Einsatz kommen, u.a. um die Produktionskosten der Schaltkreise zu senken.

Die Bilder 9.3 und 9.4 zeigen die Überführung einer KDNF in eine Schaltung (sprich: in ein boolesches Netz) für den sehr einfachen Fall der Addition zweier *ein*stelliger Dualzahlen. Die Summe ist eine ein- oder zweistellige Dualzahl mit den

Stellenbits y_1 und y_2. Ein Operator, der diese Operation ausführt, heißt **Halbaddierer**. Für die Berechnung des *letzten* Bits der Summe zweier Dualzahlen reicht der Halbaddierer aus, weil es keinen Übertrag einer vorangehenden Stellenaddition gibt. Aus der Wertetafel von Bild 9.3 lassen sich die beiden unterhalb der Wertetafel angegebenen KNDF für y_1 und y_2 ablesen, die ihrerseits unmittelbar in die Schaltung von Bild 9.4a überführt werden können. Die drei AND-Operatoren entsprechen den drei Konjunktionen in den booleschen Ausdrücken für y_1 und y_2 (die Konjunktionszeichen sind unterdrückt). Die kleinen Kreise an den Eingängen der AND-Glieder stellen Negatoren dar.

x_1	x_2	y_1	y_2
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$y_1 = x_1 x_2$

$y_2 = \bar{x}_1 x_2$ OR $x_1 \bar{x}_2$

Bild 9.3 Wertetafel des Halbaddierers und die beiden KNDF zur Berechnung der Resultatbits.

Bild 9.4b zeigt die Schaltung des Halbaddierers mit einer veränderten, matrixförmigen Topologie der Verbindungsleitungen. Die Matrix (der Teil der Schaltung rechts von den AND-Gliedern) besteht aus zwei senkrechten und drei waagerechten Leitern[2]. Die

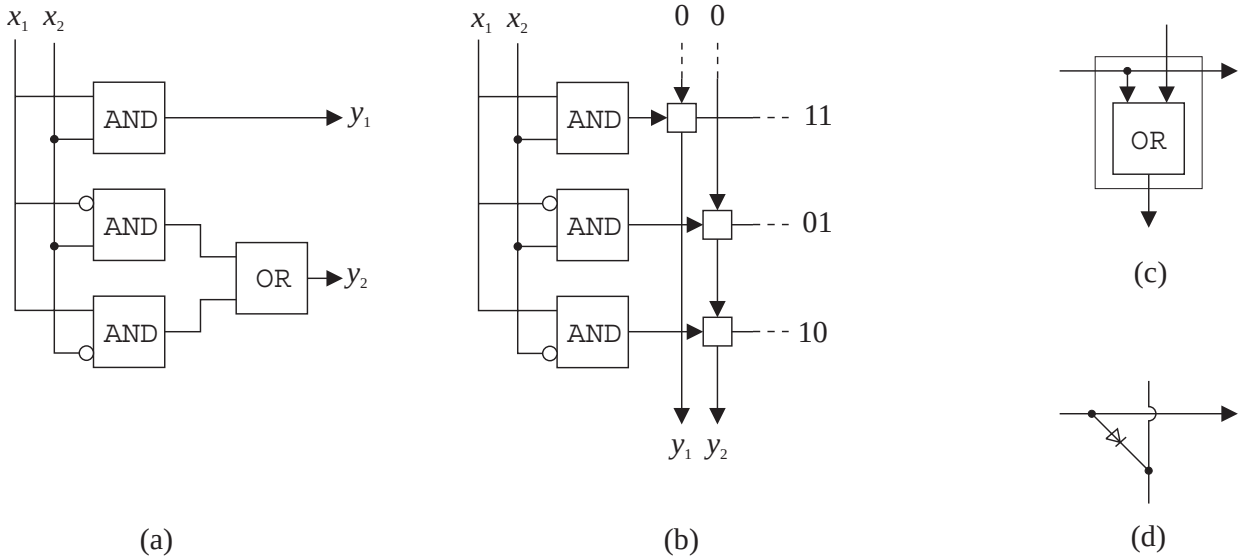


Bild 9.4 Schaltung des Halbaddierers, **(a)** als boolesches Netz, **(b)** als Leitermatrix; **(c)** - Dekomponierung eines Schnittpunktoperators; **(d)** Realisierung eines Schnittpunktoperators mittels Diode. Die kleinen Kreise stellen Negatoren dar.

2 Im Weiteren werden die Wörter *Leitung* und *Leiter* wie Synonyme verwendet, wenn von Leitungs- oder Leitermatrizen die Rede ist. Gemeint ist ein “Stück leitender Draht” bzw. eine entsprechend dotierte Strecke in einem Chip.

punktierten Verlängerungen sollen die Vorstellung hervorrufen, dass die Matrix aus einer größeren Matrix herausgeschnitten ist. An den oberen Enden der senkrechten Leiter liegt stets der Wert 0 an. Die kleinen Quadrate in den Schnittpunkten der senkrechten mit den waagerechten Leitern sind spezielle Flussknoten, die man
14 Schnittknoten nennen könnte, doch ziehen wir in diesem Fall die Bezeichnung **Schnittpunktoperator** vor. Ein Schnittpunktoperator besitzt zwei Eingänge und zwei Ausgänge und lässt sich in die Schaltung von Bild 9.4c dekomponieren. Daraus ist ersichtlich, dass ein Schnittpunktoperator den von links erhaltenen Wert nach rechts weitergibt und dass er nach unten immer dann eine 1 übergibt, wenn er über mindestens einen Eingang eine 1 empfängt. In Kap.12.1 [12.1] wird gezeigt, dass der Schnittpunktoperator durch die in Bild 9.4d gezeigte Diodenschaltung realisiert werden kann.

Überzeugen wir uns, dass es sich bei der Schaltung von Bild 9.4b tatsächlich um den Halbaddierer handelt. Man beachte, dass wir es mit einem zirkelfreien Operatorennetz zu tun haben, das keine Weichen, sondern nur starre Flussknoten enthält, also Gabeln und Vereinungen. Das trifft offensichtlich für alle DNF-Schaltungen zu.

Für jedes der Eingabepaare 01, 10, 11 belegt genau eines der drei AND-Glieder seine (waagerechte) Ausgabeleitung mit einer 1. Dadurch ist eine Zuordnung zwischen Eingabepaaren und waagerechten Leitern festgelegt. Die jeweiligen Bitpaare sind rechts neben den Leitern angegeben. Eine Schaltung, die Bitketten (Binärwör-
15 tern) Leitungen zuordnet (jedem Wort je eine Leitung), nennen wir **Wort-Leitung-Zuordner**. Sie wird in Kap.12 eine wichtige Rolle spielen. Aus der Funktionsweise der Schnittpunktoperatoren folgt, dass eine senkrechte Leitung eine Disjunktion realisiert, und zwar eine Disjunktion derjenigen Konjunktionen, mit deren Ausgabeleitungen sie über einen Schnittpunktoperator verbunden ist. Eine senkrechte Leitung stellt also zusammen mit den Konjunktionsgliedern die Schaltung einer DNF dar und zwar einer KDNF, denn die Negatoren sind so platziert, dass sämtliche möglichen Eingabetupel berücksichtigt werden. Das AND-Glied mit zwei negierten Eingängen erübrigt sich, denn die Eingabe 00 auf die Schaltung von Bild 9.4b bewirkt die Ausgabe 00 (Taktung mittels Toren vorausgesetzt).

Die Matrix enthält zwei senkrechte Leitungen, die Schaltung realisiert also zwei boolesche Funktionen in kanonischer disjunktiver Normalform. Durch Vergleich der Konjunktionen verifiziert man leicht, dass es dieselben Funktionen sind, die durch die Schaltung von Bild 9.4a berechnet werden. Die Darstellung als *Leitermatrix* entspricht der mikroelektronischen Realisierung mittels *Diodenmatrix* bzw. *Transistormatrix* (siehe Kap.12.1 [12.1]).

Zur Bestimmung der KDFN einer Funktion f hat man gemäß obiger Methode die vollständige Bedingung dafür aufzuschreiben, dass f gleich 1 ist. Am Rande sei angemerkt, dass man auch die Bedingung dafür aufschreiben kann, dass f gleich 0, also $\overline{f}$ gleich 1 ist. Wenn man den gesamten sich so ergebenden Ausdruck negiert und die morganschen Regeln mehrmals anwendet, gelangt man zu einem Ausdruck, der eine Konjunktion von Disjunktionen darstellt. Diese Darstellung einer booleschen

Funktion heißt **kanonische konjunktive Normalform (KKNF)**. Auch sie lässt sich in der Regel vereinfachen. Man wird die konjunktive Normalform der disjunktiven dann vorziehen, wenn die Funktion für relativ *wenige* Argumentwertetupel den Wert 0 annimmt. Im entgegengesetzten Fall wird man die KDNF vorziehen.

Damit ist unser erstes Ziel erreicht. Es wurde gezeigt, dass sich jede boolesche Funktion mittels Negation, Konjunktion und Disjunktion ausdrücken lässt. In diesem Sinne sagen wir, dass Negation, Konjunktion und Disjunktion zusammen einen **vollständigen Satz** boolescher Funktionen bilden. Mit diesem vollständigen Satz werden wir vorzugsweise bei der Komponierung der Hardware arbeiten, obwohl er sich reduzieren lässt. Man kann nämlich entweder auf die Konjunktion oder auf die Disjunktion verzichten, weil sich nach den morganschen Regeln die eine durch die andere (und durch die Negation) ausdrücken lässt. Es genügt sogar einzig und allein die NOR-Funktion, um sämtliche booleschen Funktionen komponieren zu können. Der Beweis lässt sich aus Bild 9.1 ablesen. Als Hinweis sei gesagt, dass z.B.

$$x \,\mathtt{NOR}\, x = \overline{x}$$

und

$$(x \,\mathtt{NOR}\, y) \,\mathtt{NOR}\, (x \,\mathtt{NOR}\, y) = x \,\mathtt{OR}\, y$$

gilt. Analoge Formeln lassen sich für die NAND-Funktion angeben.

Die Schaltungen dieses Kapitels sind zirkelfreie boolesche Netze aus elementaren booleschen Operatoren, also Kombinationsschaltungen. Doch sind sie von einem speziellen Typ, denn ihre Schaltungsstruktur entspricht der Syntax der disjunktiven Normalform. Darum nennen wir sie **disjunktive Kombinationsschaltungen** [3].

Die beschriebene konstruktive Methode zur Überführung der Wertetafel einer Binärwortfunktion in eine disjunktive Kombinationsschaltung ist offensichtlich auf beliebige Funktionstafeln anwendbar. Daraus folge der

Satz: Zu jeder Binärwortfunktion, die durch ihre Wertetafel festgelegt ist, lässt sich eine Kombinationsschaltung (ein zirkelfreies boolesches Netz) zur Berechnung der Funktion (zur Speicherung der Wertetafel) angeben. 16

Das ist ein Resultat von großer praktischer Bedeutung. Unbefriedigend ist allerdings die Beziehungslosigkeit zu den Überlegungen in Kap. 8.4.5, und es stellt sich die Frage, welche Relation zwischen booleschen und rekursiven Funktionen besteht. Die Antwort lautet: Boolesche Funktionen *sind* rekursive Funktionen. Davon kann man sich überzeugen, wenn man sich die Gültigkeit folgender Äquivalenzen klarmacht.

$$a \,\mathtt{AND}\, b \;\Leftrightarrow\; [a*b = 1] \tag{9.9a}$$

$$a \,\mathtt{OR}\, b \;\Leftrightarrow\; [a+b > 0] \tag{9.9b}$$

---

3 Diese Bezeichnung ist in der Literatur weniger üblich. Häufig wird der Begriff der kombinatorischen Schaltung ausschließlich für *disjunktive* kombinatorische Schaltungen verwendet.

$$\overline{a} \Leftrightarrow [a = 0] \tag{9.9c}$$

Das Prädikat auf der rechten Seite jedes der drei Äquivalenzen ist genau dann erfüllt (ist genau dann gleich 1), wenn der boolesche Ausdruck auf der linken Seite gleich 1 ist. Beispielsweise ist das Prädikat in (9.9b) genau dann erfüllt (also gleich 1), wenn *a* oder *b* oder beide gleich 1 sind, wenn also *a*OR*b* gleich 1 ist. Man beachte, dass *a* und *b* auf den linken Seiten der Gleichungen (9.9) boolesche Variablen, auf den rechten Seiten arithmetische Variablen darstellen. Es handelt sich jeweils um unterschiedliche Interpretationen ein und derselben abstrakten Funktion.

Die Prädikate auf den rechten Seiten der Gleichungen legen rekursive Funktionen fest, denn sie besitzen rekursive charakteristische Funktionen. Also sind auch die drei booleschen Funktionen auf den linken Seiten rekursive Funktionen. Da sich jede
17 boolesche Funktion in die disjunktive Normalform überführen lässt, folgt: ***Booleschen** Funktionen sind **rekursive** Funktionen*; und da sich jede Kombinationsschaltung aus booleschen Operatoren komponieren lässt, gilt der

18 **Satz:** Die durch Kombinationsschaltungen realisierbaren Funktionen sind rekursive Funktionen.

Damit beenden wir unseren Streifzug durch die boolesche Algebra. In Kap.11.1 werden wir ihn durch einen historischen Streifzug ergänzen und in großen Schritten den Weg durcheilen, der, in der Antike beginnend, schließlich zur booleschen Algebra und zu den mikroelektronischen Bausteinen eines Computers geführt hat.

9.4 Netzklassen

Die allgemeinen Überlegungen des Kapitels 9.1 und 9.2 über die Komponierung informationeller Operatoren haben unsere Aufmerksamkeit auf Operatorennetze unterschiedlicher Natur (boolesche und neuronale Netze) und unterschiedlicher Struktur (zirkelfreie und zirkuläre) gelenkt, wobei sich die Reihenfolge der betrachteten Netze mehr aus dem Gang unserer Gedanken und den sich dabei erhebenden Fragen und weniger aus irgendeiner Systematik heraus ergab. Dieser Mangel soll nun dadurch behoben werden, dass die bisher erwähnten Operatorennetze nach bestimmten Eigenschaften systematisiert werden. Dabei werden wir auch auf Netzklassen stoßen, die bisher noch keine Erwähnung gefunden haben. Wir beginnen die Systematisierung von Operatorennetzen mit der Frage, welche Folgen die statische Codierung auf die Komponierung von Netzen aus elementaren Operatoren hat, unabhängig davon, wie diese physikalisch funktionieren.

In Kap.9.2 [3] waren wir zu einer wichtigen Einsicht gelangt: Die elementaren Operatoren informationeller Systeme mit statischer Codierung besitzen kein Gedächtnis; sie können “sich nichts merken”. Der Ausgabeoperand (das Resultat) einer Operation liegt nur solange am Ausgang, wie der Eingabeoperand am Eingang liegt. Das gilt für elementare boolesche Operationen und für zirkelfreie Netze aus elementaren booleschen Operatoren, also auch für Kombinationsschaltungen.

Wir wollen nun die Möglichkeiten der Vernetzung elementarer Operatoren systematisch untersuchen. Dazu unterteilen wir alle denkbaren Netze in Klassen, wobei wir drei Eigenschaften als Klassifikationskriterien verwenden wollen. Wir werden Netze danach unterscheiden, ob sie aus booleschen Operatoren oder aus künstlichen Neuronen bestehen, ob sie eine zirkelfreie oder eine zirkuläre Struktur besitzen und ob sie Speicher enthalten oder nicht. Die sich ergebende Klassen und Unterklassen sind in Bild 9.5 aufgelistet. In Klammern sind Beispiele angefügt, von denen diejenigen später ausführlich behandelt werden, welche auf booleschen Netzen basieren.

- Boolesche Netze
  - boolesche zirkelfreie Netze (Kombinationsschaltung, ROM, ALU)
  - boolesche zirkuläre Netze
    - boolesche zirkuläre Netze ohne interne Speicher (Flipflop)
    - boolesche zirkuläre Netze mit internen Speichern (KR-Netz, Prozessor, Prozessorcomputer)
- Neuronale Netze
  - neuronale zirkelfreie Netze (Perzeptron, ADALINE)
  - neuronale zirkuläre Netze
    - neuronale zirkuläre Netze ohne interne Speicher (Hopfieldnetz)
    - neuronale zirkuläre Netze mit internen Speichern (Neurocomputer)

Bild 9.5 Netzklassen

Wir werden die Netzklassen der Reihe nach besprechen.

Zirkelfreie boolesche Netze

Offensichtlich lassen sich elementare boolesche Operatoren ohne weiteres (ohne Einsatz spezieller Speichereinheiten) zu zirkelfreien Netzen verbinden. Die Schaltung des Halbaddierers von Bild 9.4 ist hierfür ein Beispiel. Speichervorrichtungen zwischen den Bausteinoperatoren sind nicht erforderlich. Doch liegt die Summe nur solange am Ausgang der Schaltung, wie die Summanden an seinem Eingang liegen. Der Additionsschaltung muss also ein Speicher für die Summanden vorgeschaltet sein, d.h. eine Schaltung mit statisch stabilen Zuständen für die Codierung der Summanden. So kann der Zustand der Additionsschaltung stabil gehalten werden, sodass am Ausgang die Summe stabil codiert vorliegt.

Diese Schlussfolgerung gilt für beliebige zirkelfreie Netze aus elementaren Operatoren, d.h. für beliebige Kombinationsschaltungen und zirkelfreie neuronale Netze. *Durch das Vorschalten eines Speichers vor einen Operator ohne Gedächtnis wird*

die statische Stabilität des Zustandes des Operators gesichert und damit die statische Codierung des Ausgabeoperanden ermöglicht.

Die beiden Klassen der zirkelfreien booleschen und neuronalen Netze mit internen Speichern fehlen in Bild 9.5. Sie bedürfen keiner besonderen Betrachtung. Denn ihre Realisierung ist an keine zusätzlichen Bedingungen geknüpft im Vergleich zu den zirkelfreien Netzen *ohne* interne Speicher, und sie besitzen diesen gegenüber keine grundsätzlich neuen Eigenschaften, abgesehen von der Möglichkeit, dass sie auf ein Eingabewort nicht mit einem einzigen Ausgabewort, sondern mit einer Folge von Ausgabewörtern reagieren.

Zirkuläre boolesche Netze

Ganz anders liegen die Dinge, wenn ein Netz Rückkopplungen (zirkuläre Verbindungen) enthält. Dass Rückkopplungen *mit* internen Speichern möglich sind, ist offensichtlich. Weniger offensichtlich ist, dass auch Rückkopplungen *ohne* Speicher möglich und sinnvoll sein können. Betrachten wir den denkbar einfachsten Fall einer *booleschen Rückkopplungsschleife,* die Zurückführung des Ausganges eines Negationsgliedes auf seinen Eingang (Bild 9.6a). Die Rückkopplung lässt keinen statisch stabilen Zustand zu. Der Zustand "will" ständig hin und herspringen. (Ob sich die beiden Zustände tatsächlich abwechselnd voll ausbilden können, hängt von den Details des Übergangsprozesses ab). Die *Kopplungszirkularität* führt zu einer eigentümlichen "Widersprüchlichkeit", die an die Antinomien erinnert, die infolge *referenzieller Zirkularität* auftreten können, wie beispielsweise in der auf sich selbst bezogenen Aussage "Dieser Satz ist falsch" (vgl. Kap.6.2 [6.1]).

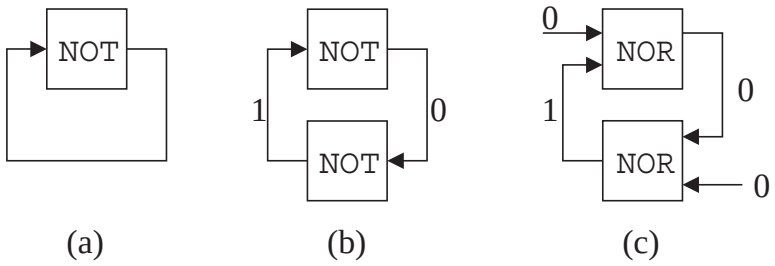


Bild 9.6 Einfache boolesche Rückkopplungsschleifen; **(a)** - widersprüchlicher Zirkel; **(b)** - Zirkel mit zwei stabilen Zuständen; **(c)** - Idee des Flipflop.

Betrachten wir daraufhin noch einmal die Prinzipschaltung des Automaten. Nehmen wir an, das Quadrat in Bild 8.5 stelle eine Kombinationsschaltung dar. Entfernt man nun den Speicher (den mit D bezeichneten Taktverzögerer), ergibt sich eine ähnlich undefinierte Situation wie im Falle der Schleife von Bild 9.6a. Der momentan berechnete Wert z' erscheint unmittelbar am Eingang der Kombinations-

schaltung, die sofort einen neuen z'-Wert berechnet, wodurch wiederum eine Neuberechnung initiiert wird und so weiter. Es liegt ein *widersprüchlicher* Zirkel vor.

Um die Widersprüchlichkeit des Zirkels in Bild 9.6a zu verhindern, muss in Analogie zu Bild 8.5 ein Speicher in die Schleife eingefügt werden. Es genügt jedoch nicht, den z'-Wert auf unbestimmte Zeit aufzubewahren, es muss der Zeitpunkt vorgebbar sein, zu dem der Wert weitergegeben wird. Zu diesem Zweck muss ein Tor in die Schleife eingebaut werden und zwar *vor* dem Speicher, da dieser den Eingabezustand des nachfolgenden Operators aufrechterhalten muss.

Diese Einsicht lässt sich verallgemeinern: *Sowohl boolesche Operatoren und als* 19
auch Kombinationsschaltungen lassen sich zu ***zirkulären booleschen Netzen*** *verbinden, wenn in jeden Zirkel (in jede Rückkopplungsschleife) ein* ***Speicher mit vorgeschaltetem Tor*** *eingebaut wird.* Der Speicher ist jedoch nicht in allen Fällen notwendig, wie sogleich gezeigt wird.

Werden alle Tore eines zirkulären booleschen Netzes *synchron* (gleichzeitig) gesteuert, ergibt sich eine *getaktete* Arbeitsweise des Netzes. In jedem Takt stellt sich ein neuer Zustand ein. Während einer Folge von Takten wird eine Folge (*Sequenz*) von Binärwörtern ausgegeben; das Netz zeigt ein *sequenzielles* Verhalten. Fasst man die einzelnen Speicher des Netzes gedanklich zu einem einzigen Speicher zusammen, gelangt man zu der Schaltung von Bild 8.5 und zum Begriff des abstrakten Automaten (vgl. Kap.8.2.3). Fasst man sie hardwaremäßig zu einem zentralen Speicher zusammen, gelangt man zur Prinzipschaltung des Prozessorrechners, wie schon hier vorgreifend auf Kap.13 bemerkt sei.

Wir betrachten nun einen autonomen Automat (oder auch einen Automat mit konstantem Eingabewert), sodass $z' = f(z)$ gilt. Es kann nun der Fall eintreten, dass der momentan berechnete neue Zustand mit dem alten identisch ist, $z' = z$, dass also gilt

$$f(z) = z. \tag{9.10}$$

Dann bleibt der Automat in diesem Zustand "stecken". Es bildet sich trotz Rückkopplung ein statisch stabiler Zustand aus, sodass der Speicher überflüssig wird. Einen solchen Zustand nennen wir **Eigenzustand**.

Der Begriff des Eigenzustandes ist in der Physik üblich, wo er im Zusammenhang mit *Differenzialgleichungen* verwendet wird. Der zugrunde liegende Tatbestand ist der gleiche: Die Stabilität eines Zustandes setzt die Gleichheit von Eingabe und Ausgabe des beschreibenden Operators voraus. In diesem Zusammenhang sei an den Analogrechner erinnert. Die Rückkopplungschleife in Bild 4.2b erzwingt die Gleichheit von Eingabe und Ausgabe, die in der zu berechnenden Differenzial*gleichung* formal notiert ist. In der Algorithmentheorie wird ein Argumentwert einer Funktion, für den (9.10) gilt, als **Fixpunkt** der betreffenden Funktion bezeichnet.

Zum Begriff des Eigenzustandes waren wir auf formalem Wege gelangt. Es stellt sich die Frage, wie sich Eigenzustände in booleschen Netzen realisieren lassen. Bild 9.6b zeigt das denkbar einfachste Beispiel, eine Schleife über *zwei* Negations-

glieder. Die Quelle der Instabilität der Schaltung von Bild 9.6a ist beseitigt, es können sich zwei verschiedene stabile Zustände einstellen (einer von ihnen ist in Bild 9.6b angegeben), die dadurch gekennzeichnet sind, dass auf den beiden Leitungen, welche die Negationsglieder verbinden, entgegengesetzte Werte liegen. Die Schleife in Bild 9.6b stellt also keinen widersprüchlichen Zirkel dar. Logisch entspricht er den beiden Sätzen in Kap.6.2 [6.2], von denen jeder behauptet, der andere sei falsch. Sie stellen *keine* Antinomie dar.

Man kann sich umfangreichere Strukturen ausdenken, in denen Eigenzustände möglich sind, z.B. Schleifen mit einer geraden Anzahl von Negatoren. Es sind auch verzweigte Strukturen (Netze) mit mehr als zwei Eigenzuständen denkbar. Das legt den Gedanken nahe, diese Zustände als *codierende* Zustände und ein solches Netz als Speicher zu verwenden. Damit ein zirkuläres boolesches Netz ohne interne Speicher als **Schreib-Lese-Speicher** dienen kann, muss die Möglichkeit bestehen, den Zustand, in dem sich das Netz gerade befindet, über einen externen Eingang in einen anderen stabilen Zustand zu überführen, d.h. den alten Gedächtnisinhalt mit einem neuen zu überschreiben.

Die Erfindung, die aus der Schleife in Bild 9.6b einen Schreib-Lese-Speicher macht, war für die Rechentechnik von fundamentaler Bedeutung, obwohl er natürlich nur ein einziges Bit speichern kann. Sie besteht in der Ersetzung der Negationsglieder durch NOR-Glieder, sodass sich eine Schleife mit zwei externen Eingängen ergibt (Bild 9.6c). Über sie lässt sich der Zustand verändern. Wenn z.B. an den linken (oberen) externen Eingang eine 1 angelegt wird, bewirkt das eine 0 auf der Ausgabeleitung des oberen NOR-Gliedes, unabhängig davon, welcher Wert vorher auf dieser Leitung gelegen hatte. Das Bemerkenswerte dabei ist, dass sich der Zustand aufrecht erhält, wenn die 1 am Eingang wieder verschwindet und an beiden Eingängen eine 0 liegt. Dieser Zustand ist in Bild 9.6c dargestellt. In entsprechender Weise lässt sich durch eine 1 am rechten (unteren) externen Eingang der umgekehrte stabile Zustand einstellen.

Um die Schleife in einen bestimmten Zustand “einflippen” zu lassen, genügt es also, einen kurzen 1-Impuls auf den entsprechenden Eingang zu geben. Wird anschließend auf den anderen Eingang ein 1-Impuls gegeben, “floppt” die Schleife in den anderen Zustand über. Von daher rührt wohl die lautmalerische Bezeichnung **Flipflop**. Der Flipflop von Bild 9.6c reagiert nur auf Einsen. In Kap.9.5 werden wir uns überlegen, wie sich der Fliflop zu einem Schreib-Lese-Speicher für ein Bit, zu einem **Ein-Bit-Speicher** erweitern lässt, und in Kap.13.2 werden wir aus Ein-Bit-Speichern Register und adressierbare elektronische Speicher komponieren. Unabhängig von der Komponierungsstufe werden wir von *booleschen* Speichern sprechen. *Speicher, deren statisch codierende Zustände Eigenzustände zirkulärer boolescher Netze sind, nennen wir* ***boolesche Speicher***.

Eigenzustände in booleschen Netzen kommen praktisch nur in Ein-Bit-Speichern als den Bausteinen von Registern und adressierbaren Speichern zur Anwendung. Demgegenüber haben Eigenzustände in neuronalen Netzen, auch in sehr großen

neuronalen Netzen, breite technische Anwendung gefunden (s.u.). Die eingeschränkte Nutzung von Eigenzuständen in booleschen Netzen ist verständlich, denn ein solches Netz kann stets durch eine Kombinationsschaltung mit vorgeschaltetem Speicher ersetzt werden.

Aus der Analyse der verschiedenen Möglichkeiten, Kompositoperatoren (Schaltungen) aus booleschen Operatoren zu komponieren, hat gezeigt, dass die Komponierung sowohl zirkelfreier als auch zirkulärer Netze zur Paarung von Speicher und Kombinationsschaltung führt. Die Notwendigkeit, einer Kombinationsschaltung einen Speicher vorzuschalten, ist die Folge der Forderung, dass statische Codierung möglich sein soll. Um diese Einsicht kompakter und aussagekräftiger ausdrücken zu können, vereinbaren wir: *Eine Kombinationsschaltung mit vorgeschaltetem Speicher wird als einfachster Vertreter von Netzen aus Kombinationsschaltungen und Speichern aufgefasst; elementare boolesche Operatoren werden als einfachste Vertreter von Kombinationsschaltungen aufgefasst.* Wenn wir noch berücksichtigen, dass es infolge der Arbitrarität der Codierung stets möglich ist, binär-statische Codierung in irgendeine andere statische Codierung umzucodieren, ohne dadurch die Allgemeingültigkeit der vorangehenden Überlegungen einzuschränken[4] gelangen wir zu folgendem

Satz: Aus booleschen Operatoren komponierte informationelle Operatoren mit statischer Codierung sind notwendigerweise Netze aus Kombinationsschaltungen und booleschen Speichern (oder in solche überführbar), wobei in jeder Verbindung (in jedem Operandenweg) zwischen zwei Kombinationsschaltungen ein Speicher mit Eingangstor liegt. 21

Zirkelfreie neuronale Netze

Offensichtlich können Schwellenoperatoren - ebenso wie boolesche Operatoren - ohne Schwierigkeiten zu zirkelfreien Netzen verbunden werden. Wenn mehrere Bausteinoperatoren (Neuronen) eines neuronalen Netzes externe Ausgänge besitzen, liefert das Netz an dem gemeinsamen ("vereinten") Ausgang ein Binärwort. Legt man an den (vereinten) Eingang ebenfalls ein Binärwort, wird dieses transformiert. Das Netz realisiert eine bestimmte Binärwortfunktion.

Betrachtet man einen derartigen Kompositoperator als schwarzen Kasten, so ist er in seinem Verhalten von einer Kombinationsschaltung nicht zu unterscheiden, solange die Gewichte (Synapsenleitwerte) konstant bleiben. Das bedeutet, dass zu jedem zirkelfreien neuronalen Netz mit definierten Gewichten eine Kombinationsschaltung mit dem gleichen Eingabe-Ausgabeverhalten angeben werden kann, sodass der gegen Ende des Kapitels 9.3 [9.18] für Kombinationsschaltungen formulierte Satz auch für für zirkelfreie neuronale Netze gilt:

---

4 In Kap,12.1 wird gezeigt, dass eine solche Umcodierung mittels zirkelfreier boolescher Netze stets möglich ist.

Satz: Die durch zirkelfreie neuronale Netze realisierbaren Funktionen sind rekursive Funktionen.

Alles, was in Verbindung mit Kombinationsschaltungen hinsichtlich der statischen Codierung gesagt wurde, lässt sich sinngemäß auf zirkelfreie neuronale Netze übertragen. Künstliche Neuronen und zirkelfreie Netze aus künstlichen Neuronen besitzen - ebenso wie elementare boolesche Operatoren und zirkelfreie boolesche Netze - keine inneren stabilen Zustände, also kein Gedächtnis. Wenn sie zur Informationsverarbeitung mit statischer Codierung verwendet werden sollen, müssen ihnen - ebenso wie Kombinationsschaltungen - Speicher vorgeschaltet werden.

Der Umstand, dass neuronale Netze, deren interne Operanden reelle Größen sind, mit binär-statischer Codierung arbeiten können, mag im ersten Augenblick Unverständnis hervorrufen. Er wird verständlich, wenn man sich folgendes klarmacht. Voraussetzung für statische Codierung ist, wie gesagt, das Vorschalten eines Speichers. Ist diese Voraussetzung erfüllt, so führt der statische Zustand des Speichers (d.h. der Speicherinhalt) zur Ausbildung eines statischen Zustandes im nachgeschalteten neuronalen Netz. Dabei entsprechen den booleschen Werten 0 und 1 unterschiedliche reelle Werte der codierenden Zustandsparameter, etwa den Spannungswerten im Netz. Das widerspricht nicht dem Prinzip der statischen Codierung. Die Änderung der codierenden Parameterwerte führt dazu, dass an unterschiedlichen Punkten des Netzes den booleschen Werten 0 und 1 unterschiedliche reelle Werte bzw. Wertebereiche entsprechen.

Ein Blick in die Literatur zeigt, dass vorzugsweise neuronale Netze mit sehr vielen Eingängen untersucht werden und dass diese der Anschaulichkeit halber zweidimensional (in einer Ebene) angeordnet werden, sodass jedem *Eingangsneuron* ein Punkt der Ebene, m.a.W. ein Pixel der *Bild*fläche entspricht. Wenn nun die Pixel der
22 angeregten Neuronen *schwarz* gefärbt werden und die übrigen Pixel *weiß* (untergrundfarben) bleiben, ergibt sich ein gerastertes Schwarz-weiß-Muster, das sog. *Eingabemuster* des Netzes. Häufig werden auch Netze mit sehr vielen *Ausgangsneuronen* untersucht, die ebenfalls zu einem Muster zusammengesetzt werden können. Eventuell kann der Betrachter die Muster interpretieren, also in ihnen irgendwelche Objekte *erkennt*. Erkennt er z.B. in einem Ausgabemuster den Buchstaben A, so ist dies kein Zeichenrealem, denn es trägt keine Symbolfunktion. Das Netz gibt ein Muster aus, das nur "wie ein A aussieht". Darum sprechen wir vom *subsymbolischen* Charakter der Muster [1.4], im Gegensatz zum *symbolischen* Charakter der Ein- und Ausgaben boolescher Netze bzw. traditioneller Rechner. Wenn ein Computer ein A ausgibt, sieht das nicht nur wie ein A aus, sondern es *ist* ein A (das Zeichenrealem A).

Wir wollen uns überlegen, welchen Nutzen man aus dem Umstand ziehen kann, dass die Ausgabeoperanden eines neuronalen Netzes - ebenso wie im Falle der Kombinationsschaltungen - Binärwörter sind, die Eingabeoperanden aber Tupel reeller Zahlen (Vektoren). Dazu fragen wir uns, was eintreten wird, wenn diese Werte stetig verändert werden. Nach Kap.9.2.2 [9] ist klar, dass bei sehr geringen Verän-

derungen des Eingabevektors keine Veränderung des Ausgabewortes zu erwarten ist. Diese tritt erst bei ausreichender Änderung einer oder mehrerer Komponenten des Eingabevektors ein.

Zur Veranschaulichung dieses Verhaltens gehen wir von 2-dimensionalen Vektoren (Eingabepaaren) aus und ordnen jedem Vektor einen Punkt in einem ebenen Koordinatensystem zu. Markiert man nun alle Punkte, die zu ein und demselben Ausgabewort gehören, mit einer bestimmten Farbe, ergibt sich eine Art Landkarte, wobei jedes "Land" ein bestimmtes Ausgabewort besitzt, das als *Name* des jeweiligen Landes dienen kann. (Ein Land kann aus mehreren nicht zusammenhängenden Gebieten bestehen.)

Diese Eigenschaft bleibt auch für 3- und mehrdimensionale Inputvektoren erhalten. Abstrahiert man von dem geometrischen Bild und fasst die Werte der Komponenten des Eingabevektors als Werte *irgendwelcher Merkmale* auf, dann wird die Verhaltensweise des Netzes zu dem, was wir in Kap.5.5 [5.15] mit *Klassifizieren* bezeichnet haben. Das Netz *bildet Klassen*, es funktioniert als Klassifikator; es fasst 23
Mengen von Vektoren zu Klassen zusammen und "bezeichnet" die Klassen mit Ausgabewörtern. Bei Änderung der Gewichte ändern sich die Klassen. Ein neuronales Netz, das alle Eingaben in zwei Klassen einteilt (das eine sogenannte **Dichotomie** ausführt), benötigt nur ein einziges Ausgabeneuron. Das Ausgabebit *codiert* die Klasse.

Die Tatsche, dass neuronale Netze klassifizieren können, sollte eigentlich nicht überraschen, denn ihr Aufbau ist, wie gesagt, dem Gehirn abgelauscht, dessen wohl erstaunlichste und vielleicht auch wichtigste Eigenschaft die Fähigkeit zum Klassifizieren ist. Es ist jedoch nicht ohne Weiteres klar, wie sich ein neuronales Netz mit einer *vorgegebenen* Klassifizierungsfunktion konstruieren lässt, oder - im Hinblick auf das Gehirn - wie ein neuronales Netz eine *nützliche* Klassifizierungsfunktion durch synaptische Veränderungen *erlernen* kann (siehe dazu [21.5]). Es sind viele einschlägige Methoden entwickelt und in der Literatur beschrieben worden.[5]

Zirkuläre neuronale Netze

Ebenso wie in booleschen Netzen und in abstrakten Automaten können auch in zirkulären neuronalen Netzen Eigenzustände auftreten. Es sind also stabile Zustände möglich. Solche Netze besitzen ein Gedächtnis. Sie können als Speicher verwendet werden. Neuronale Netze mit Eigenzuständen (internen stabilen Zuständen) spielen in der Theorie und Praxis neuronaler Netze eine hervorragende Rolle. Demgegenüber ist die Bedeutung zirkulärer neuronaler Netze mit *internen* Speichern, die selbstverständlich auch möglich sind, fast zu vernachlässigen. Die Situation ist also umgekehrt

5 Siehe z.B. [Dorffner 91], [Grauel 92], [Nauck 96], [Churchland 97], [Werner 95], [Stöcker 95], [Keller 00]. In Kap.21.3.2 [21.5], wird das Lernen mittels Stichprobe skizziert.

wie im Falle zirkulärer *boolescher* Netze. Sequenziell arbeitende neuronale Netze sind gegenwärtig die Ausnahme.

Das ist merkwürdig, denn die Introspektion in das eigene Denken lässt kaum einen Zweifel daran, dass das *Nachdenken* (das logischen Denken, das Ableiten, das Kopfrechnen) ein sequenzieller Prozess ist. Will man ihn mit Hilfe neuronaler Netze nachbilden, muss man offenbar auf sequenziell arbeitende Netze zurückgreifen, also auf Netze mit internen Speichern. Als Speicher können boolesche oder neuronale Speicher verwendet werden. *Speicher, deren statisch codierende Zustände Eigenzustände zirkulärer neuronaler Netze sind, nennen wir* ***neuronale Speicher***.

Alles, was über zirkuläre boolesche Netze mit internen Speichern gesagt worden ist, kann sinngemäß auf zirkuläre neuronale Netze übertragen werden, und der Satz, mit dem die Behandlung der zirkulären booleschen Netze abgeschlossen wurde, kann verallgemeinert werden. Wir fassen die Ergebnisse dieses Kapitels in dem folgenden **Strukturgesetz informationeller Operatoren mit binär-statischer Codierung** zusammen (der Begriff des zirkelfreien Netzes soll wieder den elementaren Operator als Sonderfall einschließen):

Ein ***Informationeller Operator*** *mit binär-statischer Codierung ist ein Kompositoperator aus elementaren booleschen Operatoren oder künstlichen Neuronen* und zwar

- entweder ein zirkelfreies Netz mit vorgeschaltetem Speicher mit Eingangstor
- oder ein zirkuläres Netz aus zirkelfreien Netzen mit vorgeschalteten (internen) Speichern mit Eingangstoren,
- oder er ist durch einen der beiden Typen ersetzbar, ohne dass die Zuordnungen, die er durchführen kann, verändert werden.

Eine Operationsausführung eines Operators des ersten Typs besteht aus einem einzigen Übergangsprozess, seine kausaldiskrete Beschreibung ist nicht dekomponierbar. Eine Operationsausführung eines Operators des zweiten Typs besteht aus einer (nicht unbedingt vollgeordneten) Folge von Übergangsprozessen, sie ist ein *sequenzieller* Prozess. Ein informationelles System, das sequenziell arbeitet und mindestens ein neuronales Netz enthält, nennen wir **Neurocomputer**. (Zuweilen wird die Forderung der sequenziellen Arbeitsweise nicht gestellt.)

Dass zirkuläre neuronale Netze mit internen Speichern bisher kaum Beachtung gefunden haben, liegt u.a. wohl daran, dass die Informatiker zu sehr von den Möglichkeiten fasziniert waren und sind, die sich aus der relativ großen Anzahl möglicher Eigenzustände großer neuronaler Netze ergeben. Das Interesse, das zirkuläre neuronale Netze mit vielen Eigenzuständen hervorriefen, ist verständlich. Wenn nämlich die Eigenzustände zu *codierenden* Zuständen erklärt werden, wird das Netz als Ganzes zu einem Speicher, obwohl es keine internen Speicher besitzt. Es fragt sich, wie man diese Möglichkeit praktisch nutzen kann und ob die Natur diese Möglichkeit nutzt. Wir werden dieser Frage nicht nachgehen, da sie über den Rahmen des Buches hinausgeht, doch wir erinnern uns, dass wir auf das gleiche Problem bereits hinsichtlich boolescher Netze gestoßen waren und zwar in Verbindung mit

Bild 9.6c. Die Frage war, wie sich das dort dargestellte boolesche Netz als Schreib-Lese-Speicher nutzen lässt. Diese Frage soll im folgenden Kapitel beantwortet werden.

Zum Abschluss dieses Kapitels ist eine Bemerkung darüber am Platze, was mit dem vielleicht etwas unmotiviert erscheinenden Abstecher in das Gebiet der neuronalen Netze bezweckt werden sollte. Der Abstecher war notwendig, um die *Vollständigkeitsforderung* zu erfüllen, die wir uns in Kap.9.2 hinsichtlich des Komponierens informationeller Systeme auferlegt hatten, denn als elementare Operatoren kommen, wie wir gesehen haben, nicht nur elementare boolesche Operatoren, sondern auch künstliche Neuronen in Frage. Ferner sollte die Wegegabel aufgezeigt werden, wo der Weg, der zum Neurocomputer führt, von unserem Weg, der zum Prozessorcomputer führt, abzweigt. Auf dem einen Weg wird der Schwellenoperator als interner Bausteinoperator verwendet, auf dem anderen Wege nicht. Im Zusammenhang mit *Prozessorcomputern* kommen Schwellenelemente ausschließlich in Analog-digital-Konvertern zur Anwendung, die dem Computer vorgeschaltet sind, während sie in neuronalen Netzen und im *Neurocomputer* auch als interne Bausteinoperatoren zur Anwendung kommen.

9.5 Ein-Bit-Speicher

Wir greifen noch einmal das Problem des Ein-Bit-Speichers auf, das sich in Kap.9.4 bei der Behandlung zirkelfreier boolescher Netze ergeben hatte, das wir dort aber nicht vollständig gelöst haben. Wie wir inzwischen erkannt haben, müssen Speicher mit Eingangstoren ausgerüstet sein. Außerdem wissen wir, dass es ausreicht, nur kurzzeitig eine 1 an den einen oder anderen Eingang eines Flipflop zu legen, um zu erreichen, dass dieser sich danach in dem einen oder in dem anderen Eigenzustand befindet.

Auf dieser Grundlage lässt sich ein Ein-Bit-Speicher mit vorgeschaltetem Tor aufbauen. Seine Schaltung ist in Bild 9.7 gezeigt. Sie ist vielleicht zu kompliziert, um sofort verstanden werden zu können. Es ist für das Verständnis des Weiteren nicht unbedingt erforderlich, dass der Leser die Wirkungsweise der Schaltung im einzelnen durchspielt. Das gleiche gilt auch für die Schaltungen in den folgenden Kapiteln, die nicht immer ganz einfach sind. Doch wer die Mühe des Durchspielens auf sich nimmt, der wird belohnt. Denn es ist schon amüsant oder sogar spannend, nachzuempfinden, wie aus elementaren logischen Operatoren ein Ein-Bit-Speicher, ein adressierbarer Speicher, ein Prozessor oder ein Computer wird.

Tatsächlich ist der Nachvollzug der Wirkungsweise der Schaltungen, die in diesem Buch behandelt werden, im Grunde auch für den Unvorbereiteten nicht allzu schwierig. Er sollte sich nicht vom ersten Anblick abschrecken lassen. Ohne Frage handelt es sich um recht komplizierte Produkte menschlicher Erfindungsgabe, und die Entwicklung effektiver und zuverlässiger Schaltungen, beispielsweise der Schal-

tung eines Mikroprozessors oder eines elektronischen Speichers, verlangte und verlangt ausdauerndes, konzentriertes und von Intuition und Phantasie beflügeltes Nachdenken zahlloser Erfinder, Ingenieure, Informatiker, Mathematiker und Physiker.

Ein Vergleich von Bild 9.7 mit Bild 9.6c lässt erkennen, dass die dort dargestellte Schaltung um ein Negationsglied und zwei AND-Glieder erweitert und dass eines der beiden NOR-Glieder um 180° gedreht ist, sodass die Ausgänge beider NOR-Glieder in die gleiche Richtung zeigen.

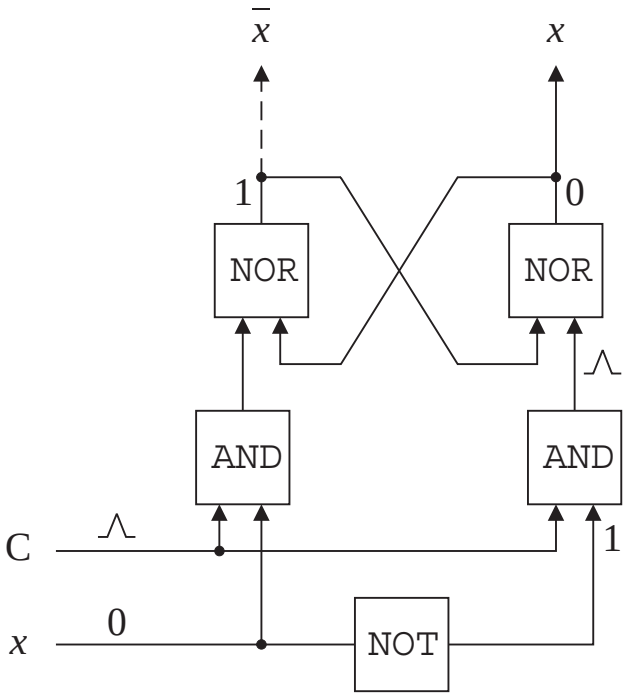


Bild 9.7 Boolesches Netz des Ein-Bit-Speichers.

Die Schaltung besitzt zwei Eingänge, einen *Arbeitseingang*, an dem das zu speichernde Bit (mit x bezeichnet) als statisch stabiler Eingabezustand anliegt, und einen *Steuereingang*, mit C bezeichnet, an dem normalerweise eine 0 liegt, an den aber kurzzeitig eine 1 angelegt werden kann. Diese "kurze 1" nennen wir Steuerimpuls oder **Steuersignal**. Die AND-Glieder fungieren als Tore. Bei $x = 0$, d.h. wenn eine Null gespeichert werden soll, wie in Bild 9.7 dargestellt, passiert das Steuersignal das rechte AND-Glied, und der Flipflop geht in den angegebenen Eigenzustand über bzw. verharrt in ihm. Am Ausgang des rechten NOR-Gliedes kann der Speicherinhalt und am Ausgang des linken NOR-Gliedes der negierte Speicherinhalt ausgelesen werden. Bei $x = 1$ wird bei Erscheinen des Steuersignals der andere Eigenzustand ausgelöst.

Die Schaltung von Bild 9.7 besitzt also diejenigen Eigenschaften, die von einem Ein-Bit-Speicher verlangt werden. Es gibt auch andere Schaltungsvarianten. Ersetzt man z.B. in Bild 9.6c die NOR-Glieder durch NAND-Glieder, bleibt die Flipflop-Eigenschaft erhalten. Eine verbreitete Realisierung des Ein-Bit-Speichers ist der **getriggerte NAND-Flipflop**. Er besteht aus vier NAND-Gliedern (vgl. z.B. [Matschke 86], S.97). Das Wort "*getriggert*" bedeutet, dass der Flipflop ein Eingangstor besitzt. Getriggerte Flipflops werden häufig als **Trigger** bezeichnet.

Über die elektronische Realisierung der booleschen Operatoren, des Flipflop und des Ein-Bit-Speichers ist bisher nichts gesagt worden. Sie wird im folgenden Kapitel behandelt. Aber auch ohne die Realisierungsmöglichkeiten zu kennen, ist es einleuchtend, dass sich durch Zusammenschalten elektronischer Ein-Bit-Speicher elektronisch arbeitende Computerspeicher bauen lassen. Es muss darauf hingewiesen werden, dass das Wort *elektronisch* in diesem Zusammenhang nicht ganz einheitlich

verwendet wird. Der Eindeutigkeit halber vereinbaren wir: Eine technische *Anordnung (eine Schaltung, ein Gerät) heißt* ***rein elektronisch****, wenn ihre Wirkungsweise ausschließlich auf der Bewegung und/oder Positionierung von Elektronen oder Ionen beruht.* 24

In der technischen Umgangssprache wird das Wort *elektronisch* in einem anderen, allgemeineren Sinne verwendet. Beispielsweise spricht man von *elektronischer Datenverarbeitung* auch dann, wenn sie magnetische oder optische Speicherverfahren einsetzt. Da im Weiteren ausschließlich auf rein elektronische Speicher eingegangen wird, sollen an dieser Stelle einige kurze Bemerkungen über magnetische und optische Speicher eingeschoben werden.

9.6 Einschub: Magnetische und optische Speicherverfahren

Große Bedeutung hatten in der Vergangenheit die sog. *Ferritkernspeicher*. Sie bestanden aus vielen kleinen Ringen aus Ferrit, *Ferritkerne* genannt. Ferrit ist ein ferromagnetisches Material. Die beiden Zustände maximaler Magnetisierung eines Ringes (rechtsherum bzw. linksherum) wurden zur Codierung eines Bit genutzt. Das Schreiben und Lesen erfolgte mittels Stromstößen durch Schreib- und Leseleitungen, auf welche die Ferritkerne aufgefädelt waren. Jeder Kern stellte einen Ein-Bit-Speicher dar

Während der Ferritkernspeicher praktisch vollständig durch rein elektronische Speicher verdrängt worden ist, verliert eine andere Variante der magnetischen Speicherung keineswegs ihre Bedeutung, die *Speicherung auf bewegtem Träger*. Die Methode verwendet zur Speicherung einzelner Bits winzige Oberflächenelemente einer dünnen Schicht aus ferromagnetischem Material, das auf Bänder oder Scheiben (Platten) aufgebracht ist. Die Träger werden in schnelle Bewegung versetzt (Rotation beim Magnetplatten- oder Diskettenspeicher, bzw. Umspulen beim Magnetband- oder Kassettenspeicher). Mittels eines Schreib-Lese-Kopfes lässt sich die Magnetisierung eines Elementes durch Induktion verändern (ein Bit einschreiben). Ebenfalls durch Induktion lässt sich die Magnetisierung eines am Kopf sich vorbeibewegenden Oberflächenelements feststellen, d.h. das gespeicherte Bit kann gelesen werden. Hierauf beruht die Arbeitsweise von *Kassetten-* und *Diskettenspeichern*. Ein Vorläufer der Magnetplatte war die *Magnettrommel*; die ferromagnetische Schicht bildete die Oberfläche eines rotierenden Zylinders.

Kassetten- und Diskettenspeicher haben den elektronischen und Ferritkernspeichern gegenüber den Nachteil relativ großer Zugriffszeiten zu einem bestimmten Speicherplatz, da Kopf und Datenträger (Band bzw. Diskette) zueinander positioniert werden müssen, was mechanisch erfolgt. Doch haben sie den Vorteil großer Speicherkapazität bei relativ niedrigen Kosten. Darum werden sie auch als *Massenspeicher* bezeichnet. Außerdem gestatten sie die räumliche Trennung von Datenträger und Laufwerk. Das Laufwerk ist Bestandteil des Computers und bewerkstelligt den

Antrieb und die Positionierung des Datenträgers. Mittels Kassetten oder Disketten können also große Datenmengen zwischen Rechnern über große Entfernungen transportiert werden. Das Laufwerk spielt dann die Rolle eines Ein-Ausgabegerätes. Diese Art des Datentransportes verliert durch die weltweite Computervernetzung zunehmend an Bedeutung.

Seit einigen Jahren setzt sich immer mehr eine optische Speichermethode durch. Dabei wird die magnetische Oberfläche durch eine optisch reflektierende Oberfläche ersetzt. In sie können mittels Laser Punkte eingebrannt werden (zu vergleichen mit dem Einbrennen oder Stanzen von Löchern in Papier), die einfallendes Licht schlechter reflektieren als die nichtverbrannte Oberfläche. Der Zustand der Oberfläche (glatt oder verbrannt) lässt sich durch Abtasten mit Hilfe eines Laserstrahls, der an der Oberfläche reflektiert wird, feststellen. Das Reflexionsvermögen dient als codierender Zustandsparameter. Der **CD-ROM-Speicher** (CD von Compact Disc) arbeitet nach diesem Prinzip. In der Massenproduktion werden CDs gepresst, ähnlich wie Schallplatten. CDs sind - ebenso wie Kassetten - als Musikträger allgemein bekannt. Der CD-ROM-Speicher ist, wie der Name sagt, ein reiner Lesespeicher. Seit einiger Zeit werden auch Schreib-Lese-CDs unter der Bezeichnung **CD-RW** (von ReWritable) angeboten. Die beiden codierenden Zustände einer Schreib-Lese-CD sind nicht “verbrannt” und “unverbrannt”, wie bei der nur einmal “brennbaren” CD, sondern es sind zwei Materialzustände mit etwas unterschiedlicher polymorpher Struktur, deren Reflexionsvermögen sich minimal, aber messbar voneinander unterscheiden. Durch genau dosierte Erwärmung und Abkühlung können die Zustände ineinander überführt werden.

Alle Speicher, die als Speichermedium eine rotierende Scheibe verwenden, fassen wir unter der Bezeichnung **Scheibenspeicher** zusammen. Dazu gehört die Festplatte, die Diskette, die CD und CD-RW und als jüngstes Produkt, das auf dem Markt erschienen ist, die **DVD** (Digital Versatile Disk), eine Weiterentwicklung der CD mit bedeutend höherer Speicherkapazität, sodass sie für Videoaufzeichnungen verwendbar ist.

Auf magnetische und optische Speicher werden wir nicht weiter eingehen. Näheres über Speicherverfahren und Massenspeicher siehe z.B. [Werner 95], [Messmer 95], [Biaesch 92]. Wenn im Weiteren von der Komponierung der Speicherhardware die Rede ist, sind rein elektronische Speicher gemeint, soweit nichts Gegenteiliges gesagt ist.

10 Realisierungsprinzip zirkelfreier boolescher Netze

Zusammenfassung der Kapitel 10 bis 12

Die erste Grundidee des elektronischen Rechnens, also die Idee, die Computerhardware auf der booleschen Algebra aufzubauen, ist die technische Fortsetzung eines langen Denkweges der Menschheit. Die über 2000-jährige Entwicklung der Logik von der *Syllogistik* des ARISTOTELES bis zu den *logischen Schaltungen* eines Computers ist ein einzigartiges Beispiel dafür, wie Philosophie *unmittelbar* zu Technik wird. Gegenstand der Logik als eines Zweiges der Philosophie war von jeher das sprachliche Modellieren des deduktiven Denkens.

Die zweite Grundidee des elektronischen Rechnens besteht darin, die beiden Stellungen (statisch stabilen Zustände) eines Stromschalters als Eingabewerte eines booleschen Operators und die beiden (statisch stabilen) Zustände "Strom fließt" und "Strom fließt nicht" als Ausgabewerte des Operators zu interpretieren. Der Schalter ist dann die Realisierung (die "*Interpretation*" im Sinne der Interpretation eines Kalküls) entweder des Identitätsoperators oder des Negators (NOT-Operators) je nachdem, wie die Zuordnungen zwischen den Zuständen und den booleschen Werten getroffen werden. Das Hindernis, das der erfindende Geist überwinden muss, um auf diese Idee zu kommen, ist das Erfassen und geschickte Ausnutzen der Arbitrarität des Codierens.

Ersetzt man den Schalter durch zwei in Reihe geschaltete Schalter, so realisiert die resultierende Schaltung den AND- bzw. NAND-Operator. Ersetzt man ihn durch zwei parallele Schalter, ergibt sich der OR- bzw. NOR-Operator. Es lässt sich also ein vollständiger Satz boolescher Operatoren aus Schaltern realisieren.

Um größere Schalternetze und damit boolesche Kompositoperatoren komponieren zu können, müssen *sämtlichen* booleschen Variablen elektrische Größen entsprechen, deren Werte abgenommen (gemessen, weitergeleitet) werden können. Dieses Problem wurde durch die Erfindung geeigneter *Schaltermechnanismen* gelöst. In den ersten Computern kam der Relais-Mechanismus zur Anwendung, in der *ersten Generation* elektronischer Rechner kam der Röhren-Mechanismus und in der *zweiten Generation* der Transistor-Mechanismus zu Anwendung. Alle drei Mechanismen beruhen darauf, dass durch Anlegen einer geeigneten Spannung an einen "Schalter" die Bewegung von Ladungsträgern durch den Schalter freigegeben bzw. unterbrochen wird.

Beim Relais-Mechanismus wird die Bewegung von Elektronen durch einen metallischen Leiter dadurch unterbrochen, dass der Leiter aufgetrennt ("zerschnitten") wird, indem ein mechanischer Schalter mittels Elektromagneten geöffnet wird. Beim Röhren-Mechanismus wird die Bewegung von Elektronen durch das Vakuum von einer Elektrode zur anderen (von der Kathode zur Anode) dadurch unterbrochen,

dass zwischen den Elektroden eine Potenzialbarriere aufgebaut wird (durch Anlegen einer geeigneten Spannung), die von den Elektronen nicht überwunden werden kann. Der Transistor-Mechanismus unterscheidet sich vom Röhren-Mechanismus dadurch, dass die Ladungsträger (diesmal Elektronen oder Löcher) sich durch einen Halbleiter bewegen und dass die Potenzialbarriere nicht im Vakuum, sondern im Halbleitermaterial aufgebaut wird.

Jeder der drei Schaltermechanismen erlaubt es, Schalternetze ("*Schaltkreise*") zu komponieren. Auf diese Weise lässt sich jeder boolesche Operator und jede Kombinationsschaltung als zirkelfreies Schalternetz aus Relais, Röhren oder Transistoren realisieren. Elektronisch realisierte elementare boolesche Operatoren werden häufig **Gatter** genannt.

In Rechnern der *dritten Generation* sind Schalternetze vorzugsweise mikroelektronisch realisiert. Große Schaltungen werden aus Bausteinen "zusammengesteckt", die oft Dioden- bzw. Transistorenmatrizen darstellen. Eine Diodenmatrix ist ein Leitergitter, genauer ein Tupel paralleler Eingabeleiter, das von einem Tupel paralleler Ausgabeleiter senkrecht geschnitten wird. An einem Schnittpunkt eines Eingabe- und eines Ausgabeleiters kann sich eine Diode befinden, über die das Potenzial des Eingabeleiters (ein Eingabebit) an den Ausgabeleiter übergeben wird.

Zwei Matrixtypen haben besondere Bedeutung, Decodiermatrizen oder Wort-Leitung-Zuordner und Codiermatrizen oder Leitung-Wort-Zuordner. Der *Wort-Leitung-Zuordner* bildet eine Menge von Binärworten auf eine Menge von Leitungen ab. Wenn auf das Eingabeleitungstupel eines der Binärworte gegeben wird, gibt die zugeordnete Ausgabeleitung und nur diese eine 1 aus. Der *Leitung-Wort-Zuordner* bildet eine Menge von Leitungen auf eine Menge von ein Binärworten ab. Wenn auf eine und nur eine der Leitungen eine 1 gegeben wird, gibt das Ausgabeleitungstupel das zugeordnete Binärwort aus. Beide Matrixtypen finden breite Anwendung, beispielsweise beim Aufbau sehr großer Kompositweichen und Kommutatoren, speziell beim Aufbau der Ein- und Ausgangsweichen eines Schreib-Lesespeichers, RAM genannt (von Random Access Memory).

Durch Hintereinanderschaltung eines Wort-Leitung- und eines Leitung-Wort-Zuordners entsteht eine mikroelektronisch realisierte Kombinationsschaltung, die auch *Codeumsetzer* genannt wird. Auf diese Weise können Kombinationsschaltungen mit Hunderten oder gar Tausenden von Ein- und Ausgabeleitungen hergestellt werden. Die Operationstafel der Kombinationsschaltung wird durch "Einlöten" von Dioden (bzw. von Transistoren, die als Dioden arbeiten) in die beiden Leitergitter festgelegt. Dafür sind verschiedene Technologien entwickelt worden. Man spricht auch von "Prägen" oder "Programmieren" der Matrizen.

Ein Codeumsetzer kann nicht nur als realer Operator, sondern auch als Speicher verwendet werden, der nach dem sog. strukturellen Speicherprinzip arbeitet. Dabei stellen die den beiden Matrizen gemeinsamen Leitungen je einen adressierten Speicherplatz dar, auf den über die Dioden der ersten Matrix zugegriffen werden kann und dessen Inhalt durch die Dioden der zweiten Matrix bei deren Herstellung

"eingeprägt" ("einprogrammiert) wird. Ein solcher Speicher heißt (elektronischer) ROM (Read Only Memory), weil er nur einmal beschrieben, aber beliebig oft gelesen werden kann.

10.1 Die zweite Grundidee des elektronischen Rechnens

Über den Aufbau traditioneller Rechner existiert eine umfangreiche Literatur. Es kann nicht unsere Aufgabe sein, den vorhandenen Büchern ein weiteres hinzuzufügen. Unser erklärtes Ziel ist es, die zentralen Ideen aufzuzeigen, die das elektronische Rechnen ermöglichen. Der Leser soll das Gefühl bekommen, dass er, wenn er selber diese Ideen gehabt hätte, eigentlich auch den Taschenrechner oder den PC (Personal Computer) hätte entwerfen können. Wir beschränken uns also auf das Grundsätzliche und überlassen es dem Leser, sich hinsichtlich spezieller Fragen, die ihn interessieren, in der Literatur kundig zu machen[1].

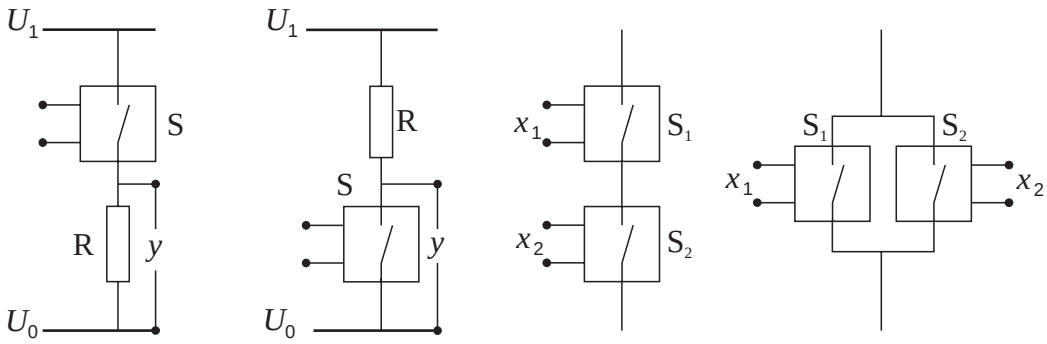
Während unser Gegenstand selber traditionell ist, nämlich die *traditionelle* Rechentechnik und der *traditionelle* Rechner, der sog. **Prozessorcomputer**, ist die folgende Darstellung weniger traditionell. Denn sie wendet durchgängig die Terminologie der USB-Methode an. Das kann sich infolge der Besonderheiten einiger Begriffe hier und da etwas erschwerend auswirken, wenn man bei der Lektüre dieses Kapitels traditionelle Darstellungen zu Rate zieht bzw. die unterschiedlichen Darstellungen miteinander vergleicht. Dann kann das Glossar nützliche Dienste leisten.

Ohne weitere Umschweife kommen wir zum eigentlichen Kern des elektronischen Rechnens. Um das Grundprinzip zu verstehen, auf dem die Funktionsweise eines *elektronischen* Rechners letzten Endes beruht, reicht die Kenntnis des ohmschen Gesetzes aus. Wir wollen es auf die in Bild 10.1 dargestellten Schaltungen (a) und (b) anwenden. Gezeigt sind jeweils zwei fett gezeichnete waagerechte Leiterschienen. An der unteren Schiene liegt die Spannung U_0, an der oberen die Spannung U_1 an. Spannungsquellen und Masse (Leiter mit Nullpotenzial) sind nicht eingezeichnet. Die Schienen sind über einen festen Widerstand R und einen mit ihm in Reihe geschalteten Schalter S miteinander verbunden.

Der Schalter sei über die Eingabespannung x steuerbar, z.B. mit Hilfe eines Elektromagneten, wie es beim Relais der Fall ist. Bei $x=x_0$ sei S geöffnet und bei $x=x_1$ geschlossen. Man beachte, dass x_0 und x_1 *Namen* (Code) für zwei bestimmte Spannungswerte sind. In geschlossenem Zustand sei der innere Widerstand des Schalters zu vernachlässigen im Vergleich zu R. Die Spannung, die sich an der Verbindungsleitung zwischen dem Schalter und dem Widerstand einstellt, ist mit y bezeichnet und kann am Ausgang gemessen werden.

---

1 Siehe z.B. [Matschke 86], [Schiffmann 92], [Flik 94],[Hennessy 94], [Messmer 95], [Werner 95].



(a) Identitäts-operator (b) Negator (c) AND- bzw. NAND-Operator (d) OR- bzw. NOR-Operator

Bild 10.1 Realisierung boolescher Operatoren mit Hilfe von Schaltern. **(a)** Identitätsoperator; **(b)** Negator; **(c)** Kompositschalter des AND- bzw. NAND-Operators; **(d)** Kompositschalter des OR- bzw. NOR-Operators.

Die Schaltungen (a) und (b) stellen Operatoren dar, deren Wertetafeln sich leicht angeben lassen. Zunächst analysieren wir die Verhaltensweise der Schaltungen. Bei geöffnetem Schalter fließt kein Strom, sodass an der Ausgabeleitung in Schaltung (a) die Spannung U_0 und in Schaltung (b) die Spannung U_1 liegt. Bei geschlossenem Schalter kehrt sich die Situation um, denn es fließt ein Strom, welcher bewirkt, dass die Spannungsdifferenz U_1-U_0 am Widerstand R abfällt. Für Schaltung (a) gilt also: $y=U_0$, wenn $x=x_0$, und $y=U_1$, wenn $x=x_1$, während Schaltung (b) die umgekehrte Zuordnung realisiert.

Wir nehmen nun eine *Umcodierung* vor und zwar codieren wir x_0 und U_0 in 0 und x_1 und U_1 in 1 um, was wegen der Arbitrarität des Codierens erlaubt ist. Es mag im ersten Augenblick überraschen, dass die Arbitrarität so weit geht und dass man völlig unterschiedliche Inhalte wie z.B. U_0 und x_0 durch das gleiche Codezeichen benennen darf. Tatsächlich liegt dem eine Interpretation der boolesche Werte 0 und 1 durch elektrische Werte zugrunde. Mit der Umcodierung wird vorausgesetzt, dass die Interpretation möglich ist und eingehalten werden kann.

Mit der Umcodierung wird die Wertetafel der Schaltung (a) zu derjenigen des booleschen *Identitätsoperators* (f_3 und f_5 in Bild 9.1), während die Wertetafel der Schaltung (b) zu derjenigen des *Negators* (f_{10} und f_{12} in Bild 9.1) wird[2]. Erklärt man die umcodierten Spannungswerte 0 und 1 zu booleschen Werten, so bedeutet das eine *Interpretation* boolescher Größen durch physikalische Größen, in diesem Fall durch Spannungen.

---

2 Es gibt noch andere Möglichkeiten, den booleschen Werten Spannungswerte zuzuordnen. Beispielsweise kann U_0 zu 1 und U_1 zu 0 umcodiert werden. Weiterhin muss die Ausgangsspannung nicht gegen die untere, sie kann auch gegen die obere Schiene gemessen werden. Natürlich wirkt sich das auf die Funktionstafel aus.

Mit der elektrotechnischen Realisierung des Negators ist noch nicht viel gewonnen. Um beliebige boolesche Funktionen und somit die boolesche Algebra zu realisieren, muss ein vollständiger Satz boolescher Operatoren in entsprechende Schaltungen überführt werden. Zu diesem Zweck ersetzen wir den Schalter in Schaltung (a) durch einen *Kompositschalter*, der aus zwei Schaltern komponiert ist. Die Komponierung "*in Reihe*" bzw. "*parallel*" erfolgen gemäß der Schaltung (c) bzw. (d) in Bild 10.1. Wir wollen die Funktionsweise der resultierenden Schaltungen analysieren.

Zunächst ersetzen wir den Schalter der Schaltung (a) durch die Reihenschaltung (c) aus zwei Schaltern S_1 und S_2. Es ergibt sich ein Operator mit zwei Eingängen, dessen Wertetafel (bei entsprechender Codierung) mit der des AND-Operators identisch ist, denn Strom fließt nur dann, wenn S_1 *und* S_2 geschlossen sind, m.a.W. am Ausgang liegt nur dann der Spannungswert 1, wenn an *beiden* Eingängen der Spannungswert 1 liegt. Verfährt man in gleicher Weise mit Schaltung (b), ergibt sich der NAND-Operator.

Damit verfügen wir bereits über eine technische Realisierung eines vollständigen Satzes boolescher Operatoren. Aus praktischen Gründen ergänzen wir ihn noch um den OR- und NOR-Operator. Diese erhält man, wenn man den Schalter S der Schaltungen (a) bzw. (b) durch die Parallelschaltung (d) ersetzt. Im Falle der Schaltung (a) ergibt sich der OR-Operator, denn nun fließt immer dann Strom, wenn S_1 *oder* S_2 geschlossen ist (oder wenn beide geschlossen sind). Wird der Schalter in Schaltung (b) durch zwei parallele Schalter ersetzt, ergibt sich der NOR-Operator. Die Idee, boolesche Operatoren mittels Schaltern zu realisieren, bezeichnen wir als **zweite Grundidee des elektronischen Rechnens**.

10.2 Zirkelfreie Schalternetze

Um aus den Schaltungen von Bild 10.1 komplexere boolesche Operatoren zu komponieren, müssen sie in genügender Stückzahl hergestellt und zu **Schalternetzen** miteinander verbunden werden. Dazu ist es notwendig, die Ausgänge der Schaltungen elektrisch an die Eingänge anzupassen, was mit Hilfe von Spannungsteilern und/oder zusätzlichen Spannungsquellen möglich ist. Auf diese Weise lässt sich im Prinzip jede boolesche Funktion und jede Kombinationsschaltung realisieren. Da jede binär codierte Wertetafel in eine Kombinationsschaltung überführt werden kann, ergibt sich folgender Schluss: *Zu jeder **Binärwortfunktion** mit bekannter Wertetafel lässt sich ein **Schalternetz** angeben, das die Funktion realisiert.*

Die *Mikroelektronik* hat Technologien entwickelt, die es ermöglichen, Netze aus Millionen von Schaltern in einem Halbleiterkristall mit einer Oberfläche von größenordnungsmäßig 1 cm^2 zu implementieren. Charakteristisch für solche Netze ist ihre matrixförmige Struktur. Bevor wir auf diese Technologien eingehen, soll ein kurzer historischer Abriß eingefügt werden, der den mühevollen Weg andeuten soll,

den viele Generationen von Philosophen, Naturwissenschaftlern und Technikern zurücklegen mussten, um das Fundament der heutigen Computertechnik zu legen. Wohin der Weg führen würde, davon hatten die Beteiligten entweder gar keine oder nur eine sehr vage Vorstellung. Und auch wir wissen nicht, wohin der Weg geht.

11 Historischer Einschub: Entwicklung der begrifflichen und physikalischen Basis der Rechentechnik

11.1 Von der Syllogistik des Aristoteles zur Schaltalgebra

Die erste Grundidee des elektronischen Rechnens, also die Idee, die Computerhardware auf der booleschen Algebra aufzubauen, setzt einen Jahrtausende langen Denkweg fort. Sie ist der erste Schritt vom *natürlichen* (biologischen, organismischen) Denken zum *künstlichen*, "elektronischen Denken". Die über 2000-jährige Entwicklung der Logik seit der *Syllogistik* des ARISTOTELES (384 - 322 v.u.Z.) bis zu den *logischen Schaltungen* eines Computers ist ein einzigartiges Beispiel dafür, wie Philosophie *unmittelbar* zu Technik wird. Zwar lässt sich die *mittelbare* Einflussnahme der Philosophie auf die Naturwissenschaft und die Mathematik und damit auf die Technik durch die gesamte Geschichte der Wissenschaft verfolgen; dass aber ein Zweig der Philosophie zu Erkenntnissen, Formulierungen und Sätzen gelangt, die geradenwegs durch immer weitergehende Abstraktion und semantische Objektivierung schließlich in ein Kalkül übergehen und durch Interpretation des Kalküls zu technischen Produkten, zu elektronischen Schaltungen werden, das ist wohl einmalig in der kulturellen Evolution.

Die Ursache dafür ist verständlich. Seit Aristoteles ist das Ziel der Logik das Verstehen und Beschreiben des deduktiven Denkens. Inhalt der technischen Informationsverarbeitung ist das maschinelle Rechnen und Schließen, m.a.W. der technische Nachvollzug des deduktiven Denkens. Es ist also ganz natürlich, dass die Rechentechnik sich die Ergebnisse der Logik zunutze macht. In Kap.3 [3.2] hatten wir vereinbart, unter *Denken* das gedankliche Modellieren der Welt zu verstehen, und in Kap.7.1 [7.3] hatten wir die Fähigkeit zum deduktiven Modellieren (zum deduktiven Denken) als deduktive Intelligenz bezeichnet. Damit lässt sich die "Verwandtschaftsbeziehung" zwischen Logik und künstlicher Intelligenz schlagwortartig folgendermaßen charakterisieren: *Die Logik ist die **natürliche** Mutter der **künstlichen** deduktiven Intelligenz.*

Die Beziehung zwischen Logik und künstlicher Intelligenz wird besonders deutlich, wenn man den Gegenstand der Logik folgendermaßen bestimmt: *Gegenstand der Logik ist das sprachliche Modellieren des gedanklichen deduktiven Modellierens.* Aus dieser Formulierung folgt die Vorreiterrolle der Logik nicht nur für die KI-Forschung, sondern für jede exakte Wissenschaft. Außerdem spiegelt sich in ihr der zirkuläre Charakter (Modellieren des Modellierens) der Logik wider. Er entspricht dem zirkulären Charakter der Informatik (vgl. Kap.6.3 [6.5]) und tritt z.B. in dem Umstand zutage, dass die Logiker die Mechanismen der Begriffsbildung nicht

nur als Phänomen (als Operand) untersuchen, sondern auch als Denkmittel (als Operator) benutzen. Das betrifft insbesondere die abstrahierende Begriffsbildung durch Klassifikation und Generalisierung sowie die konkretisierende Begriffsbildung durch Instanzierung und Präzisierung (vgl. Bild 5.4). Ein anschauliches Beispiel dafür ist die Syllogistik des Aristoteles.

Damit beginnen wir einen kurzen Streifzug durch die Geschichte der Logik[1] unter einem bestimmten Aspekt, nämlich im Hinblick auf das uns interessierende Endergebnis: die semantische Objektivierung des Modells des deduktiven Denkens in Form eines Kalküls und dessen Interpretation durch Schalternetze, also durch Hardware. Der Weg dahin ist durch schrittweise Abstraktion und im letzten Schritt durch konkretisierende Interpretation gekennzeichnet. Wir werden einige wenige Marksteine dieses Weges beleuchten.

Die antiken Untersuchungen zum deduktiven Denken beziehen sich natürlicherweise auf das *Schlussfolgern* und weniger auf das *Rechnen*. Das bekannteste Beispiel für das von Aristoteles systematisch untersuchte schlussfolgernde Denken lautet: "*Alle Menschen sind sterblich. Sokrates ist ein Mensch. Also ist Sokrates sterblich.*" Das Beispiel ist in vielen Lehrbüchern der Logik zu finden. Es stammt jedoch nicht von Aristoteles und ist für die von ihm analysierte Schlussweise auch nicht ganz zutreffend (s.u.).

Der Schluss beruht auf folgender Voraussetzung. Eine Eigenschaft, die eine Klasse von Individuen charakterisiert, trifft auf jedes einzelne Individuum zu (*Axiom des kategorialen Syllogismus*), oder unter Verwendung des Prädikatbegriffs: Ein Prädikat $P(x)$, das eine Klasse X festlegt, ist für jedes Exemplar (jedes Element, jede Instanz) der Klasse erfüllt. Die formale prädikatenlogische Notation lautet:

$$\forall x \in X\ P(x),$$

zu lesen als "Für alle x aus X ist das Prädikat $P(x)$ erfüllt". In der Terminologie der objektorientierten Programmierung (siehe Kap.18) würde man von Merkmalsvererbung sprechen und sagen: Das Merkmal "Sterblichkeit" wird von der Klasse der Menschen an ihre Instanzen (an jeden einzelnen Menschen) "*vererbt*".

In dem Beispiel beruht das Schließen auf Instanzieren. Zieht man aus der Prämisse, dass alle Menschen sterblich sind, den Schluss, dass alle Griechen sterblich sind, so liegt dem nicht *Instanzieren*, sondern *Präzisieren* zugrunde, denn das Merkmal wird an eine *Unterklasse* vererbt. Aristoteles selber hat das präzisierende, nicht das instanzierende Schließen untersucht.

Diese wenigen Bemerkungen zeigen, dass es sich bei der aristotelischen Syllogistik um eine *Klassenlogik* handelt. Interessanterweise setzt sich die älteste von den Philosophen untersuchte Schlussweise (nämlich die auf Merkmalsvererbung beruhende) erst in dem gegenwärtig jüngsten, dem *objektorientierten* Programmierpara-

1 Näheres zur Geschichte der Logik siehe z.B. in [Berka 83] oder [Störig 89].

digma durch. Früher entstandene Paradigmen machen von der Vererbung keinen expliziten Gebrauch. Es ist also eine Art Gegenläufigkeit der Entwicklung zu beobachten, die an die Gegenläufigkeit der Entwicklung der natürlichen und der künstlichen Intelligenz erinnert (vgl. Kap.7.1).

Das Wort "Logik" führten die Stoiker[2] ein. Aristoteles selber nannte die Wissenschaft vom Schlussfolgern *Analytik*. Ein wichtiger Beitrag der Stoa zur Logik ist die Benutzung der "Wenn...dann"-Wendung als eines standardmäßigen Mittels zur Artikulierung des gedanklichen Schließens (des bewussten logischen Schlussfolgerns). Wenn heutzutage ein Mensch erklären soll, was er unter *Schlussfolgern* (*Schließen* oder *Folgern*) versteht, wird er sich sehr wahrscheinlich dieser Wendung bedienen. Mit ihr werden sowohl *logische* Zusammenhänge (*Prämisse - Konklusion*) als auch *kausale* Zusammenhänge (*Ursache - Wirkung*) beschrieben. Der Beziehung zwischen beiden Arten von Zusammenhängen waren wir in Kap.8.2.5 auf den Grund gegangen.

Vorwegnehmend sei gesagt, dass "Wenn...dann"-Wendungen in der Programmierungstechnik eine wichtige Rolle spielen, beispielsweise in Form bedingter Anweisungen (siehe z.B. Bild 15.2b) oder in Entscheidungstabellen, die bei der automatischen Prozesssteuerung zum Einsatz kommen (siehe Kap.12.3.4 [12.5]). Dabei steht i.d.R. der *logische* Aspekt im Vordergrund. Auch in diesem Kapitel haben wir es nur mit logischen "Wenn...dann"-Zusammenhängen zu tun, also mit Zusammenhängen der Art: Wenn die Prämisse erfüllt ist (wenn die *Prämissenaussage* zutrifft), dann ist auch die Konklusion erfüllt (dann trifft auch die *Konklusionsaussage* zu). Man beachte, dass die Gültigkeit der Prämissen eine hinreichende, aber keine notwendige Bedingung der Konklusion ist.

In der antiken Philosophie gab es bereits Ansätze in Richtung eines Formalismus, der Schlussfolgern in Rechnen überführt. Doch erst GOTTFRIED WILHELM LEIBNIZ (1646-1716) sprach die Idee eines Kalküls auf der Grundlage einer formalisierten Sprache des logischen Schließens explizit aus. Er wollte die *Wahrheit* einer Aussage *berechenbar* machen. Er schrieb: "*Das einzige Mittel, unsere Schlussfolgerungen zu verbessern ist, sie ebenso anschaulich zu machen, wie es die der Mathematiker sind, derart, dass man seinen Irrtum mit den Augen findet und, wenn es Streitigkeiten unter Leuten gibt, man nur zu sagen braucht: 'Rechnen wir!' ohne eine weitere Förmlichkeit, um zu sehen, wer recht hat*"[3].

Es stellt sich die Frage, welcher Zusammenhang zwischen einem solchen "Wahrheitskalkül", dem Schlussfolgern im Sinne von "Wenn...dann"-Zusammenhängen und der aristotelischen Klassenlogik besteht. Um ihn aufzuzeigen, formulieren wir das obige Beispiel folgendermaßen um:

---

2 Philosophen der als Stoa bezeichneten griechischen Philosophenschule (3. bis 1. Jahrh.v.u.Z.).

3 Zitiert nach [Berka 83].

> Wenn die Aussage “Alle Menschen sind sterblich” zutrifft UND wenn außerdem die Aussage “Alle Griechen sind Menschen” zutrifft, dann trifft auch die Aussage “Alle Griechen sind sterblich” zu.

Die Richtigkeit des Schlusses kann nach den Regeln des *Aussagenkalküls* berechnet werden, eines Kalküls, der den Intentionen von Leibniz wohl ziemlich nahe kommt, aber bedeutend jünger ist. Der Aussagenkalkül - auch *Aussagenlogik* oder *Aussagenalgebra* genannt - ist eine *Interpretation* der booleschen Algebra. Die booleschen Variablen werden als Platzhalter für Aussagen interpretiert und die booleschen Werte 0 bzw. 1 als “falsch” bzw. “wahr” [9.12].

Mit dieser Interpretation *berechnet* sich der Wahrheitswert der gesamten Prämissenaussage in obigem Beispiel aus den beiden mit UND verbundenen Prämissenaussagen nach der Wertetafel der Konjunktion (der AND-Operation). Daraus folgt mit Notwendigkeit die Gültigkeit der Konklusionsaussage, wenn beide Prämissenaussagen zutreffen (beide Bedingungen erfüllt sind). Die Gültigkeit der gesamten Prämisse ist aber keine notwendige Bedingung für die Gültigkeit der Konklusion. Die Griechen könnten auch dann sterblich sein, wenn nicht alle Menschen sterblich wären, sondern beispielsweise die Römer unsterblich. Prämisse und Konklusion sind einander nicht *äquivalent* im Sinne der booleschen Äquivalenz, sondern die Gültigkeit der Prämisse *impliziert* die Gültigkeit der Konklusion. Der “Wenn...dann”-Beziehung entspricht die boolesche *Implikation*.

Die formale Entsprechung zwischen boolescher Algebra und Aussagenalgebra ist ein Beispiel für *isomorphe* Algebren oder Kalküle. Grob gesagt besteht zwischen zwei Algebren (Kalkülen) dann eine **Isomorphie** (eine **isomorphe Abbildung**), wenn sowohl zwischen den Operationen der beiden Kalküle als auch zwischen den Wertemengen der beiden Kalküle *eineindeutige* (in beiden Richtungen eindeutige) Entsprechungen bestehen und wenn einander entsprechende Operationen einander entsprechende Resultate liefern.

Vielleicht sieht mancher Leser in der Isomorphie zwischen Aussagenalgebra und boolescher Algebra eine haarspalterische Selbstverständlichkeit. Eine solche Ansicht würde der geistigen Leistung von GEORGE BOOLE (1815-1864) jedoch nicht gerecht. Sein Verdienst ist es, von der Bedeutung der Wörter “wahr” und “falsch” und von der Bedeutung “Aussage” abstrahiert zu haben und so den Weg zu einer rein formalen, d.h. von jeder Semantik freien Algebra zu öffnen (historisch geht die Aussagenlogik der booleschen Algebra voraus). Einen derartigen Abstraktionsschritt als erster zu tun, erfordert Genialität. Im Nachvollzug scheint er fast selbstverständlich zu sein.

Die obige Umformulierung des Sterblichkeits-Beispiels weist auf eine andere Isomorphie hin, die zwischen Aussagenalgebra und Klassenlogik; üblicherweise spricht man von Isomorphie zwischen *Aussagenalgebra* und **Mengenalgebra** (auch *Mengenlehre* genannt)[4]. Die Mengenlehre wurde von GEORG CANTOR (1845-1918) entwickelt. Dabei entspricht der aussagenlogischen UND- bzw. ODER-Operation

die mengenlogische *Durchschnitts*-bzw. *Vereinigungs*-Operation. Das sei an folgendem Beispiel illustriert.

Die Menge aller Berlinerinnen ist die *Durchschnittsmenge* der Menge aller Menschen, die in Berlin zu Hause sind, und der Menge aller Menschen weiblichen Geschlechts. Sie ist durch das Prädikat "*ist in Berlin zu Hause* UND *ist weiblichen Geschlechts*" definiert. Ersetzt man das logische UND durch das logische ODER, wird durch das neue Prädikat die *Vereinigungsmenge* festgelegt, das ist die Menge aller Menschen weiblichen Geschlechts erweitert um die (im Vergleich dazu kleine) Menge aller Menschen, die in Berlin zu Hause sind.

Der Inhalt des vorangehenden Absatzes kann auf wenige Zeichen komprimiert werden, wenn man ihn in die Sprache der Prädikatenlogik (des Prädikatenkalküls) übersetzt ("umcodiert"). Bezeichnet man die beiden miteinander geschnittenen bzw. vereinigten Mengen mit M_1 und M_2 und die sie definierenden Prädikate mit P_1 und P_2 und verwendet für die Durchschnitts- bzw. Vereinigungs-Operation das übliche Symbol $\cap$ bzw. $\cup$, so lassen sich die Zusammenhänge des Beispiels folgendermaßen formal notieren ("UND" bzw. "ODER" ist durch "AND" bzw. "OR" ersetzt):

$$M_1 \cap M_2 = \{x\text{: } (P_1 \text{ AND } P_2)\} \text{ bzw.} \tag{11.1a}$$

$$M_1 \cup M_2 = \{x\text{: } (P_1 \text{ OR } P_2)\}. \tag{11.1b}$$

Die beiden geschweift geklammerten Ausdrücke legen je eine Menge fest. Die Zeichenkette $\{x\text{: } (...)\}$ ist zu lesen als: "Menge aller x, für die (...) zutrifft".

Falls dieser oder jener Leser mit der prädikatenlogischen Notation Mühe hat oder wenn er die "Umcodierung" nicht nachvollziehen will, weil er Formeln nicht liebt, verliert er nichts, denn die Umcodierung bringt keinerlei Erkenntnisgewinn. Das heißt nicht, dass die Prädikatenlogik überflüssig ist. Ihr Nutzen liegt, wie der Nutzen jedes Kalküls, in der semantischen Objektivierung (vgl. Kap.5.4 [5.6]) und darin, dass komplizierte Zusammenhänge durch Abstraktion leichter durchschaubar und handhabbar werden. Der Prädikatenkalkül operiert mit Prädikaten, also mit sprachlichen Ausdrücken, welche die Form von Aussagen besitzen und eine oder mehrere Variablen enthalten (vgl. Kap.8.3[8.18]).

Wir übergehen den vollständigen Beweis der Isomorphie zwischen Aussagen- und Mengenalgebra. Mit ihm wäre dann auch die Isomorphie zwischen Mengen- und boolescher Algebra gezeigt. Der Vollständigkeit halber sei nur erwähnt, dass der Negation die **Komplementbildung** entspricht. Die **Komplementmenge** einer Menge M enthält alle Elemente, die *nicht* zu M gehören. Beispielsweise ist die Komplementmenge zur Menge aller Griechen die Menge aller Nichtgriechen. Damit die

4 Der Begriff der Menge ist etwas allgemeiner als der der Klasse. Er schließt die Möglichkeit ein, die Elemente einer Menge nicht durch ein Prädikat (*intensional*), sondern durch Aufzählung (*extensional*) festzulegen. Der Klassenbegriff wird hier nicht im Sinne der axiomatischen Mengenlehre verwendet.

Menge der Nichtgriechen eindeutig festgelegt ist, muss eine sog. **Allmenge** existieren, von der die Menge *M* gewissermaßen "subtrahiert" wird. Die Rolle der Allmenge kann z.B. die Menge aller Menschen spielen.

Es gibt eine weitere Interpretationsmöglichkeit der booleschen Algebra und zwar die für die Rechentechnik ausschlaggebende, von der in Kap.10 die Rede war, die Interpretation durch die *Schaltalgebra*. Die boolesche Algebra wird zur Schaltalgebra, wenn die booleschen Operatoren als Schalter, genauer als Kompositschalter (vgl. Bild 10.1) und die booleschen Werte 0 und 1 als "Schalter geöffnet" bzw. "Schalter geschlossen" interpretiert werden. Auf der Isomorphie zwischen boolescher Algebra und Schaltalgebra beruhten die Überlegungen des Kapitels 10, beziehungsweise - historisch betrachtet - die Überlegungen förderten die Isomorphie zutage.

Aus heutiger Sicht ist es fast selbstverständlich, dass ein Ingenieur, der eine umfangreiche Kombinationsschaltung entwirft (beispielsweise zur Steuerung einer Waschmaschine), sich der booleschen Algebra bedient. Das war keineswegs selbstverständlich, bevor die Isomorphie zwischen boolescher Algebra und Schaltalgebra entdeckt worden war. Findige Ingenieure entwickelten sich ihre eigene "Schaltalgebra", um sich die Arbeit zu erleichtern. Es bedurfte einer erheblichen *intuitiven* Intelligenz, um die genannte Isomorphie zu *sehen*. CLAUDE ELWOOD SHANNON sah sie, derselbe Shannon, der später die Informationstheorie entwickelte.

Der Intuition Shannons muss eine gedankliche Annäherung logischer Begriffe und Relationen an schaltungstechnische Begriffe und Relationen durch Abstraktion bis hin zu ihrer teilweisen Identifizierung vorangegangen sein. Einer ähnlichen Annäherung oder *Konvergenz* unterschiedlicher Begriffe durch Abstraktion waren wir bereits in Kap.8.5 [8.38] begegnet, dort hinsichtlich der Begriffe *Algebra, Kalkül* und *algorithmisches System*. Wir hatten die *begriffliche Konvergenz* (die semantische Konvergenz unterschiedlicher Begriffe) durch Abstraktion als charakteristisches Phänomen des menschlichen Denkens erkannt und als *Konvergenzprinzip des Denkens* bezeichnet.

Betrachtet man die Entwicklung der Wissenschaft im Allgemeinen, fällt einem auf, dass sie ihre entscheidenden Fortschritte offenbar der begrifflichen Konvergenz
1 verdankt. In diesem Sinne kann man auch vom *wissenschaftlichen Konvergenzprinzip* sprechen. Die Fortschritte, von denen hier die Rede ist, beruhen in der Regel auf genialen Ideen, die von dazu geeigneten Gehirnen hervorgebracht werden. Dabei handelt es sich immer um ein *Erfinden*, denn die Bedeutung des Wortes *Idee* schließt ein, dass diese nicht aus Bekanntem abgeleitet werden kann (vgl. die Definition in Kap.7.1 [7.4]). Solche Ideen führen zu neuen Erkenntnissen und zum Forschritt der Wissenschaft.

Ein anderes Beispiel für die Wirkungsweise des Konvergenzprinzips entnehmen wir der Physik. Im Denken JAMES MAXWELLS konvergierten durch Abstraktion die Begriffe und Relationen aus den Bereichen der Elektrizität einerseits und der Optik andrerseits. Das Ergebnis war eine einheitliche Theorie, artikuliert in den maxwell-

schen Gleichungen. Später werden wir weiteren Beispielen aus der Informatik begegnen.

Damit sind wir am Ende unseres Streifzugs durch die Logik angekommen, müssen aber noch eine Ergänzung anfügen. Wir haben nämlich von Boole zu Shannon einen allzu großen Sprung gemacht und des Mannes nicht gedacht, der die Idee eines "*Wahrheitskalküls*" als erster tatsächlich verwirklicht hat, nachdem Boole und viele andere Forscher wichtige Vorarbeiten geleistet hatten. Die Rede ist von FRIEDRICH LUDWIG GOTTLOB FREGE (1848-1929). Er hat eine formalisierte Sprache entwickelt, wie sie Leibniz vorgeschwebt haben mag, und **Begriffsschrift** genannt. Sie hat sich jedoch nicht durchgesetzt. Diejenige Algebra, die heute *boolesche Algebra* genannt wird und deren Grundlagen in Kap.9.3 skizziert wurden, hat ihre endgültige Ausformung erst durch die Nachfolger von Frege erhalten.

11.2 Rechnergenerationen

An die historische Skizze des *geistigen* (begrifflichen) Hintergrundes der Rechentechnik soll sich eine entsprechende Skizze der *materiellen* (physikalisch-technischen) Basis der elektronischen Rechentechnik anschließen. Zu diesem Thema existiert eine umfangreiche Literatur[5].

Die ersten *nicht* rein mechanisch arbeitenden Rechner waren *Relaisrechner*. Als Schalter dienten Relais, aus denen das *Rechenwerk* (die Rechenoperatoren) und der *Speicher* (die 1-Bit-Speicher) aufgebaut wurden. Der erste funktionstüchtige programmierbare Relaisrechner mit der Bezeichnung Z3 wurde von KONRAD ZUSE entwickelt und 1941 fertiggestellt. An der Weiterentwicklung von Relaisrechnern wurde etwa noch 10 Jahre hindurch gearbeitet, doch erreichte sie bald ihre Grenzen und zwar aus mehreren Gründen. Relais nehmen zu viel Platz in Anspruch, sie sind zu fehleranfällig, ihre Leistungsaufnahme und Wärmeabgabe ist zu hoch, und sie sind zu teuer. Dadurch, dass es gelang, Schalter zu entwickeln, die in all diesen Eigenschaften das Relais um das Millionenfache übertreffen, wurde die moderne Rechentechnik möglich.

Die Entwicklung vollzog sich in drei großen Schritten, die durch eine Reihe von Erfindungen ausgelöst wurden. Dabei brachte jeder Schritt eine enorme Miniaturisierung der Schalter und Schalternetze mit sich. Im ersten Schritt wurde das Relais durch eine Elektronenröhre, die Triode, und im zweiten durch ein Halbleiterbauelement, den Transistor ersetzt. Der dritte Schritt nutzte die Erfolge der Mikroelektronik, wodurch es möglich wurde, riesige **Schaltkreise** (sprich: Schalternetze) auf kleinen Halbleiterplättchen zu implementieren. Die drei Schritte unterteilen die Geschichte der Rechentechnik in drei Epochen und die Rechner in drei **Generationen**. Jede

5 Siehe z.B. [Hennessy 94] und die dort zitierte Literatur.

Generation	Zeitraum	Technologie	Größe/Aufstellungsart
(0.)	40er Jahre	Relais	Labor
1.	1950-1959	Röhren	Haus-Saal
2.	1960-1968	Transistoren	Saal-Zimmer
3.	1969-1977	Integrierte Schaltungen	Zimmer-Tisch
4.	seit 1978	LSI, VLSI	Tisch-Tasche

Bild 11.1 Rechnergenerationen

Epoche entwickelte ihre charakteristischen Bauelemente und ihre charakteristischen Technologien für den Aufbau von Schalternetzen.

Der erste elektronische, röhrenbestückte Universalrechner wurde während des zweiten Weltkrieges in den USA von J.PRESPER ECKERT und JOHN MAUCHLY gebaut. Der Rechner wurde ENIAC (Electronic Numerical Integrator and Calculator) genannt und von der Armee für militärische Zwecke verwendet. In den Entwurf der ENIAC flossen wichtige Ideen von JOHN VON NEUMANN ein. Eine Weiterentwicklung, die UNIVAC I, wurde 1951 als erster kommerzieller Rechner auf den Markt gebracht. Bei der folgenden kurzen Behandlung der Rechnergenerationen werden wir uns auf die Grundideen beschränken.

Ein elektrischer Schalter schließt oder unterbricht einen elektrischen Stromkreis, d.h. er ermöglicht bzw. unterbindet das Passieren beweglicher Ladungsträger, in der Regel freier Elektronen. Insofern stellt ein Schalter ein Tor für Elektronen dar.

Das Relais unterbricht den Strom dadurch, dass es den Leiter, durch den die Elektronen fließen, “zerschneidet”, indem ein Elektromagnet einen Kontakt mechanisch trennt. Die Schalterfunktion der Triode und des Transistors beruht darauf, dass den sich bewegenden Elektronen bzw. “Löchern” (s.u.) eine Potenzialbarriere in den Weg gestellt werden kann, indem eine ortsfeste negative bzw. positive elektrische Ladung aufgebaut wird, welche die ankommenden Ladungsträger zurückstößt.

Im Falle der Triode bewegen sich Elektronen durch das *Vakuum* von der Kathode zur Anode. Die Barriere wird aufgebaut, indem ein Drahtgitter durch eine äußere Spannung, die sog. *Gitterspannung*, negativ aufgeladen wird. Da der Elektronenstrom auf seinem Wege zur Anode das Gitter passieren muss, kann er durch Anlegen einer ausreichend hohen negativen Gitterspannung unterbrochen werden. Die Triode gelangte in den dreißiger Jahren zur technischen Reife, wozu WALTER SCHOTTKY wesentlich beitrug.

Im Falle des Transistors bewegen sich Elektronen durch das Kristallgitter eines *Halbleiters*. Als Halbleitermaterial hat sich z.B. Silizium als besonders geeignet erwiesen. Die ortsfeste Potenzialbarriere wird durch gittergebundene Ladungen, i.d.R. durch *Fremdatome* erzeugt, deren Ionisierungswahrscheinlichkeit (d.i. die über die Zeit gemittelte Differenz zwischen Kernladungzahl und Anzahl der Hüllenelektronen) durch eine äußere Spannung, verändert und so der Strom durch den Halbleiter gesteuert und gegebenenfalls unterbrochen werden kann. Das dosierte Einbringen (*Dotieren*) der Fremdatome kann mittels Diffusion bewerkstelligt werden.

Aus praktischen Gründen ist es vorteilhaft, in Halbleitern zwischen negativen und positiven freien Ladungsträgern zu unterscheiden; letztere werden auch *Löcher* genannt, weil sie gewissermaßen "bewegliche Löcher" oder "Blasen" in der strömenden "Elektronenflüssigkeit" darstellen. In diesem Bilde ist zwischen sog. *n*-Leitern, die negative Ladungsströme (Elektronenströme) tragen, und *p*-Leitern, die positive Ladungsströme (Löcherströme) tragen, zu unterscheiden.

In der Grenzschicht zwischen einem *p*- und einem *n*-Leiter bildet sich ein Potenzialsprung aus. Er wird in Halbleiterdioden zur Stromgleichrichtung ausgenutzt. Ein Transistor besteht aus drei Zonen; entweder liegt eine *p*-leitende Zone zwischen zwei *n*-leitenden oder eine *n*-leitende Zone liegt zwischen zwei *p*-leitenden Zonen. Der Transistor wurde in seiner ersten Variante (*Spitzentransistor*) 1947 von JOHN BARDEEN und WALTER H. BRATTAIN und in seiner zweiten Variante (*Flächentransistor*) 1948 von WILLIAM B. SHOCKLEY erfunden. Alle drei erhielten 1956 gemeinsam den Nobelpreis.

Für die dritte Rechnergeneration lässt sich kaum eine einzelne Erfindung als Auslöser der Entwicklung angeben. Eine wichtige Rolle spielte die Erfindung und ständige Vervollkommnung der *Maskentechnik*. Sie macht es möglich, in einem Siliziumkristall äußerst feine räumliche Strukturen aus Zonen unterschiedlichen Leitungstyps und unterschiedlicher Leitfähigkeit herzustellen. Auf diese Weise lassen sich auf einem einzigen Kristallplättchen (*Chip*) viele Transistoren, Widerstände und andere Schaltelemente zu evtl. sehr großen Schaltkreisen *integrieren*. Die Technologie wurde ständig weiterentwickelt, aus *integrierten* Schaltkreisen wurden *LSI-Schaltkreise* (LSI von Large Scale Integration) und VLSI-Schaltkreise (Very Large Scale Integration).

Zuweilen wird die LSI- und VLSI-Technik als das bestimmende Charakteristikum der *vierten* Rechnergeneration angesehen (siehe Bild 11.1). Ein anderer Vorschlag verbindet die vierte Rechnergeneration mit der Einführung massiv paralleler Verarbeitungsprinzipien, auf die in diesem Buch nicht eingegangen wird. Die Japaner haben den Begriff der *fünften* Generation eingeführt, allerdings in einem Sinne, der von dem ursprünglichen Prinzip abweicht, eine Generation mit einer bestimmten Hardwaretechnologie zu verbinden. Derartige nicht nur an der Hardware orientierten Generationsbezeichnungen haben sich aber nicht durchsetzen können.

In der modernen Mikroelektronik kommt vorzugsweise der sog. MOS-Feldeffekt- 2
transitor, kurz MOS-FET zur Anwendung (MOS von Metal-Oxide-Semiconductor).

Die Steuerung des elektrischen Stroms durch einen MOS-FET kann mit der Steuerung des Wasserflusses durch einen Kanal mit leichtem Gefälle verglichen werden, der einen Gummiboden besitzt. Die pro Sekunde durchfließende Wassermenge kann dadurch vergrößert oder verkleinert wird, dass der Gummiboden nach unten oder oben ausgewölbt wird. Durch starke Wölbung des Bodens nach oben kann das Wasser vollständig aus dem Kanal verdrängt werden; der Kanal trocknet aus und es gibt nichts mehr, was fließen kann. Dem entspricht im MOS-FET das Verdrängen freier Ladungstäger aus einem leitenden "Kanal" in einem Halbleiter ("Semiconductor"). Wenn keine freien Ladungsträger mehr vorhanden sind, kann kein Strom mehr fließen. Zu erreichen ist dies durch Aufladung einer Elektrode, einer Metallschicht ("Metal"), die entlang des Kanals angebracht (aufgedampft) und gegen den Halbleiter durch eine Oxydschicht ("Oxide") isoliert ist. Dadurch wird ein elektrisches Feld aufgebaut, von dessen Größe die Anzahl der freien Ladungsträger und damit die Leitfähigkeit des Kanals abhängt.

Allgemein heißen Transistoren, deren Leitfähigkeit durch ein *äußeres Feld* gesteuert werden kann, das durch isolierte Ladungen erzeugt ist, *Feldeffekttransistoren.* Vor der Erfindung der Feldeffekttransistoren erfolgte die Steuerung ausschließlich durch ein *inneres* Feld, das durch Anlegen einer Potenzialdifferenz über leitende Verbindungen mit dem Halbleiter in diesem erzeugt wird.

Gemeinsam mit der Miniaturisierung durch die LSI- und VLSI-Technik erhöht sich die Arbeitsgeschwindigkeit elektronischer Schaltungen und speziell des Computers. Der Grund liegt in der Verkleinerung der Laufzeiten der Ladungsträger durch den Halbleiter, insbesondere durch die Bereiche großen Potenzialgefälles in den
3 Grenzschichten. Denn von der Laufzeit hängt die *Bandbreite* des Transistors ab. Die Bandbreite charakterisiert, anschaulich ausgedrückt, die Reaktionsgeschwindigkeit des Transistors auf Änderungen der anliegenden Spannungen. Je schneller die Ladungsträger die Schicht durchqueren , umso kürzer ist die Schaltzeit des elektronischen Schalters, und umso kürzer ist die Zeitdauer, die ein Flipflop benötigt, um in seinen anderen Zustand überzuspringen. Diese Zeitdauer begrenzt die Anzahl der Bits, die ein Flipflop pro Sekunde scharf voneinander trennen kann und damit die maximale Taktfrequenz eines Schaltkreises bzw. Computers.

Für die Leistungsfähigkeit eines Computers ist neben seiner Taktfrequenz die Kapazität seines Arbeitsspeichers entscheidend. Die Arbeitsspeicher moderner Rechner sind in aller Regel VLSI-Schaltkreise. Auf ihre Arbeitsweise wird in Kap.13.2.2 eingegangen. Durch die fortscheitende Miniaturisierung, die noch nicht ihr Ende erreicht hat, steigt die Speicherkapazität elektronischer Speicher ständig. Parallel dazu läuft die Entwicklung neuer Speicherprinzipien. Gegenwärtig werden Speicherchips mit Milliarden von Schaltelementen produziert, und Speicherkapazitäten von über 10^8 Bit werden erreicht.

Neben den rein elektronischen [9.24] Speichern kommen auch magnetische und optische Speicherverfahren zur Anwendung. Die Grundprinzipien ihrer Funktionsweise waren in Kap.9.6 dargelegt worden. In der Speichertechnik der 1. und 2.

Rechnergeneration spielten magnetische Effekte die dominierende Rolle. Damals war die Herstellung rein elektronischer Speicher mit der erforderlichen Speicherkapazität zwar möglich, aber zu teuer. Rechner der 1. und 2. Generation waren mit unterschiedlichen Speichern ausgerüstet, häufig mit Trommelspeichern. Sie alle sind inzwischen aus der Computertechnik verschwunden. Das gilt auch für Ferritkernspeicher, die in Rechnern der 2. und der 3. Generation zum Einsatz kamen. Mit Ausnahme des Trommelspeichers finden Speicher mit bewegtem Träger auch heute noch breite Anwendung, allerdings nicht als *Arbeitsspeicher*, sondern als zusätzliche ("externe") *Massenspeicher* zur Aufbewahrung großer Datenmengen. Die relativ lange Zugriffszeit nimmt man dabei in Kauf[6].

6 Siehe z.B. [Messmer 95], [Werner 95].

12 Technische zirkelfreie boolesche Netze

12.1 Codeumsetzer und elektronischer Festwertspeicher

Nach diesem historischen Einschub nehmen wir den Faden wieder auf, den wir am Ende des Kapitels 10 unterbrochen haben, und wenden uns den technologischen Prinzipien der Mikroelektronik zu. Als Ausgangspunkt wählen wir ein Speicherproblem und fragen: Wie lässt sich die Wertetafel einer Binärwortfunktion maschinell verfügbar machen? Zwei unterschiedliche Lösungen bieten sich an. Zum einen kann die Wertetafel als Kombinationsschaltung realisiert werden [9.16]. Dann sprechen wir von **struktureller Speicherung**, weil die Wertetafel in der *Struktur* der Schaltung "gespeichert" ist. Zum anderen kann die Wertetafel in einem Speicher abgelegt werden, sodass auf die gewünschten Werte über die Adressen der jeweiligen Speicherplätze zugegriffen werden kann. Dann sprechen wir von **adressierter Speicherung**.

Fragt man sich, nach welchem Prinzip das menschliche Gedächtnis arbeitet, wird die Antwort offenbar zugunsten der strukturellen Speicherung ausfallen. In künstlichen neuronalen Netzen erfolgt das Erlernen einer Wertetafel durch langsame Veränderung der Synapsenleitwerte. Fasst man die Leitwerte als Merkmale der Netzstruktur auf, kann man von struktureller Speicherung sprechen. Es ist anzunehmen, dass das Gehirn, das natürliche Vorbild der künstlichen neuronalen Netze, ähnlich arbeitet. Das legt den Verdacht nahe, dass ein entscheidender Unterschied zwischen technischer und biologischer Informationsverarbeitung, zwischen Computer-IV und Human-IV im Speicherprinzip zu suchen ist. Es wird sich jedoch herausstellen, dass der Unterschied zwischen struktureller und adressierter Speicherung gar nicht so grundsätzlicher Art ist, wie es den Anschein haben könnte.

Als Zugang zu der nun zu besprechenden mikroelektronischen Technik des strukturellen Speicherns eignet sich das Problem des Umcodierens. Wir wissen bereits, dass sich die Funktionen eines Taschenrechners, der mit binär verschlüsselten Zahlen rechnet, in Form von Kombinationsschaltungen realisieren lassen. Wir wissen aber noch nicht, wie Dezimalzahlen in Dualzahlen bzw. wie Ziffern (bei ziffernweiser Codierung) in Bitketten umcodiert werden können. Ein Rechner mit dezimaler Ein- und Ausgabe muss über einen *Verschlüsseler* oder **Codierer** verfügen, der Ziffern zu Bitketten "verschlüsselt", d.h. in die computerinterne Darstellung *codiert*, sowie einen *Entschlüsseler* oder **Decodierer**, der Bitketten zu Ziffern "entschlüsselt", d.h. aus der internen in die externe Darstellung *decodiert*.

Wir wollen einen Codierer und einen Decodierer entwerfen. Der Codierer soll den Ziffern (Dezimalzahlen) 0 bis 9 die entsprechenden Dualzahlen zuordnen und der Decodierer soll den Dualzahlen (Bitketten, Binärwörtern) 0000 bis 1001 die jeweilige Dezimalzahl zuordnen. Der Codierer verfüge über 10 Tasten für die Ziffern*eingabe* (man denke an die Zifferntasten der Tastatur eines PC), und der Decodierer verfüge

über 10 Anzeigefelder für die Ziffern*ausgabe*, für jede Ziffer ein Feld. Der Codierer besitzt 4 Ausgabeleitungen und der Decodierer 4 Eingabeleitungen, für jede Stelle der Dualzahl (für jedes Bit des Binärwortes) eine Leitung.

Wir beginnen mit dem Decodierer und fragen, wie seine Schaltung aussehen könnte. Dazu vergegenwärtigen wir uns seine Aufgabe. Der Decodierer muss einem bestimmten Eingabewort eine bestimmte Leitung zuordnen, beispielsweise dem Eingabewort (der Dualzahl) 0101 diejenige Ausgabeleitung, die zum Anzeigefeld der 5 führt. Darum nennen wir eine solche Schaltung auch **Wort-Leitung-Zuordner**. Wer durch diese Bezeichnung an die Schaltung des Halbaddierers von Bild 9.4b erinnert wird, hat die Lösung unseres Problems vielleicht schon erraten (siehe [9.15]).

Bild 12.1a zeigt einen Wort-Leitung-Zuordner mit drei Ein- und drei Ausgabeleitungen. Über das Eingabeleitungstupel können Eingabewörter der Länge 3 Bit eingegeben werden. Jede Ausgabeleitung ist mit einem Anzeigefeld verbunden. Die Schaltung unterscheidet sich von dem Wort-Leitung-Zuordner in Bild 9.4b (dem linken Teil der dort dargestellten Schaltung bis einschließlich der AND-Glieder) in der Anzahl der Eingabeleitungen und darin, dass die NOT-Glieder, die sich ursprünglich vor den Eingängen der AND-Glieder befanden, an den Eingang der gesamten Schaltung vorgezogen sind, wodurch NOT-Glieder eingespart werden. Sie können ganz entfallen, wenn der Zuordner seine Eingaben von Ein-Bit-Speichern erhält, die zu jedem Wert auch dessen Negation liefern (vgl. Bild 9.7).

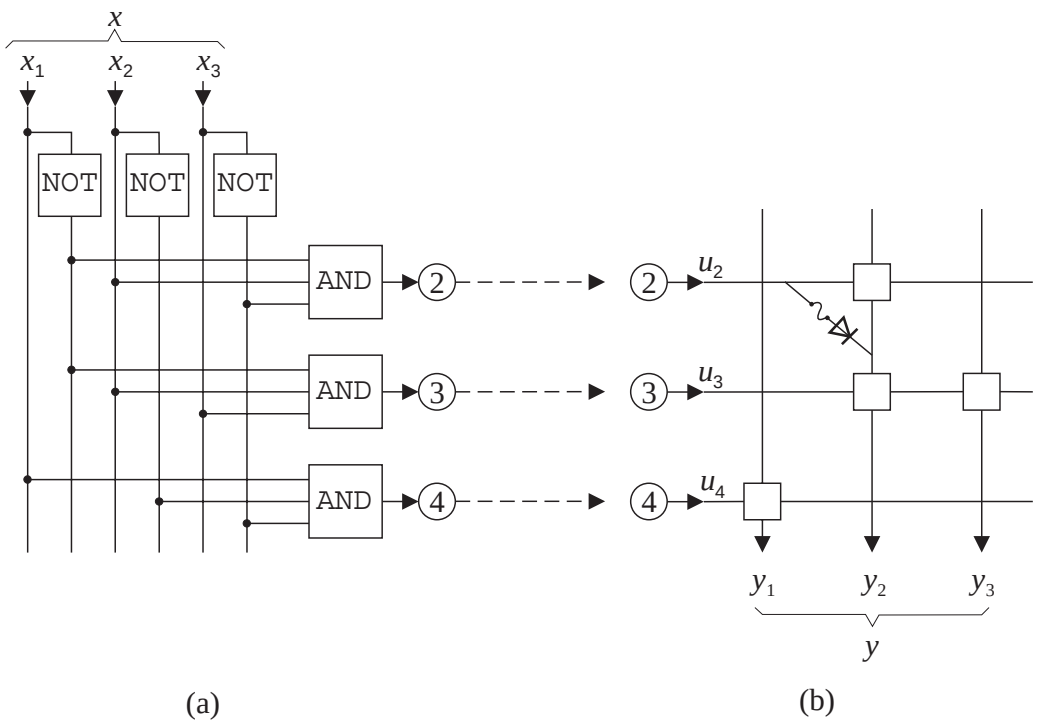


Bild 12.1 **(a)** - Wort-Leitung-Zuordner oder Adressierer; die Kreise stellen Anzeigefelder dar; **(b)** - Leitung-Wort-Zuordner; die Kreise stellen Eingabetasten dar.

Im Weiteren werden wir die durch die Ziffern 2, 3 und 4 gekennzeichneten waagerechten Leiter in Bild 12.1 als 2- bzw. 3- bzw. 4-Leiter bezeichnen[1]. Man verifiziert leicht, dass bei Eingabe des Wortes 010 bzw. 011 bzw. 100 der 2- bzw. 3- bzw. 4-Leiter eine 1 ausgibt, was die Anzeige der Ziffern 2 bzw. 3 bzw. 4 bewirkt. Wenn ein anderes Binärwort der Länge 3 eingegeben wird, führen alle Ausgabeleitungen eine 0. In diesem Sinne sagen wir, dass Bild 12.1a einen *partiellen* Wort-Leitung-Zuordner darstellt.

Ein Codierer hat die entgegengesetzte Aufgabe; er ordnet - verkürzt ausgedrückt - seinen Eingabeleitungen Binärwörter zu. Darum nennen wir eine solche Schaltung auch **Leitung-Wort-Zuordner**. Bild 12.1b zeigt einen Leitung-Wort-Zuordner mit drei Eingabeleitungen. Die Eingabe einer 1 auf den 2- bzw. 3- bzw. 4-Leiter bewirkt die Ausgabe des Wortes 010 bzw. 011 bzw 100. Um das zu erkennen, erinnere man sich an die Funktionsweise des *Schnittpunktoperators*. Sie ist in Kap 9.3 [9.14] erläutert worden, allerdings auf der booleschen Ebene, d.h. unter Bezugnahme auf Bild 9.4c. Die Erläuterung soll jetzt auf der technischen (elektronischen) Ebene wiederholt werden.

Die folgenden Überlegungen entsprechen denen von Kap.10, wo wir die booleschen Operatoren technisch als Schalternetze realisiert haben. Wir übernehmen die dortigen codierenden Zustandsparameter und ordnen der booleschen 0 den (niedrigen) Spannungswert U_0 und der 1 den (hohen) Spannungswert U_1 zu. Im Ruhezustand liege auf allen Leitungen die Spannung U_0. Bei Drücken einer Taste, z.B. der Taste 2 (in Bild 12.1b durch den Kreis mit der 2 im Eingabeleiter dargestellt), wird die Spannung am 2-Leiter von der Ruhespannung U_0 auf die Spannung U_1 angehoben. Die Spannung wird an den (senkrechten) y_2-Leiter weitergegeben. Die Übergabe erfolgt über eine Diode, sodass in der entgegengesetzten Richtung keine Spannungsweitergabe erfolgen kann. Jedes Quadrat in dem Leitergitter stellt eine solche 1
Diodenverbindung dar; an einem der Schnittpunkte ist sie eingezeichnet (die Bedeutung des geschlängelten Leitungstücks wird später erklärt). Das Gitter mit den eingebauten Dioden heißt **Diodenmatrix** und realisiert eine bestimmte Verbindungsstruktur. In der Praxis wird die Gleichrichterfunktion der Dioden häufig durch Transistoren realisiert, sodass sich eine *Transistorenmatrix* ergibt. Wenn im Weiteren von Diodenmatrizen die Rede ist, sind Transistorenmatrizen eingeschlossen.

Werden die Spannungen U_0 und U_1 als die Binärwerte 0 und 1 interpretiert, liefert die Matrix die genannte Verschlüsselung (Codierung). Davon überzeugt man sich durch "waagerechtes Lesen" der Matrix. Wenn z.B. der 3-Leiter eine 1 führt, übergibt er diese an den y_2- und y_3-Leiter, jedoch nicht an den y_1-Leiter. Es wird also das Binärwort 011 (die duale 3) ausgegeben. Man beachte, dass sich die den waagerechten Leitern zugeordneten Binärwörter unmittelbar aus der Matrix ablesen lassen,

---

1 Es sei daran erinnert, dass die Wörter *Leitung* und *Leiter* wie Synonyme verwendet werden, wenn von Leitungs- oder Leitermatrizen die Rede ist.

indem ein Leiterschnittpunkt ohne Quadrat als 0 und ein solcher mit Quadrat als 1 gelesen wird.

Man kann die Matrix auch "senkrecht lesen". Dann ergeben sich die Bedingungen dafür, dass die senkrechten Leitungen eine 1 führen, zum Beispiel: Der y_2-Leiter führt eine 1, wenn der 2-Leiter ODER der 3-Leiter eine 1 führt (oder beide). Ein senkrechter Leiter realisiert also - ebenso wie die senkrechten Leiter des Halbaddierers - eine Disjunktion, in welche diejenigen Eingabevariablen (in Bild 12.1b mit u_2, u_3 und u_4 bezeichnet) eingehen, deren Leiter mit dem betreffenden senkrechten Leiter über eine Diode verbunden sind. Beispielsweise ergibt sich für y_2 die Disjunktion

$$y_2 = u_2 \,\mathtt{OR}\, u_3. \tag{12.1}$$

Die Codierungsfunktion des Leitung-Wort-Zuordners von Bild 12.1b ist *strukturell* gespeichert. Die Matrix kann erweitert werden, sodass umfangreichere Funktionstafeln gespeichert werden können. Um sämtliche Ziffern zu verschlüsseln (zu codieren), muss das Gitter mindestens 4, um Ziffern und Buchstaben zu codieren muss es mindestens 6 senkrechte Leiter besitzen. Damit ein Leitung-Wort-Zuordner als Codierer dienen kann, muss die Zuordnung eineindeutig (in beiden Richtungen eindeutig) sein. Diese Bedingung ist für die Matrixschaltung von Bild 9.4b, die ebenfalls einen Leitung-Wort-Zuordner darstellt, *nicht* erfüllt, denn der mittlere und der untere Leiter liefern die gleiche Bitkette.

Wir verbinden jetzt die Ausgabeleiter des Wort-Leitung-Zuordners mit den Eingabeleitern des Leitung-Wort-Zuordners über die gestrichelt gezeichneten Leiter. Das scheint wenig Sinn zu haben, denn die Decodierung hebt die vorangegangene Codierung wieder auf. Die Ausgabewörter sind mit den Eingabewörtern identisch. Die Schaltung berechnet die Identitätsfunktion, aber nur partiell, und zwar für die Argumentwerte 010, 011 und 100. Wird ein anderes Binärwort eingegeben, also 000, 001, 101 oder 111, liefert die Schaltung stets das Ausgabewort 000. Damit der Operator die *totale* (vollständige) Identitätsfunktion für Binärworte der Länge 3 berechnet, müsste die Anzahl der wagerechten Leiter auf 7 erhöht werden (ein Leiter für die Eingabe 000 kann entfallen, Taktung mittels Toren vorausgesetzt).

Der Leser hat den Zweck der soeben durchgeführten "sinnlosen" Verbindung von der linken zur rechten Schaltung in Bild 12.1 vielleicht schon erraten. Wenn nämlich die rechte Schaltung *nicht* die Umkehroperation der linken ausführt, sondern irgendeine andere Zuordnung, dann berechnet der Kompositoperator eine nichtidentische Binärwortfunktion, deren Wertetafel unmittelbar der Struktur der Matrixschaltung entnommen werden kann. Ein nächstes Aha-Erlebnis: Wir haben eine sehr durchsichtige Technologie zur Herstellung von Kombinationsschaltungen nacherfunden. Aber erst eine weitere Idee hat für die Computerherstellung den durchschlagenden Erfolg gebracht.

In Kap.12.2 werden wir erfahren, dass sich Matrizenstrukturen sehr kostengünstig herstellen lassen. Daraus ergibt sich der Wunsch der Hardwareproduzenten, nicht nur Leitung-Wort-Zuordner, sondern auch Wort-Leitung-Zuordner als Matrizen

aufzubauen. Die Idee, wie sich dieser Wunsch verwirklichen lässt, liegt gar nicht so fern, wenn man sich klar macht, dass zu diesem Zweck AND- in OR-Glieder überführt werden müssen, und dass diese Überführung nach der *morganschen Regel* möglich ist (siehe Bild 9.1 und Formel (9.6b). Für einen 3-stelligen AND-Operator lautet sie in verkürzter Notation (die Konjunktionssymbole sind unterdrückt):

$$x_1x_2x_3 = \texttt{NOT}(\texttt{NOT}x_1\ \texttt{OR}\ \texttt{NOT}x_2\ \texttt{OR}\ \texttt{NOT}x_3). \tag{12.2}$$

Wir wollen zeigen, dass durch Anwendung dieser Formel die Schaltung von Bild 12.a in die linke, mit ADR bezeichnete Leitermatrix von Bild 12.2 überführt wird.

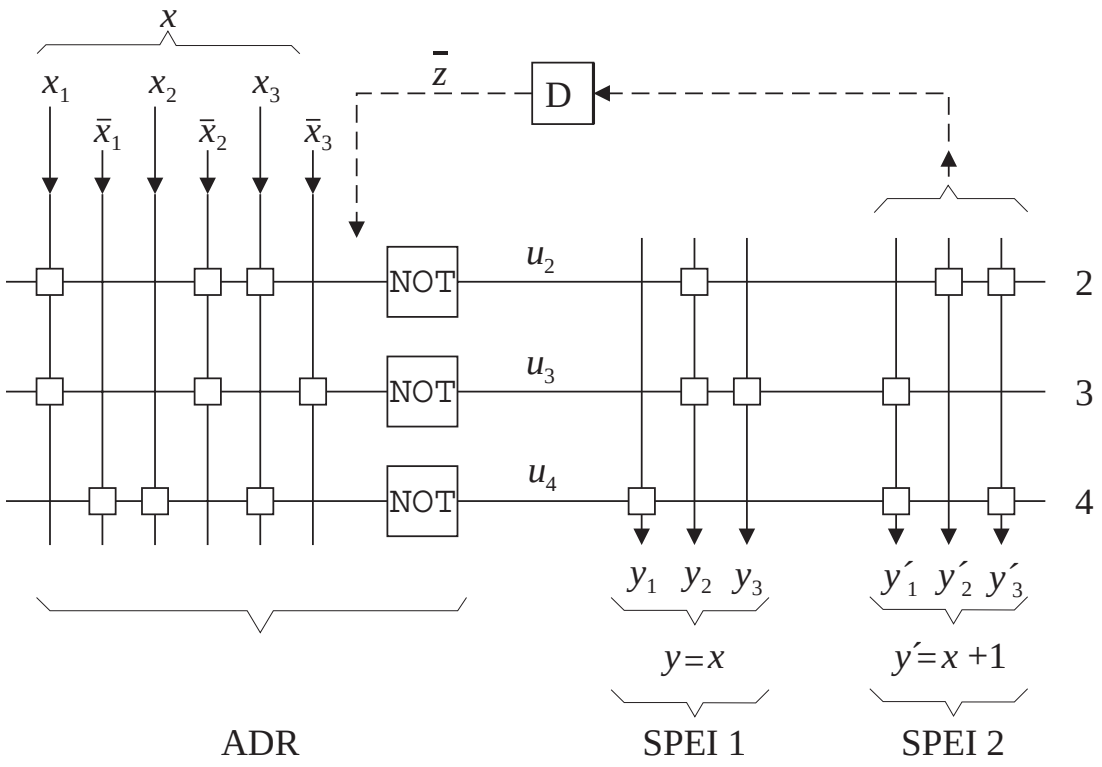


Bild 12.2 Codeumsetzer bzw. Festwertspeicher. ADR - Adressiermatrix; SPEI - Speichermatrix.

Bild 12.2 zeigt eine Schaltung aus drei hintereinandergeschalteten Diodenmatrizen, die mit ADR, SPEI1 und SPEI2 bezeichnet sind. ADR steht für Adressiermatrix, SPEI für Speichermatrix. Die Berechtigung dieser Bezeichnungen wird sich etwas später herausstellen. Die Matrizen besitzen gemeinsame waagerechte Leiter. Der Bitwert auf dem 2- bzw. 3- bzw. 4-Leiter sind mit u_2 bzw. u_3 bzw. u_4 bezeichnet. Die ADR-Matrix (die NOT-Glieder eingeschlossen) ergibt sich aus dem Wort-Leitung-Zuordner von Bild 12.1a bei Anwendung der morganschen Regel (12.2) auf jeden der AND-Operatoren. Dadurch werden die 3-stelligen Konjunktionen in negierte

3-stellige Disjunktionen überführt. Beispielsweise gilt für u_3 gemäß Bild 12.1a $u_3 = (\texttt{NOT}x_1)x_2x_3$. Daraus wird bei Anwendung von (12.2)

$$u_3 = \texttt{NOT}(x_1 \texttt{ OR } \texttt{NOT}x_2 \texttt{ OR } \texttt{NOT}x_3). \tag{12.3}$$

Diese Disjunktion ohne den NOT-Operator vor der Klammer lässt sich, wie wir wissen, als Leiter (diesmal als waagerechter Leiter) mit entsprechend platzierten Dioden (Schnittpunktoperatoren) realisieren, wie es in der ADR-Matrix von Bild 12.2 dargestellt ist. Auf die gleiche Weise ergeben sich die Platzierungen der übrigen Schnittpunktoperatoren der ADR-Matrix. Die bisher nicht berücksichtigten NOT-Operatoren sind in die Ausgabeleitungen der ADR-Matrix (des Wort-Leitung-Zuordners) eingefügt.

Wie man unschwer erkennt, führen bei jeder Eingabe (010, 110 oder 100) zwei der drei waagerechten Leiter vor dem NOT-Operator eine 1. Nach dem NOT-Operator führt eine einzige Leitung eine 1 und zwar diejenige, die auch in Bild 12.1a bei dem gleichen Eingabewort eine 1 führt. Da die SPEI1-Matrix mit der Matrix von Bild 12.1b identisch ist, liefert die Schaltung von Bild 12.2 (ohne die SPEI2-Matrix) das gleiche Ausgabetripel $y = x$ wie die gesamte Schaltung von Bild 12.1. Durch andere Platzierung der Diodenverbindungen können andere Zuordnungen "*eingelötet*" oder "*eingeprägt*" werden. Beispielsweise liefert die SPEI2-Matrix das Ausgabetripel $y' = x+1$, wobei die Tripel x und y' als Dualzahlen zu interpretieren sind. Sieht man von der Interpretation ab, kann die Operation, welche die ADR- und SPEI-Matrix gemeinsam ausführen, als *Umcodierung* aufgefasst werden. Aus diesem Grunde werden derartige Schaltungen **Codeumsetzer** genannt.

Obwohl die Bezeichnungen ADR und SPEI bisher nicht erklärt worden sind, haben sie beim Leser vielleicht ein weiteres Aha-Erlebnis ausgelöst. Die waagerechten Leiter in Bild 12.2 können als Speicherplätze aufgefasst werden. Durch das Eingabewort wird ein bestimmter Platz angewählt (*adressiert*). In diesem Sinne kann der Wort-Leitung-Zuordner als *Adressiermatrix* (ADR) aufgefasst werden. Die Eingabe einer Adresse x bewirkt die Ausgabe des "Inhalts" y der angewählten Speicherzelle, sodass der Leitung-Wort-Zuordner als *Speichermatrix* (SPEI) bezeichnet werden kann. Den Inhalt einer Speicherzelle kann man aus der SPEI-Matrix ablesen, indem man die Quadrate (die Symbole der Schnittpunktoperatoren) auf dem betreffenden waagerechten Leiter als Einsen und die übrigen Gitterpunkte als Nullen liest. Danach ist auf dem 3-Leiter das Wort 011 (die Ziffer 3) gespeichert. Der Inhalt eines Speicherplatzes ist durch die "eingelöteten" Dioden der Speichermatrix ein für allemal "strukturell" festgelegt; er kann nur ein einziges mal eingespeichert werden. Mit anderen Worten, die gesamte Schaltung (Adressier- und Speichermatrix) stellt einen **Nur-Lese-Speicher**, einen sogenannten **ROM** (**Read Only Memory**) dar, genauer einen **elektronischen ROM**.

Damit ist gezeigt, dass die Unterscheidung zwischen struktureller und adressierter Speicherung nur bedingte Bedeutung hat. Auch aus der Sicht der Anwendungsmöglichkeiten ist sie unwesentlich. In dieser Hinsicht ist die Unterscheidung zwischen

ROM und RAM wichtiger. **RAM** (**Random Access Memory**) ist die Kurzbezeichnung für **Schreib-Lese-Speicher**, das heißt für Speicher, auf deren Speicherzellen sowohl zum Zwecke des Einschreibens als auch des Auslesens zugegriffen werden kann. Sie werden auch als **Direktzugriffsspeicher** oder **Speicher mit wahlfreiem Zugriff** bezeichnet. In Kap.13.2 werden wir den Aufbau elektronischer RAM nacherfinden.

Zur ROM-Funktion hatte uns "waagerechtes Lesen" der Matrixschaltungen geführt. Waagerechtes Lesen allein der ADR-Matrix und ebenso "Senkrechtes Lesen" der Speichermatrix führt jeweils zu einer booleschen Funktion in Form einer Disjunktion, beispielsweise zur Disjunktion in (12.1), wenn man den y_2-Leiter "senkrecht liest". Ersetzt man in (12.1) die Variablen u_2 und u_3 durch die entsprechenden Konjunktionen, ergeben sich für die Ausgabevariablen boolesche Funktionen in der disjunktiven Normalform (DNF), beispielsweise für y_2

$$y_2 = (\texttt{NOT}\,x_1)x_2(\texttt{NOT}\,x_3)\ \texttt{OR}\ (\texttt{NOT}\,x_1)x_2x_3. \tag{12.4}$$

Wegen der Vorrangregeln (`NOT` vor `AND` vor `OR`) können die Klammern entfallen. Für die übrigen y-Leiter lassen sich entsprechende Ausdrücke angeben. Jeder Leiter liefert den Wert einer booleschen Funktion von drei Variablen. Die gesamte Schaltung stellt eine Kombinationsschaltung mit je drei Eingabe- und Ausgabeleitungen dar, m.a.W. einen Operator, dessen Ein- und Ausgabeoperanden dreistellige Bitketten sind. Wir sind zu einem Ergebnis mit großem praktischen Wert gelangt. Wir haben eine sehr durchsichtige Methode nacherfunden, Kombinationsschaltungen und damit auch elektronische ROM-Speicher mittels Diodenmatrizen zu realisieren.

Es gibt andere Möglichkeiten, reine Lesespeicher herzustellen. Es sei nur an das in Kap.11.2 erwähnte optische Speicherprinzip erinnert, auf dem der sog. CD-ROM basiert. Doch ist ein CD-ROM mit den Nachteilen des bewegten Trägers behaftet. Der rein elektronische ROM hat diesen Nachteil nicht. Dafür ist seine Speicherkapazität niedriger. Doch nimmt sie ständig zu, denn infolge der Perfektionierung der Herstellungstechnologie lassen sich immer größere Matrixschaltungen herstellen. Gleichzeitig sinkt der Preis pro Diode bzw. Transistor und damit pro Speicherplatz. Wir wenden uns nun dieser Technologie zu.

12.2 Herstellung von Diodenmatrizen

Wir beginnen mit einer Aufwandsabschätzung und fragen, wie viele waagerechte und senkrechte Leiter ein ROM etwa enthalten muss, wenn in ihm ein Deutsch-Englisches Wörterbuch mit 1000 Wörtern abgespeichert werden soll. Jedes Wort werde in einer gesonderten Speicherzelle, d.h. auf einem waagerechten Leiter abgespeichert. Der ROM enthält also 1000 waagerechte Leiter. Die Wortlänge sei in jeder Sprache auf 20 Buchstaben, d.h. auf 100 Bit begrenzt (5 Bit pro Buchstabe). Die Adressiermatrix enthält also 200 und die Speichermatrix 100 senkrechte Leiter.

Damit ergeben sich 300*1000 = $3*10^5$ Schnittpunkte, die jedoch bei weitem nicht alle mit Dioden zu besetzen sind.

Die Mikroelektronik ermöglicht die Implementierung eines solchen Wörterbuches auf einem einzigen Chip, die Negatoren in den waagerechten Leitern eingeschlossen. Das Wörterbuch darf auch durchaus noch umfangreicher sein, und es können zusätzliche Schaltungen integriert werden, z.B. ein Eingabecodierer (zur Umcodierung der Buchstaben in Bitketten) und ein Ausgabedecodierer.

Das Wörterbuch steht hier als Beispiel für die unterschiedlichsten Einsatzmöglichkeiten in Industrie, Wirtschaft, Büro und Haushalt. In Kap.9.2 [9.7] war bereits auf die vielfältigen Verwendungsmöglichkeiten von Kombinationsschaltungen in Auskunftssystemen hingewiesen worden. Dadurch wird verständlich, welchen Druck der Markt auf die Entwicklung kostengünstiger Technologien zur Herstellung derartiger integrierter Schaltungen ausübt. Die Ergebnisse sollen in aller Kürze dargestellt werden.

Abgesehen von der Perfektionierung der Bearbeitung des Halbleitermaterials (Chipherstellung, Beschichtung, Maskentechnik, Ätzung, Diffusion u.s.w.) brachte folgende Idee eine erhebliche Kostensenkung. Man stelle zunächst eine vollbesetzte Matrix her, die an jedem Schnittpunkt eine Diode enthält, und erzeuge nachträglich die gewünschte Struktur durch Stillegung der überflüssigen Dioden. Da es für die Herstellungskosten relativ unwesentlich ist, wieviele Bauelemente eine Matrix enthält und andrerseits die Stillegung einfach und billig zu bewerkstelligen ist, ergibt sich die Möglichkeit einer äußert profitablen Massenproduktion bei gesichertem Absatz.

Das Stilllegen (*Ab*schalten) ausgewählter Bauelemente kann z.B. durch Wegätzen von Leitern mittels der Maskentechnik erfolgen oder durch Wegschmelzen durch geeignet dosierte Ströme. In Bild 12.1b ist das zu schmelzende Leiterstück geschlängelt (als Schmelzsicherung) dargestellt. Auch der umgekehrte Weg ist möglich. Es wird eine vollbesetzte Matrix “stillgelegter” Transistoren hergestellt, die in *beiden* Richtungen als *Nichtleiter* wirken. Durch eine geeignete Spannung kann ein innerer Durchschlag hervorgerufen werden, wodurch der Transistor in eine funktionsfähige Diode überführt wird (*an*geschaltet wird).

Das Strukturieren - man sagt auch Prägen oder Programmieren - der Matrix nach dem Durchschmelz- bzw. Durchschlagverfahren erfolgt durch Anlegen einer vorgeschriebenen Spannung an diejenigen Matrixelemente, die ab- bzw. angeschaltet werden sollen. Das Programmieren kann vom Produzenten, aber auch vom Nutzer vorgenommen werden. Man spricht dann von einem **programmierbaren ROM**, abgekürzt: **PROM**.

Nach den genannten Methoden lässt sich ein PROM nur ein einziges mal programmieren. Der Wunsch nach wiederholter Programmierung führte zur Entwicklung des **löschbaren PROM**, abgekürzt: **EPROM** (von erasable PROM). Eine gängige Variante des EPROM verwendet spezielle Transistoren, die sogenannten *Floating-Gate-MOS-FET*. Sie ersetzten die Schmelzsicherung in Bild 12.1b.

Der MOS-FET war in Kap.11.2 [11.3] beschrieben worden. Der Zusatz “Floating-Gate” zeigt an, dass die steuernde Elektrode entlang des Leitungskanals keinen äußeren Kontakt besitzt, sondern sich innerhalb der isolierenden Schicht befindet. Auf diesem isolierten Leiterstück, dem “floating gate”, kann durch Anlegen einer ausreichend hohen Spannung eine Ladung induziert werden (Durchtunnelung der isolierenden Schicht). Ist die Ladung so groß, dass sie alle freien Ladungsträger aus dem Kanal verdrängt, ist die Leitung unterbrochen, was einem geöffneten Schalter oder einer durchgeschmolzenen Sicherung entspricht. Der Isolationswiderstand des isolierenden Materials ist so hoch, dass die Ladung über Jahre bestehen bleibt. Doch kann er durch ultraviolettes Licht so stark herabgesetzt werden, dass die Ladung im Laufe von Minuten abfließt. Auf diese Weise ist es möglich, einen EPROM durch Bestrahlen mit UV-Licht durch ein Fenster in der Verkapselung zu löschen.

Ein Codeumsetzer wird unterschiedlich bezeichnet je nachdem, welche Matrix vom Nutzer programmierbar ist. In einem **ROM**-Baustein ist *keine* Matrix programmierbar; in einem **PROM**- und einem **EPROM**-Baustein ist die Speichermatrix programmierbar; in einem **PAL**-Baustein (von Programmable Array Logic) ist die Adressiermatrix programmierbar und in einem **PLA**-Baustein (von Programmable Logic Array) sind beide Matrizen programmierbar.

12.3 Anwendungen von Diodenmatrizen

12.3.1 Elementare Hardware-Software-Schnittstelle

Wir kehren noch einmal zu dem Begriffspaar *Wort-Leitung* zurück und unterziehen es einer kritischen Analyse. In Kap.12.1 hatten wir uns Schaltungen für den Decodierer und den Codierer überlegt und sie *Wort-Leitung*-Zuordner bzw. *Leitung-Wort*-Zuordner genannt, ohne daran Anstoß zu nehmen, das die Begriffe “Leiter” und “Wort” verschiedenen Bereichen, verschiedenen Ebenen des sprachlichen Modellierens angehören. Leiter sind Elemente der Hardware, Wörter sind Elemente der Software. Diese Ungereimtheit wird noch auffälliger, wenn man “Wort” durch “Adresse” oder durch “Name” ersetzt. Mit “Name” ist gemeint, dass ein Leiter durch ein Binärwort “*benannt*” wird. Tatsächlich zeigt die “Ungereimtheit” an, dass sich in den beiden Schaltungen Hardware und Software treffen, mit anderen Worten: ***Wort-Leitung-Zuordner*** *und* ***Leitung-Wort-Zuordner*** *(Decodierer und Codierer) stellen* ***Schnittstellen*** *zwischen Hardware und Software auf einer sehr elementaren Ebene dar*. Diese Feststellung ist unüblich, trifft aber den Kern, wenn man davon ausgeht, dass die Hardware mit Bauelementen und elektrischen Größen, die Software dagegen mit Zeichen zu tun hat, oder allgemeiner, dass die Hardware mit Elektronik und die Software mit Sprache zu tun hat.

Tatsächlich ist die Trennung von Hardware und Software eine Frage des Betrachtungsstandpunktes. Das wird deutlich, wenn man bedenkt, dass die Hardware als *Träger* und die Software als *Beschreibungsmittel* informationeller Prozesse dient und

dass jedem Beschreibungselement Merkmalswerte von Trägerelementen entsprechen, wobei die Entsprechung sehr kompliziert sein kann. Aber wie undurchsichtig sie auch sein mag, letzen Endes wird sie durch Wort-Leitung-Zuordner und Leitung-Wort-Zuordner (Decodierer und Codierer) bewerkstelligt.

Ob man ein informationelles System mehr aus der Sicht des Elektronikers oder der des Programmierers betrachtet, hängt davon ab, wieweit man den *sprachlichen Überbau* der elektronischen Schaltungen bzw. den *elektronischen Unterbau* der sprachlichen Ausdrücke im Auge hat. Je höher man in der Softwarehierarchie aufsteigt, umso mehr abstrahiert man zwangsläufig vom Hardware-Unterbau.

12.3.2 Multiplexer, Demultiplexer und Bus

Mit Hilfe der Adressiermatrix (des Decodierers, des Wort-Leitung-Zuordners) lassen sich Mehrfachweichen aufbauen, das sind Weichen mit mehreren Ein- bzw. Ausgängen. Wie wir wissen, können Weichen aus Toren komponiert und Tore als AND-Glieder verwirklicht werden. Die Bilder 12.3a und b zeigen die Realisierung der einfachen Sammel- bzw. Zweigeweiche mittels AND-Gliedern. Die Sammelweiche gibt in dem Moment, in dem an einer der beiden Steuerleitungen ein Steuerimpuls (d.h. kurzzeitig der Spannungswert, der dem booleschen Wert 1 entspricht) erscheint, den Wert, der auf der durch das Steuersignal "angewählten" Eingabeleitung liegt (der Spannungswert bzw. der ihm entsprechende boolesche Wert) an die gemeinsame

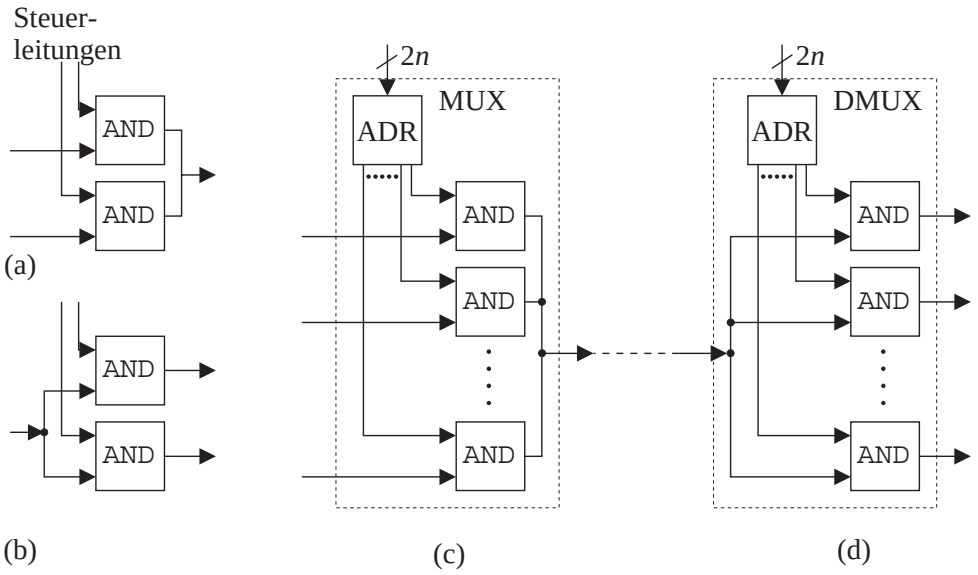


Bild 12.3 Weichen; **(a)** - einfache Sammelweiche; **(b)** - einfache Zweigeweiche; **(c)** - Multiplexer (MUX); **(d)** - Demultiplexer (DMUX)

Ausgabeleitung weiter. Im Falle der Zweigeweiche wird der Wert der gemeinsamen Eingabeleitung an die "angewählte" Ausgabeleitung weitergegeben.

Nachdem wir die Adressiermatrix (im Weiteren ADR-Baustein genannt und mit ADR bezeichnet) nacherfunden haben, fällt es nicht schwer die Schaltungen der Einfachweichen auf Mehrfachweichen zu erweitern. Wenn z.B. eine Zweigeweiche mit *m* Ausgängen gesteuert werden soll, muss einer von *m* Leitern "angewählt" werden, wofür ein ADR-Baustein eingesetzt werden kann. Bild 12.3c bzw. d zeigt die Verwirklichung dieser Idee für eine mehrfache Sammel- bzw. Zweigeweiche. Die Schaltungen heißen **Multiplexer** (abgekürzt **MUX**) bzw. **Demultiplexer** (**DMUX**).

Über den Steuereingang (Eingang des ADR-Bausteins) wird ein **Steuerwort** eingegeben, das den anzuwählenden Leiter "öffnet". Das Steuerwort kann als *Name* oder *Adresse* des Leiters aufgefasst werden. Der kurze, schräge Querstrich durch die Steuerleitung mit der Angabe "2*n*" bedeutet, dass es sich um ein Leitungsbündel aus 2*n* Einzelleitungen handelt. Der Faktor 2 zeigt an, dass sowohl die Adresse als auch deren Negation eingegeben wird. Da sich mit einer *n*-stelligen Adresse 2^n Datenleitungen adressieren lassen, ist bei vorgegebener Anzahl *m* von Datenleitungen eine Adresslänge $n \geq \mathrm{ld} m$ erforderlich. Multiplexer und Demultiplexer können als **Kompositflussknoten** aufgefasst werden, denn sie lassen sich, wie man leicht erkennt, aus den elementaren Flussknoten von Bild 8.2 komponieren.

Wir verbinden jetzt die Ausgabeleitung des Multiplexers mit der Eingabeleitung des Demultiplexers (in Bild 12.3 durch die gestrichelte Linie angedeutet). Es entsteht eine Konfiguration von Leitungen, über welche Verbindungen zwischen verschiedenen *Sendern* (links) und *Empfängern* (rechts) hergestellt werden können. Die Leitungskonfiguration stellt eine **einkanalige**, **gerichtete Kommunikationsstruktur** dar. *Als **Kommunikationsstruktur** bezeichnen wir jede Konfiguration von Verbindungen, über welche verschiedene "Teilnehmer" kommunizieren können.* Die vorliegende Kommunikationsstruktur heißt gerichtet, weil Verbindungen nur in einer Richtung (von links nach rechts) möglich sind. Sie heißt *einkanalig*, weil jeweils nur eine einzige Verbindung hergestellt werden kann.

Die Wörter "Nachricht" und "Teilnehmer" hat der Leser sicher richtig interpretiert, obwohl sie etwas unvermittelt verwendet worden sind. Sie zeigen an, dass wir in ein anderes Gebiet übergewechselt sind, in die Kommunikationstechnik. Das ist kein Zufall, denn die Datenübergabe in einem Rechner einerseits und die Nachrichtenübertragung in Kommunikationsnetzen andrerseits stellen die gleichen Probleme, und diese werden nach weitgehend gleichen Prinzipien gelöst. Das gilt auch für die Datenübertragung zwischen Computern, die zu einem sog. **Rechnernetz** miteinander verbunden sind. Demzufolge ist es durchaus möglich und sogar üblich, für die Datenübertragung in Rechnernetzen vorhandene *Tele*kommunikationsnetze zu benutzen.

Die Entwicklung ist soweit fortgeschritten, dass sich kaum noch eine Grenze zwischen Telekommunikationsnetz und Rechnernetz ziehen lässt, denn die "Endstationen", die sog. *Terminals*, d.h. die Geräte, derer sich die Abonnenten eines Kommunkationsnetzes bedienen (den Telefonapparat eingeschlossen) enthalten in

zunehmendem Maße kleinere oder größere Computer. Ein weltumspannendes Kommunikations- und Rechnernetz nimmt immer mehr Gestalt an[2]. Die Menschheit ist dabei, das "Nervensystem" der Informationsgesellschaft aufzubauen. Die inzwischen allgemein bekannte Abkürzung **WWW** für **World Wide Web** ("weltweites Gewebe") ist Nomen und Omen.

Nach dieser Abschweifung kehren wir zu den *gerichteten* Kommunikationsstrukturen der Bilder 12.3c und d zurück. Um Verbindungen in *beiden* Richtungen herstellen zu können, muss an jedem Ende ein MUX-DMUX-Paar vorhanden sein. Ein solches Paar nennen wir **Halbkommutator** (abgekürzt HK). Die sich ergebende Kommunikationsstruktur ist in Bild 12.4a gezeigt, wobei nicht nur die Steuerleitungen, sondern auch die Datenleitungen zu Bündeln zusammengefasst sind. Durch Steuerung der vier als kleine Kreise dargestellten steuerbaren Flussknoten wird die Richtung der Datenübertragung bestimmt.

Bild 12.4b zeigt die Kommunikationsstruktur von Bild 12.4a, wobei ein Halbkommutator als (etwas größerer) Kreis dargestellt ist, der eine **ungerichtete Mehrfachweiche** symbolisiert, die durch eine - evtl. recht lange - Bitkette (Adresse) gesteuert wird. Die Quadrate symbolisieren die kommunizierenden Teilnehmer. Der Begriff des Teilnehmers ist in einem verallgemeinerten Sinne zu verstehen. Ein

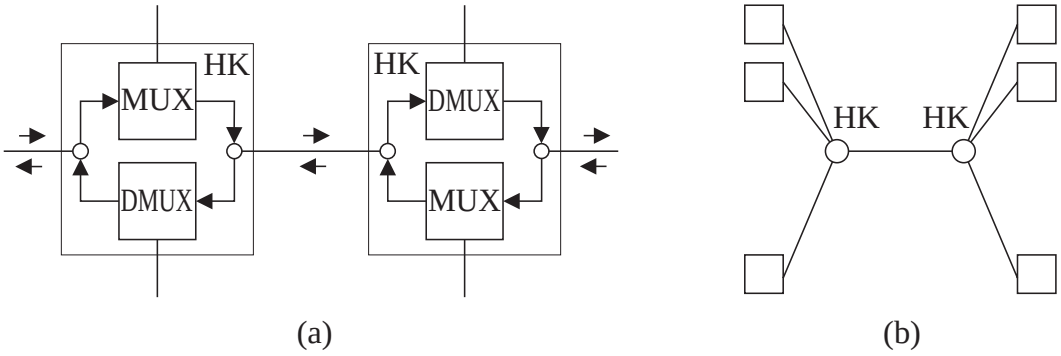


Bild 12.4 Verbindung über Halbkommutatoren (HK)

"Teilnehmer" kann ein Fernsprechteilnehmer sein, er kann ein Computer sein (im Falle eines Rechnernetzes), er kann ein Speicher oder auch ein Speicherplatz sein, wie wir gleich sehen werden.

Geht man davon aus, dass die linke Teilnehmermenge in Bild 12.4b mit der rechten identisch ist, und vereinigt man die Halbkommutatoren zu einem einzigen Knoten, entsteht eine sternförmige Kommunikationsstruktur, die wir **Vollkommutator** nennen (präziser müssten wir Einkanal-Vollkommutator sagen). Über ihn kann jedes beliebige Teilnehmerpaar kommunizieren, aber jeweils nur ein einziges Paar.

2 Näheres siehe z.B. in [Tanenbaum 98],[Lockemann 93].

In konkreten Anwendungsfällen ist es oft nicht erforderlich, dass jedes Paar verbunden werden kann. Aus diesem Grunde führen wir einen verallgemeinerten Kommutatorbegriff ein und vereinbaren:

*Ein **Kommutator** ist eine sternförmige Kommunkationsstruktur, über die zwischen* 2
jeweils zwei Teilnehmern Nachrichten übertragen werden können, nicht unbedingt zwischen allen möglichen Paaren und nicht unbedingt in beiden Richtungen. Wenn im Weiteren von Kommutator die Rede sein wird, ist stets ein *einkanaliger* Kommutator gemeint.

Der Begriff des Kommutators und seine Darstellung als Stern kann bei der Analyse und Synthese komplizierter Kommunikationssysteme zunächst sehr hilfreich sein, selbst dann, wenn man den Kommutator (die *Kompositweiche*) nachträglich in *Bausteinweichen* dekomponiert und die Bausteine in Richtung Peripherie verlegt, wodurch eine verästelte Verbindungsstruktur entsteht. Sie ist für Telefon- und Rechnernetze typisch, aber auch für interne Verbindungsstruktur eines Computers. Vorgreifend sei angemerkt, dass die Kommunikationsstruktur von Bild 12.4b derjenigen sehr ähnlich ist, über die der Prozessor eines Prozessorcomputers mit seinem Arbeitsspeicher kommuniziert (siehe Bild 13.7). Dort ist der rechte Halbkommutator durch einen Kommutator ersetzt. Die Quadrate des linken Halbkommutators in Bild 12.4 entsprechen in Bild 13.7 den Speicherplätzen des Arbeitsspeichers. Die Speicherplätze spielen also die Rolle der “Teilnehmer”. Größere Computer bestehen aus vielen Einheiten, die untereinander Daten austauschen. Der Austausch kann streckenweise über *gemeinsame* Leitungen gemäß Bild 12.4b erfolgen. Ein solches Verbindungssystem innerhalb eines Computers heißt **Bus**. Ein Bus arbeitet i.d.R. *einkanalig*, d.h. er kann jeweils nur einem einzigen Teilnehmner zur Verfügung gestellt (zugeteilt) werden und diesen mit dem gewünschten Adressaten verbinden. Das Gleiche gilt für Telekommunikationsnetze, auch für Rechnernetze. Ein *Kommunikationskanal* kann jeweils nur einem einzigen Teilnehmer zugeteilt werden. Die Zuteilung kann zentral durch einen speziellen Steueroperator oder auch dezentral erfolgen, indem z.B. der Teilnehmer selber feststellt, ob der Kanal (die Leitung, der Bus) frei oder besetzt ist.

Im Sinne einer Dekomposition des Halbkommutators führen wir nun gedanklich folgende Operation aus. Wir zerschneiden zeilenweise die Diodenmatrizen (die Adressiermatrizen des Multiplexers und Demultiplexers) und verteilen die Zeilen an die zugeordneten Teilnehmer (Adressaten). Jeder Teilnehmer erhält eine MUX- und eine DMUX-Zeile, über welche die Verbindung zum bzw. vom Bus hergestellt werden kann. Die Vorgehensweise wird oft als **Schlüssel-Schloss-Prinzip** bezeich- 3
net, Das hat folgenden Grund.

In Kap.12.1 hatten wir die Funktionsweise des Decodierers und des Codierers bzw. der Adressier- und der Speicher-Matrix unter Verwendung der Begriffspaares *Wort-Leitung* bzw. *Speicheradresse-Speicherinhalt* beschrieben. Jetzt wollen wir die Funktionsweise der Adressiermatrix (des Decodierers) unter Verwendung des Be-

griffspaares *Schlüssel-Schloss* beschreiben und greifen damit auf den Beginn des Kapitels 12.1 zurück, wo wir den Decodierer *Entschlüsseler* genannt hatten.

Die Funktionsweise eines Entschlüsselers (z.B. des Decodierers von Bild 12.1a) lässt sich folgendermaßsen beschreiben. Der Ent*schlüssel*er vergleicht das jeweilige Eingabewort (den *Schlüssel*) mit den in der Decodiermatrix (Adressiermatrix) strukturell gespeicherten Wörtern (*Schlössern*). Wenn er Übereinstimmung erkennt, m.a.W. wenn der Schlüssel zum Schloss "passt", belegt er den betreffenden Leiter mit einer 1 und öffnet damit das Schloss, d.h. den Zugang zu dem betreffenden Speicherplatz. An die Stelle des Speicherplatzes tritt jetzt ein Teilnehmer, ein "Adressat", und zwar derjenige, der mit dem zum Schlüssel passenden Schloss versehen ist.

Damit ist es möglich, Nachrichten über eine einzige Leitung, den **Datenbus**, sozusagen "an alle" zu versenden und zwar zusammen mit dem jeweiligen Schlüssel, der *Adresse* des Teilnehmers (des Adressaten). Das erfolgt im Bussystem eines Computers i.d.R. über einen speziellen **Adressbus**. Außerdem müssen die Weichen in Bild 12.4a auf "*Senden*" beziehungsweise, wenn ein Teilnehmer eine Nachricht abstezen will, auf "*Empfangen*" gestellt werden. Dafür wird häufig ein spezieller **Steuerbus** eingerichtet.

Wenn nur eine einzige Leitung zur Verfügung steht (wie z.B. in einem Telefonnetz oder in einem Rechnernetz, dessen Teilnehmer über ein Telefonnetz kommunizieren), muss der Schlüssel gemeinsam mit der Nachricht versendet werden, ähnlich wie die Adresse auf einem Briefumschlag zusammen mit dem Brief versendet wird.

Das Schlüssel-Schloss-Prinzip ähnelt dem Adressierungsmechanismus für Steuerinformationen in lebenden Organismen. Schlüssel und Schloss sind hier molekulare Strukturen (Enzyme, Hormone, Transmitter), die "zusammenpassen" müssen, damit an dem betreffenden Ort ein bestimmter Steuereffekt ausgelöst wird. So wird beispielsweise der Stoffwechsel (Austausch von Baustoffen) im lebenden Organismus gesteuert. Im Gegensatz zum Bus kann sich eine riesige Anzahl von Bauelementen (Eiweißverbindungen) und Steuerenzymen gleichzeitig durch das "Kommunikationssystem" bewegen; man hat es also nicht mit einem *Ein*kanal-, sondern mit einem *Mehr*kanalkommutator zu tun.

Der Mehrkanalkommutator ist auch für die technische Nachrichtenübertragung oft eine unabdingbare Notwendigkeit. Eine naheliegende, aber aufwendige Lösung ist der **Kreuzschienenverteiler** (Bild 12.5). Er enthält pro Teilnehmer je einen senkrechten und einen waagerechten Leiter. In jedem Schnittpunkt (die Schnittpunkte auf der Diagonalen ausgenommen) ist ein Schnittpunktoperator angeordnet, über den eine gerichtete Verbindung nach unten oder nach rechts hergestellt werden kann. Auf diese Weise kann gleichzeitig eine größere, aber doch begrenzte Anzahl von Verbindungen hergestellt werden; man spricht dann von einem **Mehrkanalkommutator**.

Bild 12.5 erinnert an die Leitermatrizen von Bild 12.1b. Doch reichen die einfachen Diodenmatrizen nicht aus, um Verbindungen nach Wunsch herzustellen. Die

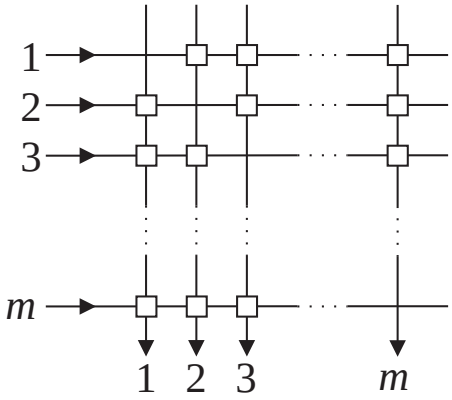


Bild 12.5 Kreuzschienenverteiler

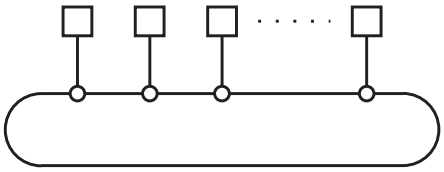


Bild 12.6 Ringverbindung

Schaltungen sind recht aufwendig. Es sind viele weniger aufwendige und dennoch ausreichend flexible Mehrkanalkommutatoren entwickelt worden, die - ebenso wie Diodenmatrizen - letztlich aus *Schaltern* bestehen. Darum werden sie auch *Schalternetze* genannt, eine Bezeichnung, die jedoch, wie wir aus Kap.10.2 wissen, für die gesamte rein elektronische Computerhardware zutreffend ist.

Bild 12.6 zeigt eine andere mögliche Kommunikationsstruktur, eine *Ringverbindung*. Es ist gewissermaßen die der *Sternverbindung* diametral entgegengesetzte Struktur. Alle Teilnehmer benutzen ein und dieselbe Leitung, den **Ringbus**. Jeder Teilnehmer speist seine adressierten Nachrichten in den Ringbus ein, in dem sie solange kreisen, bis der Adressat die Adresse als die seinige erkennt und die Nachricht dem Ring entnimmt. Weitere Methoden, Strukturen und technische Einzelheiten zur Kommunikationstechnik findet der Leser in der Literatur[3].

Bevor auf eine andere wichtige Anwendung der Diodenmatrizen eingegangen wird, soll eine viel elegantere Methode der Mehrkanalkommunikation erwähnt werden, die jeder kennt, die Methode der Senderwahl durch Frequenzabstimmung. Sie kommt in Radio- und Fernsehempfängern zur Anwendung. Während aber die sog. “Radiowellen” oder “Ätherwellen” (die elektromagnetischen Träger der zu übertragenden Signale) den gesamten “Äther” erfüllen, verwendet die moderne Nachrichtentechnik Methoden, welche die Wellen auf engem Raum (in engen “Schläuchen”) zusammenhalten. Das lässt sich entweder mit Lasern bewerkstelligen, die u.a. in Satelliten-Komunikationssystemen eingesetzt werden, oder mit Hilfe spezieller Kabel, in denen sich die Trägerwellen fortbewegen. Die Anzahl der realisierbaren Kanäle pro Kabel und die Anzahl der pro Kanal und Sekunde übertragbaren Bit nimmt mit der Trägerfrequenz zu (vgl. Kap.11.2 [11.3]). Darum ist man bemüht, möglichst hohe Trägerfrequenzen zu verwenden, bis hinauf in den Bereich

3 Siehe z.B. [Tanenbaum 98], [Werner 95].

des sichtbaren Lichts und darüber hinaus. Das ist der Hauptgrund für die Verwendung von *Lichtleitern* in der Nachrichtentechnik.

Diese kurzen Bemerkungen über die Anwendung von Trägerwellen in der Kommunikationstechnik wurden der Vollständigkeit halber eingefügt, obwohl sie nicht zum eigentlichen Thema des Kapitels gehören, dem wir uns nun wieder zuwenden.

12.3.3 Funktionsgeneratoren

In Kap.4.2 war von analog arbeitenden Funktionsgeneratoren, speziell vom Sinusgenerator die Rede gewesen (vgl. Bild 4.2b). Mit Hilfe eines Codeumsetzers lässt sich das digitale Pendant eines solchen Generators aufbauen. Dazu wird zunächst die Funktionstafel in die Matrizen des Codeumsetzers eingetragen (eingprägt), die Argumentwerte in die Adressmatrix und die Funktionswerte in die Speichermatrix. Damit sind die Funktionswerte und deren Adressen gespeichert. Wenn die Möglichkeit bestehen soll, die Wertetafel zu löschen, um eine andere Tafel zu speichern, muss eine PLA verwendet werden, also ein Codeumsetzer mit programmierbarer Adress- und Speichermatrix. Will man die Funktion ausdrucken, muss man die Speicherplätze über die als Adressen dienenden Argumentwerte aufrufen und jeweils beide Werte ausgeben.

Diese Prozedur lässt sich vereinfachen, indem die Speicherplätze durchnummeriert und die Nummern in die Adressmatrix eingetragen werden, während in die Speicherplätze (die Zeilen der Speichermatrix) je ein Argumentwert und der zugeordnete Funktionswert eingetragen werden. In diesem Fall kann ein EPROM-Baustein verwendet werden, denn die Adressiermatrix kann fest programmiert sein.

Um die Argument-Funktionswertepaare der Reihe nach ausgeben zu lassen, kann man einen Zahlengenerator verwenden, der die Nummern sequenziell generiert. Dafür eignet sich ein Taktgeber (z.B. der Taktgeber des benutzten Computers). Durch Zählen und Untersetzen (wiederholtes Halbieren) der Anzahl der Taktimpulse werden die Dualzahlen in steigender Folge generiert. Gibt man sie der Reihe nach auf die PLA, so liefert diese die Wertepaare der Funktion. Diese können gedruckt oder in ein Diagramm überführt werden, indem jedes Paar als Koordinaten je eines Kurvenpunktes interpretiert wird. Auf diese Weise kann der Funktionsverlauf auf dem Bildschirm dargestellt oder durch einen Drucker oder *Plotter* gezeichnet werden. Ein Plotter ist ein Zeichengerät, das an einen Computer als zusätzliche Ausgabeeinheit angeschlossen werden kann.

12.3.4 Steuermatrix

Eine andere Standardaufgabe der Rechentechnik ist die Prozesssteuerung. Betrachten wir als Beispiel die Steuerung des Waschprozesses in einer automatischen Waschmaschine. Die Maschine muss über einen *Steueroperator* verfügen, der eine Folge von *Steuersignalen* generiert, welche die verschiedenen Funktionseinheiten (Wassereinlassventil, Heizung, Trommelmotor, Wasserpumpe) jeweils im richtigen Augenblick ein- bzw. ausschaltet. Als Steuersignal zur Steuerung einer Funktions-

einheit genügt ein einziges Bit, ein *Steuerimpuls*. Wenn gleichzeitig mehrere Einheiten angesteuert werden sollen (z.B, Wasser ab, Heizung an), muss der Steueroperator *Steuerwörter* generieren.

Welches Steuerwort zu einem gegebenen Zeitpunkt generiert werden muss, kann von der Uhrzeit abhängen (z.B. wenn um 10 Uhr gestartet oder wenn 3 Minuten lang geschleudert werden soll) oder vom momentanen *Zustand* des Waschprozesses, der durch Messeinrichtungen, auch *Fühler* genannt, gemessen wird, z.B. durch Temperatur- oder Wasserstandfühler. Die Messwerte müssen dem Steuergenerator *gemeldet* werden, zweckmäßigerweise binär codiert. Die Uhrzeit kann *intern* durch den Steueroperator selbst generiert werden, wenn er über eine Uhr (Impulsgenerator mit Impulszähler) verfügt.

Der Steuersignalgenerator hat also binäre Eingabewörter in binäre Ausgabewörter zu transformieren. Hat man das erkannt, liegt die Idee nicht mehr ferne, mit der M.V.WILKES 1951 berühmt wurde, die Idee der Steuermatrix. Sie besteht darin, dass die Steuerwörter in die Speichermatrix und die Bedingungen für die jeweiligen Steueraktionen in die Adressiermatrix eines ROM eingetragen werden. Die Bedingungen können Bedienaktionen des Nutzers, die Uhrzeit oder Meldungen sein, d.h. Messwerte, die der gesteuerte Operator (die Waschmaschine) dem Steueroperator "meldet".

Im Falle einer Waschmaschine ohne Automatik ist der "Steueroperator" der Mensch, der die Maschine *bedient* und entsprechend der aktuellen Situation (Bedingung) die richtige *Steuerentscheidung* fällt. Die Bedienungsanleitung (die Vorschrift, der sprachliche Operator, der Algorithmus), nach der er zu verfahren hat, muss jeder möglichen *Bedingung* eine bestimmte *Steueraktion* zuordnen. Eine Tabelle, die in der linken Spalte die Bedingungen und in der rechten die notwendigen Aktionen enthält, heißt **Entscheidungstabelle**. Jede Zeile der Tabelle enthält eine **Regel** der Form

Wenn b_1 und b_2 und...und b_n, dann a_1 und a_2...und a_m, (12.5)

worin b_i die Bedeutung hat "Bedingung b_i ist erfüllt" und a_i "Aktion a_i ist auszuführen". Aus der Entscheidungstabelle lässt sich unmittelbar ablesen, wie die Schnittpunktoperatoren (Dioden) in der Adressiermatrix und der Speichermatrix zu platzieren sind.

Es kann zweckmäßig, evtl. sogar notwendig sein, das beschriebene **Matrixsteuerwerk** zu einem *Automaten* zu erweitern. Angenommen, der Waschautomat soll das Spülen, Schleudern und Abpumpen drei mal durchführen. Das kann dadurch erreicht werden, dass drei Regeln programmiert (in den ROM eingeprägt) werden, für jeden Teilvorgang eine Regel. Wenn die drei Teilvorgänge identisch sind, genügt es, eine einzige Regel abzuspeichern und einen Zähler einzubauen. Wenn der Zähler bis 3 gezählt hat, wird der Teilvorgang nicht mehr wiederholt. Der Zählerstand kann als *innerer Zustand* des Steueroperators angesehen werden, sodass dieser zu einem *Automaten* (zu einer Realisierung des endlichen Automaten) wird.

Die Erweiterung einer ROM-Schaltung zu einem Automaten (vgl. Bild 8.5) erfolgt durch Einbau einer Rückkopplung von der Speichermatrix zur Adressiermatrix über ein Verzögerungsglied (D), welches sich das aktuelle Steuerwort merkt und im nächsten Takt auf den Eingang des ROM gibt, der dadurch zu einem *Steuerautomaten* wird. In Bild 12.2 ist eine solche Rückkopplung gestrichelt angedeutet; das zurückgegebene Steuerwort ist (in Analogie zum inneren Zustand des endlichen Automaten) mit *z* bezeichnet. Die Anzahl der senkrechten Leiter der Adressiermatrix muss entsprechend erhöht werden. Auf diese Weise kann das aktuelle Steuerwort vom Steuerwort des vorangehenden Arbeitstaktes abhängig gemacht werden. Da *x* und *y* in dem Beispiel Tripel sind, gibt es 9 verschiedene (*x*,*y*)-Paare, die der Decodierer erkennen muss. Er besitzt also 9 waagerechte Leiter.

Der Inhalt einer Steuermatrix stellt ein oder mehrere als Hardware realisierte Programme dar, die i.d.R. vom Nutzer nicht verändert werden können. Darum werden sie nicht als *Software*, sondern als **Firmware** bezeichnet.

Mit der Einführung einer Rückkopplung haben wir das Thema dieses Kapitels, die mikroelektronische Realisierung zirkelfreier boolescher Netze bereits verlassen und den ersten Schritt zum programmierbaren Rechner getan.

13 Von der Kombinationsschaltung zum Von-Neumann-Rechner

Zusammenfassung

Die Realisierung von Funktionen durch Kombinationsschaltungen hat zwei wesentliche Mängel. Zum einen müssen die Funktionswerte vor dem Entwurf der Schaltung bekannt sein. Zum anderen ist es bei einer Erweiterung der Funktionstafel, beispielsweise bei Erhöhung der Genauigkeit der Funktionswerte, nicht mit einer Erweiterung der Schaltung getan, vielmehr muss sie völlig neu entworfen und aufgebaut werden. Beide Mängel werden durch die Verwirklichung der dritten Grundidee des elektronischen Rechnens, der Idee des programmierbaren Rechners behoben.

Unbeschränkt erweiterbare Funktionstafeln (z.B. die Additionstafel) lassen sich bei binär-statischer Codierung nur durch zirkuläre boolesche Netze in Form von KR-Netzen (Netze aus Kombinationsschaltungen und Registern) realisieren. Der Prototyp des programmierbaren Rechners ist ein als *Von-Neumann-Rechner* bezeichnetes KR-Netz. Seine Hauptbestandteile sind ein *Prozessor* und dessen Arbeitsspeicher, *Hauptspeicher* genannt. Der Prozessor kann imperative Algorithmen abarbeiten, die als *Maschinenprogramme*, d.h. als Folge von Maschinenbefehlen formuliert sind. Dabei holt sich der Prozessor die Befehle und die notwendigen Operanden der Reihe nach aus dem Hauptspeicher und liefert die Ergebnisse an den Hauptspeicher zurück. Prozessor und Hauptspeicher liegen in einer (äußeren) Rückkopplungsschleife. Die Datenübergabe in beiden Richtungen erfolgt über eine einzige Verbindung, den sog. *von-neumannschen Flaschenhals*.

Der Prozessor enthält eine innere Rückkopplungsschleife, in welcher die ALU (arithmetisch-logische Einheit), der *Akkumulator* (AC) und ein oder mehrere Datenregister für die schnelle Zwischenspeicherung von Operanden liegen. Die ALU, der Akkumulator, die Datenregister und einige weitere Register sind zu einem KR-Netz verbunden und bilden eine funktionelle Einheit, RALU genannt, die ihrerseits zusammen mit einem Steueroperator den Prozessor bildet. Aus den ALU-Operationen werden durch Steuerung des Operandenflusses, der durch das RALU-Netz fließt, die Maschinenoperationen komponiert. Die Flusssteuerung wird zum Teil von der RALU selbst (dezentral), zum Teil vom Steueroperator (zentral) durchgeführt. Dabei können die Operanden sowohl in der inneren als auch in der äußeren Rückkopplungsschleife zirkulieren.

Der Steueroperator kann als Matrixsteuerwerk ausgebildet sein, d.h. er kann eine oder mehrere ROM-Bausteine enthalten, in denen Programme (als sog. Firmware) gespeichert sind, sodass ein Maschinenbefehl eventuell eine umfangreiche Kompositaktion auslöst. Damit ein Maschinenbefehl vom Prozessor ausgeführt werden kann, muss er das Format des *Befehlsregisters* besitzen, in das der Prozessor jeden

aus dem Hauptspeicher geholten Befehl einliest. Das Format legt fest, welche Teile des Befehls die Operation und welche die Operandenadressen darstellen (codieren). Die Befehle einer *Zwei-Adress-* bzw. *Drei-Adress-Maschine* enthalten zwei bzw. drei Adressen.

Der Prozessor komponiert Operatoren (Operationen) auf zwei Ebenen. Auf der unteren Ebene komponiert er Maschinenoperationen aus den ALU-Operationen und auf der oberen Ebene Kompositoperationen aus den Maschinenoperationen gemäß den Vorgaben von *Maschinenprogrammen*. Ein Maschinenprogramm ist eine Folge von Maschinenbefehlen; es stellt also keinen Datenflussplan, sondern einen *Aktionsfolgeplan* dar, und die Komponierung erfolgt nicht nach den Prinzipien der USB-Methode. Demgegenüber erfolgt die Komponierung der Maschinenoperationen durch Steuerung des Operandenflusses in der RALU nach den Prinzipien der USB-Methode.

Dennoch kann der Von-Neumann-Rechner jeden Datenflussplan realisieren und folglich jede rekursive Funktion berechnen. In diesem Sinne ist er universell. Der Datenflussplan als solcher tritt nicht zutage, weil die Operandenplätze im Hauptspeicher zentralisiert sind (wie im Falle des endlichen Automaten) und die Operationen sequenziell ausgeführt werden. Voraussetzung der Universalität ist die Existenz eines Sprungbefehls zur Simulierung von Zweigeweichen. Ein *Sprungbefehl* veranlasst das Herausspringen aus der normalen Befehlsfolge und die Fortsetzung der Abarbeitung an einer anderen Stelle des Programms.

Mit Hilfe des Sprungbefehls lassen sich Iterationsschleifen programmieren. Theoretisch sind Iterationsschleifen ohne Abbruchkriterium denkbar. Eine Iteration, die nicht terminiert, hat zwar keinen praktischen Sinn und widerspricht dem Realisierbarkeitsprinzip, doch hat sie theoretische Bedeutung. Mit Hilfe nichtterminierender Iterationen lassen sich Funktionen mit abzählbar unendlicher Wertetafel definieren und programmieren. Die Ausführung der Vorschrift ist ein nicht endender Prozess, d.h. eine abzählbar unendliche Folge von Ereignissen.

Eine Funktion, für deren Berechnung eine Vorschrift angebbar ist, welche von einem realen Operator ausgeführt werden kann, der mit statischer Codierung arbeitet, heißt *statisch berechenbare Funktion*. Die Klasse der statisch berechenbaren Funktionen ist mit der Klasse der rekursiven Funktionen und der Klasse der mittels Von-Neumann-Rechner berechenbaren Funktionen identisch.

13.1 Die dritte Grundidee des elektronischen Rechnens

Wir sind nun ausreichend vorbereitet, um den Computer nachzuerfinden, genauer den **Prozessorcomputer**, d.h. einen Computer, der aus einem oder mehreren Prozessoren und einem oder mehreren Speichern besteht. Neben diesem Computertyp wird seit längerem an einem ganz anderen Typ gearbeitet, dem **Neurocomputer**. An die Stelle der Prozessoren und Speicher treten Netze aus künstlichen Neuronen, die

sowohl die Verarbeitungsfunktion als auch die Speicherfunktion übernehmen. Wir interessieren uns ausschließlich für den *Prozessor*computer. Vergegenwärtigen wir uns zu diesem Zweck noch einmal unser Ziel sowie die wichtigsten Aussagen und Einsichten, zu denen wir gelangt waren.

Wir hatten in Kap.8.5 unser ursprüngliches Ziel - wir nennen es Ziel 1 - umformuliert und ein scheinbar ganz neues Ziel - Ziel 2 - festgelegt. Wir stellen die beiden Ziele noch einmal einander gegenüber:

Ziel 1: Entwurf eines Gerätes, das jede berechenbare Funktion berechnen kann. 1

Ziel 2: Realisierung eines *Basiskalküls*, in den sich alle Kalküle transformieren lassen, derer sich die natürliche Intelligenz bedient, und Durchführung dieser Transformationen in den Basiskalkül.

Aus Kap.8.5 wissen wir, dass beide Formulierungen einander äquivalent sind.

Die erste Idee zur Erreichung von Ziel 2 (die “erste Grundidee des elektronischen Rechnens”) bestand darin, die boolesche Algebra als den zu realisierenden Basiskalkül zu wählen. Die zweite Grundidee ermöglichte die elektronische Implementierung der elementaren booleschen Operatoren, sodass sich die boolesche Algebra nach der USB-Methode realisieren lässt, soweit das Realisierungsprinzip dies zulässt. Es bleibt die Frage offen, wie sich beliebige Kalküle, derer sich die natürliche Intelligenz bedient, in die boolesche Algebra transformieren lassen. Auf diese Frage kommen wir in Teil 3 zurück. Kapitel 13 ist der Frage gewidmet, wie sich einfache zahlenmäßige Rechnungen in den booleschen Kalkül transformieren und mittels elektronischer Schaltung ausführen lassen. Um an den Ergebnissen der Kapitel 10 und 12 leichter anknüpfen zu können, bleiben wir zunächst bei der Zielstellung, wie sie unter “Ziel 1" formuliert ist.

Aus Kap.9.3 [9.16] wissen wir, dass sich die Wertetafel jeder Funktion in eine Kombinationsschaltung überführen lässt. Damit scheint der Weg zum Ziel 1 frei zu sein. Leider ist dieser Weg aus Aufwandsgründen nicht gangbar. Es müssten nicht nur “unendlich viele” Kombinationsschaltungen gebaut werden, sondern bereits die Größe der einzelnen Schaltungen würde die Grenzen des Machbaren übersteigen. Das haben die Abschätzungen in Kap.9.2 [9.6] gezeigt. Der Grund des Dilemmas liegt in der Starrheit von Kombinationsschaltungen.

Nach dieser Bemerkung liegt die **dritte Grundidee des elektronischen Rechnens** auf der Hand. Sie besteht in der Komponierung *steuerbarer* boolescher Netze. Steuerbarkeit setzt das Vorhandensein steuerbarer Flussknoten (Weichen bzw. Tore) voraus. Die einfachsten steuerbaren Netze sind *steuerbare Kombinationsschaltungen*, also zirkelfreie boolesche Netze, die eine oder mehrere Weichen enthalten. In einem solchen Netz können verzweigte, nichtzirkuläre Operandenflüsse gesteuert werden. Vorgreifend sei erwähnt, dass die ALU ein solches Netz ist.

Die Verwendung von Weichen beim Komponieren von Operatorennetzen eröffnet aber weit größere Möglichkeiten, denn mit ihrer Hilfe können zirkuläre Kopplungsstrukturen aufgebaut und zirkuläre Operandenflüsse gesteuert werden, vorausgesetzt, das Netz enthält Speichereinheiten [9.19]. Die charakteristischen Merkmale

steuerbarer Netze sind bereits aus Bild 8.1 zu erkennen, nämlich die Existenz von Speichern und Weichen und evtl. von Rückkopplungen. Wir wollen uns die weitreichenden Konsequenzen der Verwendung von Weichen klarmachen.

In Kap.8.4.5 hatten wir die Komponierung steuerbarer Operatorennetze auf abstrakter Ebene, ohne Bezugnahme auf boolesche Operatoren untersucht. Dort waren wir auf ganz anderem Wege auf das Komponierungsproblem gestoßen, nämlich bei der Suche nach universellen Methoden der Algorithmenbeschreibung. Die Gemeinsamkeit mit unserer jetzigen Fragestellung ergibt sich aus der Notwendigkeit der Steuerung. Diese ist von einem Steueroperator (Gerät oder Mensch) nach einer bestimmten Steuervorschrift auszuführen. Die Steuervorschrift muss genau festlegen, welche Operationen in welcher Reihenfolge an welchen Operanden auszuführen sind, m.a.W. sie muss ein *imperativer Algorithmus* sein, wie er in Kap.7.2 [7.10] eingeführt worden ist. Wenn nichts anderes gesagt wird, ist in Kapitel 13 unter einem Algorithmus ein imperativer Algorithmus zu verstehen.

Die Steuervorschrift für ein steuerbares boolesches Netz nennen wir **Programm** und das Erstellen (Artikulieren) von Programmen **Programmieren**. In diesem Kapitel ist also Programmieren das Artikulieren imperativer Algorithmen. In Kap.8.4 hatten wir gefragt, ob es eine universelle Methode für die Artikulierung von Algorithmen gibt. Aus jetziger Sicht lautet die Frage: Gibt es eine *universelle* Methode für die *Komponierung* und *Programmierung* steuerbarer boolescher Netze? *Universell* bedeutet - analog zu Kap.8.4 -, dass sich für jede Funktion ein Netz und ein Programm (Algorithmus) angeben lässt nach welchem das Netz die betreffende Funktion berechnet. (Es wird davon ausgegangen, dass das Netz Ein- und Ausgang besitzt, also ein Operator ist.)

In Kap.8.4 musste die Frage abstrakt gestellt und beantwortet werden. Jetzt können wir sie konkret stellen, indem wir von realen Operatoren ausgehen, welche die Programme (Algorithmen) ausführen. Früher war der zugrunde gelegte ausführende Operator der Mensch, und eine konkrete Lösung des Problems (im Sinne des Trägerprinzips) hätte von der Funktionsweise des Gehirns ausgehen müssen. Jetzt besteht die Aufgabe zunächst darin, ein geeignetes universelles steuerbares boolesches Netz zu erfinden, also einen variablen Kompositoperator, der sich auf jede effektiv berechenbare Funktion programmieren lässt. Das wird uns eine ganze Weile beschäftigen, bevor wir uns dem Programmieren und den Sprachen zuwenden können, in denen programmiert wird.

Zuerst muss aber noch die oben gestellte Frage nach den Konsequenzen der Steuerbarkeit vollständig beantwortet werden. Die Konsequenzen sind verstreut in den vorangehenden Kapiteln enthalten und in der folgenden Auflistung zusammengefasst und kurz kommentiert.

1. **Programmierbarkeit**. Ein steuerbares boolesches Netz ist variabel, es lässt sich auf die Berechnung verschiedener Funktionen “programmieren”.
2. **Komponierbarkeit zirkulärer Netze**. Ihr entspricht in Kap.8.4.5 die Möglichkeit der Operatorkomponierung mittels Iteration (vgl. Bild 8.9).

3. **Notwendigkeit von Toren und Speichern**. Sie ergibt sich für zirkuläre Netze aus dem Prinzip der statischen Codierung (siehe Kap.9.4 [9.19]).
4. **Halteproblem**. Im Falle zirkulärer Netze muss das Programm angeben, wann die Operationsausführung zu beenden ist. Das kann durch explizite Vorgabe der Iterationszahl oder durch ein Prädikat erfolgen. Es ist nicht immer feststellbar, ob ein gegebenes Programm terminiert, d.h. ob der programmierte zirkuläre Prozess anhält (siehe Kap.8.3 [8.22]), m.a.W. ob es einen Algorithmus beschreibt oder nicht.
5. **Universalität**. Es besteht die Möglichkkeit, universelle steuerbare boolesche Netze zu komponieren. Dafür ist - neben der Realisierung der Flussknoten - ein einziger elementarer Operatortyp ausreichend, der *Inkrementierer*, also ein Operator, der eine Zahl um 1 erhöht (siehe Kap.8.4.5.). Er kann als zirkelfreier boolescher Operator realisiert werden, denn die Addition einer 1 kann vom Halbaddierer ausgeführt werden (siehe die Bilder 9.3 und 9.4).

Die Punkte 1 und 2 sind unmittelbare Folgen der Steuerbarkeit. Die Punkte 3 und 4 sind Konsequenzen von Punkt 2 und insofern bedingte Konsequenzen der Steuerbarkeit. Sie entfallen für steuerbare zirkel*freie* Netze. Punkt 5 liefert uns den Wegweiser zu unserer Erfindung, denn er enthält de facto das Arbeitsprinzip des Prozessors. Wir müssen “nur” noch die steuerbare Schaltung erfinden, in die der Inkrementierer so eingebettet ist, dass sich mittels Iteration “alles” berechnen lässt. In den Kapiteln 13.4 und 13.5 werden wir sehen, wie sich diese Idee in etwas abgewandelter Form verwirklichen lässt (die Rolle des Inkrementierers wird von der ALU übernommen, die unter anderem auch inkrementieren kann).

Übergangen wurde bislang das Problem der Speicherung. Das zu komponierende steuerbare Netz kann nur dann zu Recht als *boolesches* Netz bezeichnet werden, wenn auch die erforderlichen Tore und Speicher aus elementaren booleschen Operatoren, konkret aus Schaltern komponiert sind. Im folgenden Kapitel werden wir uns überlegen, wie sich dies bewerkstelligen lässt.

Die wichtigste Konsequenz der Steuerbarkeit ist zweifelsohne die Programmierbarkeit. Sie war zunächst von uns gar nicht beabsichtigt, denn auf die Idee der Steuerbarkeit hatte uns das Dilemma mit den Kombinationsschaltungen gebracht, die immense Ausmaße anzunehmen drohten. So betrachtet ist die Steuerbarkeit lediglich als Nebenprodukt anzusehen, was jedoch angesichts ihrer prinzipiellen Bedeutung kaum gerechtfertigt ist. Denn Programmierbarkeit macht erst möglich, was man von einem Rechner erwartet, nämlich die Berechnung von Funktionswerten, die zuvor noch nicht berechnet worden waren. Die Berechnung setzt freilich voraus, dass ein Berechnungsalgorithmus, ein terminierendes Programm vorliegt. Dagegen setzt die Realisierung einer Funktion als Kombinationsschaltung voraus, dass ihre Wertetafel vorliegt.

Sicher hat beides die Erfinder bei der Suche nach dem universellen Rechner beflügelt, sowohl die technische Machbarkeit als auch die Programmierbarkeit. Bei den Pionieren des “maschinellen Rechnens”, den Erfinder der ersten universellen

"Rechen-Maschinen", hat aber wohl doch die Programmierbarkeit an erster Stelle gestanden, d.h. *die dritte Grundidee des elektronischen Rechnens, die* ***Programmsteuerung***. Diese Idee stammt von CHARLES BABBAGE, wenn das Wort "elektronisch" zu "maschinell" verallgemeinert wird.

Zu den Wörtern "Maschine" und "maschinell" ist eine Bemerkung am Platze. Bei dem Wort "Maschine" werden die meisten Menschen zunächst einmal an Werkzeugmaschinen, Baumaschinen oder Kraftmaschinen denken. Man könnte sie alle unter der Bezeichnung "Energie verarbeitende Maschinen"[1] zusammenfassen. Die analoge Wortverbindung "Information verarbeitende *Maschine*" ergibt sich fast "automatisch" als zusammenfassende Bezeichnung für alle "künstlichen Vorrichtungen", die der Verarbeitung von Zeichen und Zeichenketten dienen; dazu gehören Kassenautomaten ebenso wie mechanische Rechenmaschinen und moderne Computer. Auch sie als *Maschinen* zu bezeichnen, den Computer eingeschlossen, ist durchaus "passend". Denn das griechische Wort, von dem sich das Wort "Maschine" herleitet, hat die Bedeutung "künstliche Vorrichtung" oder "Werkzeug". Und schließlich ist die kulturgeschichtliche und philosophische Problematik, die durch die Gegenüberstellung "Mensch und Maschine" angesprochen wird, für Energie verarbeitende und Information verarbeitende Maschinen ein und dieselbe. Man denke an die Wechselwirkung zwischen Mensch und Maschine und an die Rückwirkung der Maschine auf die kulturelle Evolution, konkret die Rückwirkung der Dampfmaschine als Initiator der "*industriellen Revolution*" im 19. Jahrhundert und an die Rückwirkung der Rechenmaschine als Initiator der gegenwärtigen "*informationellen Revolution*".

Dieser Bemerkung soll eine zweite angefügt werden, jedoch für Leser, die sich in der Entwicklung der Rechentechnik auskennen. Der Prozessor, der in diesem Kapitel beschrieben wird, und manche Begriffe, die dabei verwendet werden, machen auf den Fachmann eventuell einen etwas antiquierten Eindruck. Wer spricht beispielsweise heute noch von Zwei-, Drei- oder Vier-Adress-Rechnern? Der Weg, den wir gehen werden ist durch den Wunsch vorgegeben, als Nichtfachmann den Computer *nach*zuerfinden. Die Vorsilbe "nach" bedeutet, dass wir uns in der *Nach*folge der Pioniere der Computertechnik befinden. Man muss den "ersten", noch ganz "primitiven" Computer nacherfunden haben, um das Grundprinzip zu verstehen, nach dem alle späteren Computer arbeiten. Dies ist auch der didaktisch beste Weg zum Kern der elektronischen Rechentechnik. In Kap.19.2 werden die wichtigsten zusätzlichen Ideen nachgetragen, welche die Entwicklung von der "ersten elektronischen Rechenmaschine" zum modernen Computer markieren.

---

1 Richtiger wäre die Bezeichnung "Energie transformierende Maschinen".

13.2 Elektronische Speicher

13.2.1 Register

Bevor wir den Gedanken der Programmierbarkeit weiter verfolgen, schieben wir einige wichtige Überlegungen zum Problem der Aufbewahrung (Speicherung) von Operanden ein und wenden uns zunächst der Frage zu, wie die erforderlichen Tore und Speicherplätze (die kleinen Quadrate in Bild 8.1a; die fetten Seiten symbolisieren die Eingabetore), über welche die Operanden in einem Operatorennetz von Operator zu Operator weitergegeben werden, aus booleschen Operatoren aufgebaut werden können.

Der Leser ahnt vielleicht schon, wie ein solcher Speicher funktionieren könnte. Er braucht sich nur an den Ein-Bit-Speicher aus Kap.9.5 und an das Schlüssel-Schloss-Prinzip aus Kap.12.3.2 [12.3] zu erinnern, um eine Idee zu haben, wie der Arbeitsspeicher eines Computers aufgebaut werden kann. Doch wollen wir nicht mit dem Endprodukt beginnen, sondern Schritt für Schritt vorgehen. Zunächst muss das boolesche Netz des Ein-Bit-Speichers in ein Schalternetz überführt werden. Zu diesem Zweck sind die booleschen Operatoren durch Schalterkombinationen zu ersetzen, wie in Kap.10.1 dargelegt wurde. Die technische Realisierung der Schalter kann, wie besprochen, mittels Transistoren erfolgen.

Den Speicher für eine Bitkette, die einen Operanden intern codiert, nennen wir **Register**. Die Operandenlänge ist i.Allg. nicht länger als die sog. **Rechnerwortlänge**; das ist die Bitkettenlänge, mit der ein bestimmter Computer standardmäßig hantiert. Sie stellt einen wichtigen technischen Parameter eines Computers dar. Von ihr hängt die Rechengenauigkeit und Rechengeschwindigkeit ab. Beide lassen sich durch Erhöhung der Rechnerwortlänge steigern. Vor gar nicht langer Zeit setzte sich als PC-Standard eine Länge von 32 Bit durch. Aber schon sind 64 Bit im Gespräch. Ältere PCs arbeiteten mit 16 Bit und noch ältere mit 8 Bit.

Ein Speicher (Register) für 32 Bit lässt sich aus 1-Bit-Speichern komponieren, indem man die Schaltung von Bild 9.7 (oder eine andere Flipflop-Variante) 32-mal nebeneinander anordnet, wie in Bild 13.1a dargestellt ist. Jedes Quadrat symbolisiert einen Ein-Bit-Speicher. Die Gabel spaltet die Bits der zu speichernden Bitkette in einzelne Bits auf (gedankliches Aufspalten durch eine Spaltegabel), die Vereinung fasst die Bits wieder zu einem Wort zusammen. Hardwaremäßig handelt es sich um das “Aufbinden” bzw. “Zusammenbinden” der Einzelleitungen eines mehradrigen Kabels bzw. - bei mikroelektronischer Realisierung - eines Leitertupels auf einem Chip. Als Symbol für Register verwenden wir ein Quadrat oder Rechteck mit einer fetten Seite, die das Eingabetor bzw. die Eingabetore symbolisiert.

In das Register von Bild 13.1a werden die einzelnen Bits *gleichzeitig* (**parallel**) ein- bzw. ausgelesen. Unter Umständen kann es aber zweckmäßig oder sogar notwendig sein, sie *nacheinander* (**sequenziell**) ein- oder auszulesen. Das lässt sich auf zweierlei Weise bewerkstelligen, zum einen nach dem FIFO-oder **Silo**-Prinzip,

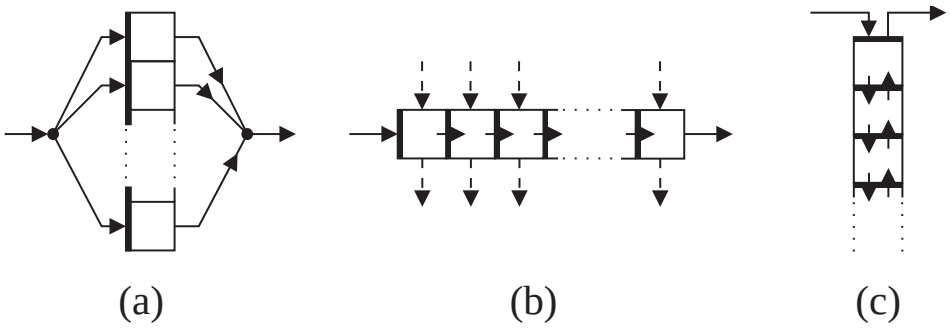


Bild 13.1 Register. **(a)** - Parallelregister; **(b)** - Schieberegister; **(c)** - Stapelregister.

zum anderen nach dem LIFO- oder **Stapel**-Prinzip (FIFO von "First In First Out", LIFO von "Last In First Out"). Beim Siloprinzip wird, wie beim Getreidesilo, zuerst (beim Silo unten) ausgegeben, was auch zuerst (beim Silo oben) eingespeichert wurde. Das Stapelprinzip entspricht dem Wachsen und Abnehmen eines Aktenstapels, auf den Zugänge obenauf gelegt werden und dem die jeweils oberste Akte zur Bearbeitung entnommen wird.

Für Register, die nach dem Siloprinzip arbeiten, hat sich die Bezeichnung **Schieberegister** eingebürgert, weil die Bits durch das Register "durchgeschoben" werden. Es bietet sich z.B. für die Speicherung der Summanden eines sequenziellen Addierers an, der die einzelnen Stellenwerte der Reihe nach addiert, wie man es in der Schule gelernt hat. Ein solcher Addierer kann aus einem Volladdierer (vgl. Kap. 9.3 [9.13]), einem Verzögerungsglied (vgl. Bild 12.2) und drei Schieberegistern aufgebaut werden. Aus den beiden Summandenregistern wird in jedem Takt je ein Bit herausgeschoben und dem Volladdierer übergeben, der das Summenbit (y in Bild 9.2) in das Summenregister hineinschiebt und den Übertrag (z in Bild 9.2) dem Verzögerungsglied übergibt, das ihn im nächsten Takt auf den Eingang des Volladdierers zurückgibt.

2 Die Addition lässt sich dadurch beschleunigen, dass 32 Volladder parallelgeschaltet werden. Der so komponierte **Paralleladdierer** kann die Addition von Bitketten in einem einzigen Schritt ausführen. Seine Arbeitsregister müssen Parallelregister sein. Der Übergang von der sequenziellen zur parallelen Verarbeitung und der entgegengesetzte Übergang sind Standardoperationen der Hardware. Sie lassen sich mit Registern realisieren, deren Inhalt sowohl parallel (über die gestrichelten Pfeile in Bild 13.1b) als auch sequenziell (über die ausgezogenen Pfeile) ein- und ausgegeben werden kann.

13.2.2 Adressierbarer elektronischer Speicher

Nach dem Prinzip des hierarchischen Komponierens lassen sich Register als Bausteinspeicher für die Komponierung komplexerer Speicherstrukturen einsetzen. Die Register werden dann Speicherplätze genannt. Im Normalfall wird jeweils auf *einem* Speicherplatz *ein* Rechnerwort abgespeichert. Die Speicherplätze werden

durchnummeriert. Die Nummern spielen die Rolle von *Adressen* zur Adressierung der Plätze. Auf diese Weise ist eine vollständige Ordnung der Speicherplätze und damit auch der in ihnen abgespeicherten Rechnerworte festgelegt.

Das Abspeichern und Auslesen einer Folge von Rechnerworten kann wiederum nach dem Silo- oder nach dem Stapelprinzip erfolgen, also gemäß den Bildern 13.1b bzw. c, wo nun aber ein Quadrat ein Register (einen Speicherplatz) darstellt. Die so entstehenden Speicher heißen Silo- bzw. Stapelspeicher. Stapelspeicher werden häufiger **Kellerspeicher** oder - wie im Englischen - **Stack** genannt. Den Kellerspeicher kennen wir bereits aus Kap.8.4.6 (siehe Bild 8.11). Wenn die Speicherung völlig unsystematisch, d.h. ohne jedes Ordnungsprinzip erfolgt, spricht man von **Heap-Speicherung** (heap = Haufen).

Silospeicher sind am Platze, wenn sich Warteschlangen bilden, z.B. eine Folge von Operanden, die auf die Bearbeitung durch einen Operator warten. Der Einsatz von Kellerspeichern kann bei rekursiven Berechnungen zweckmäßig sein. Silo- und Kellerspeicher erlauben keinen unmittelbaren Zugriff auf die einzelnen Speicherplätze.

Wenn auf die Speicherplätze eines Speichers über ihre Adressen direkt zugegriffen werden soll (sog. *Direktzugriff*), muss der Speicher mit einer geeigneten Zugriffshardware ausgerüstet sein. Das ist z.B. notwendig, wenn der Speicher einem Prozessor als Arbeitsspeicher dient. Denn der Prozessor muss während der Abarbeitung eines Programms ständig gezielt auf Speicherplätze zugreifen, um Daten zu holen oder zu speichern. Er muss also mit jeder Speicherzelle über deren Adresse verbunden werden können. Dazu sind Tore in den Ein- und Ausgabeleitungen der einzelnen Speicherzellen erforderlich, die per Adresse geöffnet werden können. Wenn alle Datenein- und -ausgaben über einen gemeinsamen Bus erfolgen, ergibt sich die Kommunikationsstruktur des Halbkommutators (Bild 12.4).

Damit liegt die Idee, wie ein *adressierbarer Speicher* aufgebaut werden kann, auf der Hand. Zuerst wird die Schaltung eines Registers entworfen, das für jedes Bit eines Rechnerwortes je einen Ein-Bit-Speicher enthält. Sodann werden so viele Register zu einem Speicher zusammengefasst, wie zur Realisierung einer geplanten Speicherkapazität erforderlich sind. Um den Speicher zu einem *adressierbaren Speicher* zu machen, muss jeder Speicherplatz mit einem “Eingangsschloss” und einem “Ausgangsschloss” versehen werden, d.h. in seiner Eingabeleitung und Ausgabeleitung muss je ein AND-Glied vorgesehen werden, und zwar das AND-Glied der betreffenden Zeile des Demultiplexers (DMUX) bzw. des Multiplexers (MUX) eines Halbkommutators in Bild 12.4.

All diese “Bausteinschaltungen”, aus denen ein adressierbarer Speicher “komponiert” wird, können auf einem einzigen Chip realisiert werden. Auf diese Weise lassen sich gegenwärtig Speicherchips mit Speicherkapazitäten von mehreren MByte[2] herstellen.

Damit haben wir den elektronischen **Direktzugriffspeicher** oder **RAM** (Random Access Memory) nacherfunden[3]. Neben dem soeben beschriebenen RAM hat sich

der sog. **dynamische RAM**, abgekürzt **DRAM** durchgesetzt. Er nutzt - ebenso wie der Floating-Gate-MOS-FET (siehe Kap.12.2) - den heute erreichbaren sehr hohen Isolationswiderstand von Kondensatoren. Die codierenden Zustände eines Bit sind zwei Ladungszustände eines winzigen Kondensators. Da der Isolationswiderstand aber nicht unendlich groß ist, fließt die vorhandene Ladung mit der Zeit ab und der Speicherinhalt muss periodisch *aufgefrischt* werden, worauf das Wort "dynamisch" hinweist.

In Kap.12.3.2 [12.4] war die Möglichkeit erwähnt worden, die Adresse in Form eines *Schlüsselwortes* mit der Nachricht direkt zu verbinden (voranzustellen oder anzuhängen). Diese Methode lässt sich auch auf die Speicheradressierung anwenden. Dann wird das Schlüsselwort de facto zu einem Teil des Speicherinhalts. Die Festlegung, dass das Schlüsselwort die Adresse (die Nummer) der Zelle ist, kann fallen gelassen werden, denn der Zugriff kann über beliebig vereinbarte Schlüssel-
3 wörter erfolgen. Das führt zur Idee des **Assoziativspeichers**.

Die Bezeichnung "*Assoziativ*speicher" bringt zum Ausdruck, dass mit einem Teil des abgespeicherten Inhalts, dem Schlüsselwort oder **Suchargument**, der restliche Inhalt "*assoziiert*" wird. Das Suchargument spielt die Rolle einer Adresse. Damit eröffnen sich neue Möglichkeiten für das Auffinden von Speicherinhalten. Da als Suchargument jedes Binärwort zulässiger Länge erlaubt ist, können geeignet gewählte Bezeichnungen oder Namen als Suchargumente dienen. Das kann z.B. der (codierte) Name eines Mitarbeiters in einer Mitarbeiterdatei sein. Mit ihm könnten alle unter diesem Suchwort abgespeicherten Personaldaten abgerufen (assoziiert) werden. Das Beispiel weist auf die Bedeutung des assoziativen Zugriffs in Datenbanken hin (siehe Kap.16.2[16.6]). Dabei können die "adressierten Speicherplätze" recht umfangreich sein und viele hardwaremäßig realisierte, adressierbare Speicherplätze umfassen.

Eine weitere wichtige Bedeutung der assoziativen Methode liegt in der Möglichkeit, mehrere Speicherplätze gleichzeitig anzuwählen. Wenn beispielsweise in einer Bibliotheksdatei als Suchargument der Autorenname dient, kann durch Eingabe eines bestimmten Namens auf sämtliche Titel aller Autoren dieses Namens zugegriffen werden.

13.3* Einschub: Berechenbarkeits-Äquivalenzsatz

Bevor wir unseren Weg zum Von-Neumann-Rechner fortsetzen, wollen wir uns überlegen, was die Schaltungen, die wir bisher nacherfunden haben, die Kombinati-

---

2 Ein Byte sind 8 Bit; ein MByte (Megabyte) sind 2^{20}, also etwas mehr als 10^6 Byte.

3 Es ist zu beachten, dass "RAM" zuweilen auch als Abkürzung für die Registermaschine (in Kap.8.4.3 mit "URM" bezeichnet) verwendet wird. Ferner ist zu erwähnen, dass auch nicht rein elektronische [9.24] Direktzugriffsspeicher existieren, z.B. Scheibenspeicher oder die heute kaum noch anzutreffenden Ferritkernspeicher.

onsschaltungen und Register, zu leisten in der Lage sind, wenn aus ihnen Netze komponiert werden. In Kap.9.4 [21] waren wir zu einer Aussage gelangt, die zu folgendem Satz verallgemeinert werden kann.

Satz 1. Informationelle Operatoren mit statischer Codierung sind notwendigerweise Netze aus Kombinationsschaltungen und Speichern, wobei in jeder Verbindung (in jedem Operandenweg) zwischen zwei Kombinationsschaltungen ein Speicher mit Eingangstor liegt, oder sie sind in solche Netze überführbar.

Die Verallgemeinerung besteht darin, dass an die Stelle von booleschen Speichern beliebige Speicher treten können. Das bedeutet, dass bei der Überführung, von der in Satz 1 die Rede ist, jeder nicht rein elektronische Speicher des informationellen Operators (Computers), durch einen rein elektronischen zu ersetzen ist, z.B. durch einen elektronischen RAM.

Inzwischen wissen wir, dass Speicher mit Eingangstoren in Form von Registern (geordneten Mengen von Ein-Bit-Speichern) realisiert werden können, sodass sich ein Netz aus Kombinationsschaltungen und Registern ergibt. Ein solches Netz nennen wir KR-Netz. ***KR-Netze*** *sind* ***zirkelfreie*** *oder* ***zirkuläre*** *Netze aus Kombinationsschaltungen und Registern, wobei in jeder Leitung für die Bitkettenübergabe zwischen zwei Kombinationsschaltungen ein Register liegt*. Es wird davon ausgegangen, dass Register stets mit Eingabetoren ausgerüstet sind.

Ein KR-Netz mit einem externen Eingang und einem externen Ausgang ist ein steuerbarer Kompositoperator höherer Komponierungsstufe. Wir nennen ihn **KR-Operator**. Der einfachste KR-Operator ist eine Kombinationsschaltung mit vorgeschaltetem Register. Damit kann Satz 1 kompakter artikuliert werden: *Informationelle Operatoren mit binär-statischer Codierung sind KR-Operatoren oder in solche überführbar*. In diesem Satz kann das Adjektiv “binär” ohne Einschränkung der Allgemeinheit gestrichen werden, denn jede statische Codierung kann mittels Kombinationsschaltung in binär-statische umcodiert werden. Es gilt der verallgemeinerte

Satz 2. Informationelle Operatoren mit statischer Codierung sind KR-Operatoren oder in solche überführbar.

Eine Funktion, für deren Berechnung ein KR-Operator angegeben werden kann, nennen wir **KR-berechenbare** Funktion oder **KR-Funktion**. Eine Funktion, die von einem informationellen Operator mit statischer Codierung berechnet werden kann - eine solche Funktion hatten wir *statisch berechenbare* Funktion genannt -, ist also eine KR-Funktion. Damit ergibt sich

Satz 3. Die Klasse der statisch berechenbaren Funktionen ist mit der Klasse der KR-Funktionen identisch.

Enthält ein KR-Operator zwei oder mehrere Kombinationsschaltungen, die über Register hintereinandergeschaltet sind, oder enthält er eine Rückkopplungsschleife, in der ein Register liegt, so arbeitet er zwangsläufig **sequenziell**, d.h. während der Operationsausführung eines solchen Operators werden zwei oder mehrere Bausteinoperationen in einer bestimmten zeitlichen Reihenfolge ausgeführt. Aus diesem Grund werden solche KR-Operatoren auch **Folgeschaltungen** genannt.

Welche Werte ein bestimmter KR-Operator berechnet, hängt davon ab, welche Tore in welcher Reihenfolge geöffnet werden. Die Öffnungsimpulse müssen von einem *Steueroperator* in der erforderlichen Reihenfolge generiert werden. Man betrachte unter diesem Aspekt noch einmal Bild 8.1 und nehme an, dass die Bausteinoperatoren Kombinationsschaltungen und die Operandenplätze Register sind. Das Stellen der Weichen in Bild 8.1a erfolgt durch Steuersignale, welche die Eingangstore der entsprechenden Register öffnen.

4 Man erkennt, dass die Schaltung (die graphische Darstellung) eines KR-Operators nichts anderes ist als ein Operandenflussplan, in welchem die Operatoren als Kombinationsschaltungen und die Operandenplätze einschließlich der Weichen als Register realisiert sind. ***KR-Operatoren*** *sind also Kompositoperatoren, die aus booleschen, d.h. rekursiven Operatoren nach der USB-Methode komponiert sind.* Demzufolge berechnen KR-Operatoren rekursive Funktionen und nur diese. Andrerseits ist jede rekursive Funktion USB-berechenbar und folglich auch KR-berechenbar, m.a.W. für jede rekursive Funktion kann ein KR-Operator für ihre Berechnung angegeben werden. Damit ergibt sich

Satz 4. Die Klasse der KR-berechenbaren Funktionen ist mit der Klasse der rekursiven Funktionen identisch.

Aus Satz 3 und Satz 4 folgt der

5 **Berechenbarkeits-Äquivalenzsatz :** *Statische Berechenbarkeit und rekursive Berechenbarkeit sind einander äquivalent,* oder anders ausgedrückt: *Die Klasse der* ***statisch berechenbaren*** *Funktionen ist mit der Klasse der* ***rekursiven*** *Funktionen identisch.*

Dies ist das Ergebnis einer langen Schlusskette, die bis in das Kapitel 8 zurückreicht. Wir wollen die Aussagen noch einmal Revue passieren lassen, die den Weg markieren. Dabei werden wir das Wort “Funktion” i.Allg. den Wörtern Operator und Operation vorziehen. Als USB-Funktionen hatten wir Funktionen bezeichnet, die nach der USB-Methode komponiert sind.

(1) Rekursive Funktionen sind USB-Funtionen [8.26], denn die rekursiven Komponierungsmittel (funktionale Substitution, Selektion, rekursive Iteration, Minimalisierung) sind mittels der USB-Methode beschreibbar (siehe Kap.8.4.5).

6 (2) Die aus rekursiven Operationen komponierbaren USB-Funktionen sind rekursive Funktionen [8.29], denn die Komponierungsmittel der USB-Methode (die Flussknoten) sind rekursiv beschreibbar, und nichtwohlstrukturierte Operatorennetze lassen sich in wohlstrukturierte überführen [8.28] (siehe Kap.8.4.5).

(3) Folglich sind USB-Funktionen rekursive Funktionen, denn sie werden aus den elementaren booleschen Funktionen komponiert [9.4], also aus rekursiven Funktionen [9.17].

(4) Die Klasse der USB-Funktionen ist mit der Klasse der rekursiven Funktionen identisch (als Folge von (1) und (3)).

(5) Die Klasse der KR-Funktionen ist mit der Klasse der USB-Funktionen identisch, denn KR-Funktionen werden aus den gleichen elementaren Funktionen mit Hilfe der gleichen Komponierungsmittel komponiert wie USB-Funtionen.
(6) Aus (4) und (5) zusammen mit obigem Satz 1 folgt der Berechenbarkeits-Äquivalenzsatz.

Er wird zur These von CHURCH, wenn "statisch berechenbar" durch "effektiv berechenbar" ersetzt wird. Der Äquivalenzsatz folgt also unmittelbar aus der churchschen These, die behauptet, dass die Klasse der *effektiv berechenbaren* (d.h. irgendwie tatsächlich berechenbaren) Funktionen mit der Klasse der *rekursiven* Funktionen identisch ist. Denn der Äquivalenzsatz schränkt die Klasse der *effektiv* berechenbaren Funktionen auf die *statisch* berechenbaren ein und schließt die mittels dynamischer Codierung berechenbaren Funktionen aus, falls es solche gibt. Doch hat der Äquivalenzsatz den Vorteil, dass er keine Hypothese, sondern unter der sehr allgemeinen Voraussetzung statischer Codierung *ableitbar* ist. Die Voraussetzung ist in traditionellen Computern (Prozessorcomputern) erfüllt. Wieweit sie in biologischen und in zukünftigen technischen informationellen Systemen erfüllt ist bzw. erfüllt sein wird, wissen wir nicht. Insofern bleibt die churchsche These eine Hypothese.

Doch jede Funktion, die durch eine "irgendwie" berechnete Wertetafel festgelegt wird, ist eine rekursive Funktion, unabhängig davon, ob das berechnende informationelle System mit statischer oder dynamischer Codierung arbeitet. Denn da die Wertetafel berechnet worden ist, kann sie nicht unendlich sein. Folglich ist sie in eine Kombinationsschaltung überführbar[9.16] , das heißt, sie ist eine rekursive Funktion [9.18].

Hinsichtlich des Exaktheitsanspruchs der Herleitung des Äquivalenzsatzes gilt auch hier die Schlussbemerkung von Kapitel 8. Die Herleitung stellt keinen strengen, mathematischen Beweis dar, doch ist sie logisch folgerichtig.

Die obigen 6 Aussagen können durch zwei weitere ergänzt werden, die schon jetzt *voraussagen*, zu welchem Ergebnis unsere weiteren Bemühungen führen werden. Unser Ziel ist der Prozessorcomputer. Er soll mit statischer Codierung arbeiten. Das hat zwei Konsequenzen. Zum einen folgt aus obigem Satz 1 die Aussage

(7) Der Prozessorcomputer und ist ein KR-Netz oder in ein solches überführbar.

Zum anderen folgt aus dem Berechenbarkeits-Äquivalenzsatz die Aussage

(8) Der Prozessorcomputer und speziell der Von-Neumann-Rechner kann alle rekursiven Funktionen berechnen und nur diese.

Dieser Schluss wird in Kap.13.7 auf anderem Wege bestätigt.

Unser Ziel ist der "universelle" Computer, der *sämtliche* rekursiven Funktionen berechnen kann. Wir müssen also einen "*programmierbaren*" Rechner entwerfen, der sich für die Berechnung jeder rekursiven Funktion konditionieren (programmieren) lässt. Das ist unser nächstes Ziel. Damit kehren wir zum eigentlichen Thema des Kapitels 13 zurück, zur Realisierung der dritten Grundidee des maschinellen Rechnens, der Programmsteuerung.

13.4 Taschenrechner

Zur Verwirklichung der dritten Grundidee des maschinellen Rechnens, der Programmsteuerung, wird ein Steueroperator benötigt, der *Programme interpretieren* kann; wir nennen ihn **interpretierenden Steueroperator**. Bevor wir darangehen, ihn zu entwickeln, wollen wir uns überlegen, wie ein einfacher, *nicht*programmierbarer Taschenrechner aufgebaut ist, der in der Lage ist, eine begrenzte Anzahl arithmetischer Operationen, zumindest die vier Grundrechnungsarten auszuführen. Durch Druck der entsprechenden Funktionstaste soll die gewünschte Funktion (Operation) aufgerufen, d.h. die Operationsausführung ausgeführt werden.

Unser Taschenrechner muss über Codeumsetzer zur Umcodierung zwischen dezimaler und binärer Zahlendarstellung verfügen, über Register zur Abspeicherung der Zahlen (der Argument- und Funktionswerte) und Operatoren, die den Argumentwerten die Funktionswerte zuordnen. Die *Arbeitsoberfläche* (Ein- und Ausgabemittel) ist seit langem standardisiert, sodass viele Leser mit ihr vertraut sein werden.

Zur Realisierung der Operatoren bieten sich zwei Möglichkeiten an:

1. Für jede Funktion wird eine Kombinationsschaltung entworfen.
2. Es wird ein einziger variabler KR-Operator und für jede Funktion ein Steueroperator entworfen, der die Tore des KR-Operators entsprechend steuert.

In beiden Fällen kann der Schaltungsaufwand dadurch verringert werden, dass *variable* Kombinationsschaltungen eingesetzt werden.

Bild 13.2 zeigt einen KR-Operator einschließlich Steueroperator für die Multiplikation durch iterative Addition. Der Addierer `ADD` ist ein Paralleladdierer. Der Übersichtlichkeit halber sind die Register bis auf den Taktverzögerer D nicht eingezeichnet. Der Multiplikator *a* wird dem Steueroperator, der Multiplikand *b* dem KR-Operator eingegeben. Dieser führt *a*-mal eine Addition mit *b* aus. Im ersten Schritt wird *b* zu 0 addiert.

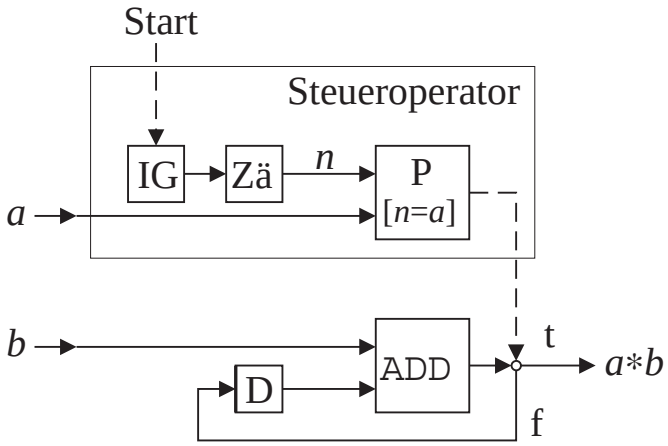


Bild 13.2 Multiplizierer als iterativer Addierer

Der Steueroperator enthält einen Impulsgenerator (IG), einen Zähler (Zä) und einen Vergleichsoperator, der mit P bezeichnet ist, um anzuzeigen, dass es sich um einen Prädikatoperator handelt[4]. Der Impulsgenerator generiert Taktimpulse, die der Zähler zählt, beginnend

mit dem Start der Multiplikation. In jedem Takt wird eine Addition ausgeführt. Der Zähler gibt also die Iterationszahl aus, die mit *n* bezeichnet ist. Der nachfolgende Vergleichsoperator vergleicht *n* mit *a* und gibt das erforderliche Steuersignal zur Steuerung der Zweigeweiche aus. Solange das Prädikat [*n=a*] nicht erfüllt ist, wird die vom Addierer berechnete Summe auf den Eingang des Addierers zurückgegeben. Sobald das Prädikat erfüllt ist, wird die Iteration beendet und die Summe, also das Produkt $a*b$, ausgegeben. (Es wird angenommen, dass *a* ganzzahlig ist; Verallgemeinerung auf Dezimalzahlen bereitet keine Schwierigkeiten.) Der Vergleichsoperator hat also das Prädikat [*n=a*] zu *entscheiden*.

KR-Operator und Steueroperator sind Kompositoperatoren der zweiten Komponierungsstufe. Die erste Stufe beinhaltet die Komponierung von Kombinationsschaltungen, die zweite die Komponierung eines KR-Operators. Der Steueroperator seinerseits benötigt keinen übergeordneten Steueroperator, sondern lediglich einen Eingang für den Startimpuls, der durch Druck auf die Multiplikationstaste generiert wird.

Wenn der Wert von *a* sehr hoch ist, kann die Multiplikation einige Zeit in Anspruch nehmen. In Bild 13.3 ist eine effektivere Variante gezeigt. Ihre Arbeitsweise beruht auf der Methode, die man in der Schule gelernt hat, nach der zwei Zahlen schriftlich multipliziert werden. Der Faktor *a* ist in dem Parallelregister R1 und *b* ist in dem Schieberegister R2 eingespeichert, wobei das letzte (rechte) Bit der Kette der ersten (höchsten) Stelle von *b* entsprechen muss, sodass sie als erste vom Schieberegister ausgegeben (“eingelesen”) wird, wie es bei Taschenrechnern üblich ist. Das Einlesen in R2 kann auch parallel erfolgen. Wenn der Leser die Arbeitsweise nachvollziehen möchte, muss er sich zweierlei klarmachen.

1. Nach dem Schulalgorithmus wird der erste Faktor jeweils mit einer Stelle des zweiten Faktors multipliziert, beginnend mit der ersten Stelle. Diese stellenweise Multiplikation führt der AND-Operator durch. Die Unterstreichung bedeutet, dass der Operator aus so vielen elementaren AND-Operatoren besteht, wie die Wortlänge des Taschenrechners angibt. Dass der AND-Operator die Multiplikation von *a* mit jeweils einer Stelle von *b* ausführt, ist leicht einzusehen. Wenn nämlich die laufende Stelle des zweiten Faktors (das laufende, d.h. zuletzt aus dem Schieberegister R2 herausgeschobene Bit der Bitkette, die den Faktor *b* codiert) den Wert 1 besitzt, muss der AND-Operator die Bitkette des ersten Faktors (*a*) liefern, andernfalls eine Kette von Nullen. Die Bits des Resultats der Stellenmultiplikation ergeben sich demnach aus der Konjunktion der jeweiligen Bits der Bitketten der Faktoren.

2. Nach dem Schulalgorithmus werden die so berechneten Ergebnisse (die Ausgaben des AND-Operators) untereinander, aber jeweils um eine Stelle nach rechts verschoben aufgeschrieben und dann addiert. Die Verschiebung um eine Stelle entspricht bei Dezimalzahlen einer Multiplikation mit 10, bei Binärzahlen einer

---

4 Operatoren, die Prädikate entscheiden, hatten wir in Kap.8.4.5 Prädikatoperatoren genannt.

Multiplikation mit 2, in beiden Fällen also dem Anhängen einer 0. Die Multiplikation wird in der Rückkopplungsschleife ausgeführt. In Bild 13.3 sind der Steueroperator und einige weitere Details des KR-Operators unterschlagen, um nicht vom Wesentlichen abzulenken.

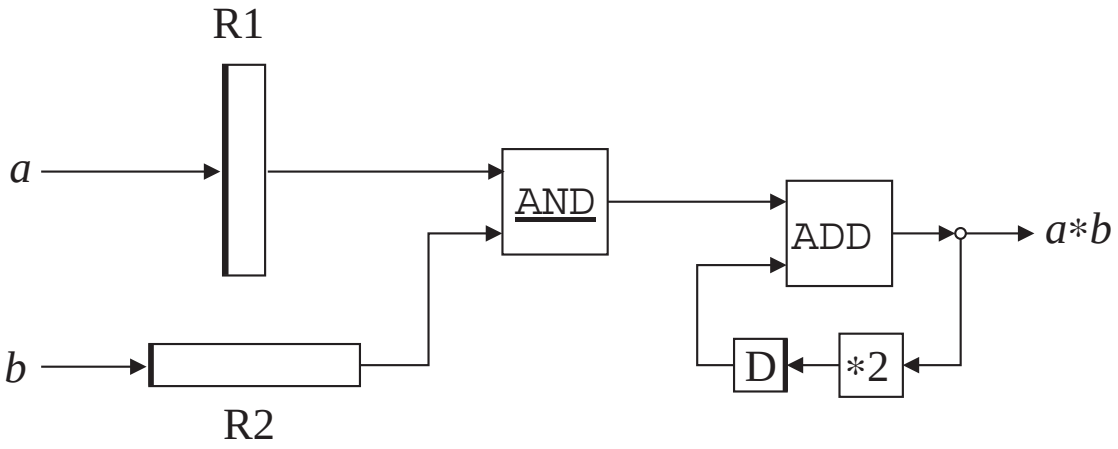


Bild 13.3 Multiplizierer mit Stellenverschiebung

Wir wollen uns überlegen, wie der Steueroperator entworfen und realisiert werden kann. Dazu greifen wir auf Kap.12.3.4 zurück, wo wir uns eine Methode für den Entwurf eines Steueroperators zur Steuerung einer Waschmaschine ausgedacht haben [12.5]. Danach ist zunächst die Entscheidungstabelle des Steueroperators aufzuschreiben und anschließend als ROM zu realisieren. Im Falle des Multiplizierers von Bild 13.2 genügt es, in die Bedingungsspalte der Entscheidungstabelle lediglich die laufende Taktnummer einzutragen, denn der Multiplizierer arbeitet getaktet und der Steueroperator empfängt keinerlei Rückmeldungen vom KR-Operator, sodass die Bausteinoperationen, die der Reihe nach auszuführen sind, nur von der laufenden Taktnummer abhängen. In die Aktionsspalte sind die Steuerwörter einzutragen. Ein Steuerwort enthält die Steuersignale für sämtliche (auch die nicht eingezeichneten) Eingangstore der Register bzw. Ein-Bit-Speicher in der erforderlichen Reihenfolge. Die Entscheidungstabelle, die sich so ergibt, stellt eine Sonderform dar. Es braucht nämlich nicht in jedem Schritt nach der Zeile mit der aktuellen Bedingung *gesucht* zu werden, sondern die Zeilen werden der Reihe nach abgearbeitet.

Es gibt noch viele andere Möglichkeiten, einen Multiplizierer zu entwerfen. Beispielsweise kann der Addierer sequenziell arbeiten. Arbeits- und Steueroperator sind dann Kompositoperatoren der dritten Stufe. Der Steueroperator des Addieres ist dem des Multiplizierers untergeordnet, sodass sich eine *Steuerhierarchie* ergibt.

Auf analoge Weise wie der Multiplizierer lässt sich jede andere Operation realisieren, die unser Taschenrechner “können” soll, d.h. für die er über eine Funktionstaste verfügen soll. Dabei lassen sich sowohl die Arbeitsoperatoren (die Opera-

toren des KR-Netzes) als auch die Steueroperatoren als ROM realisieren. Der Taschenrechner stellt dann eine *ROM-Hierarchie* dar.

13.5 Prozessor

13.5.1 Idee des Prozessors und seiner Programmierung

Da sich für jede berechenbare Funktion ein gesteuerter KR-Operator angeben lässt, könnte man auf die Idee kommen, einen universellen Rechner dadurch zu realisieren, dass man einen universellen KR-Operator und zahllose Steueroperatoren baut. Die Idee stößt auf dieselbe Grenze wie die Idee von den zahllosen Kombinationsschaltungen, die wir bereits in Kap.9.2.1 verworfen hatten. Ihre Verwirklichung ist nur dann sinnvoll, wenn die Anzahl der zu berechnenden Funktionen gering ist, wie im Falle eines Taschenrechners oder eines Steuerrechners, der ein hinsichtlich der Steuerung relativ unkompliziertes Objekt steuert, z.B. eine Waschmaschine, einen Fotoapparat oder einen Automotor.

Eine andere Idee besteht darin, nicht eine Menge, sondern eine Hierarchie von Steueroperatoren zu entwerfen. Dieser Weg war hinsichtlich des Taschenrechners angedeutet worden. Auch er ist möglich, aber wiederum nur für Spezialfälle. Beispielsweise kann es zweckmäßig sein, einen Spezialrechner als ROM-Hierarchie zu 7
konzipieren, wenn er Funktionen berechnen soll, die oft auftreten, aber zu kompliziert sind, um das einfache Taschenrechnerprinzip anwenden zu können. Diese Situation ist z.B. für statistische Auswertungen charakteristisch. Aber mit derartigen *reinen Hardwarelösungen* ist kein universeller Rechner zu bauen, sodass nach einer *Softwarelösung* gesucht werden muss. Gesucht ist ein universeller Rechner in Form eines gesteuerten KR-Operators, der beliebige Operationsvorschriften (Vorschriften zur Berechnung beliebiger rekursiver Funktionen) ausführen kann; gesucht ist ein per Programm beliebig steuerbarer oder *frei programmierbarer KR-Operator*.

Historisch führte der Weg zum programmierbaren Rechner über viele Ideen und realisierte Schaltungen. Relativ schnell hat sich eine Schaltung durchgesetzt, die nachträglich **Von-Neumann-Rechner** genannt worden ist. Sie besteht im Wesentlichen aus einem **Prozessor** und dessen **Arbeitsspeicher**, auch **Hauptspeicher** genannt (siehe Bild 13.7). In seiner einfachsten Form ist der Prozessor ein KR-Netz mit einem einzigen Bausteinoperator, einer steuerbaren Kombinationsschaltung. Sie wird **arithmetisch-logische Einheit** oder kurz **ALU** genannt (U für unit).

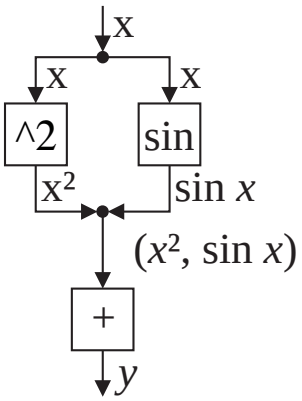
Eine ALU kann einige sehr einfache arithmetische Operationen wie Inkrementieren, Dekrementieren, Addieren und Subtrahieren ausführen, ferner Verschiebungen von Bitketten nach rechts und links sowie einige boolesche Operationen. Die Einstellung ("*Konditionierung*") auf eine bestimmte Funktion erfolgt durch Steuerung von Weichen, die in die Kombinationsschaltung eingebaut sind.

Die Beschränkung auf die ALU als einzigen Bausteinoperator des KR-Operators bedeutet den Übergang von *verteilter Verarbeitung* durch mehrere Operatoren eines

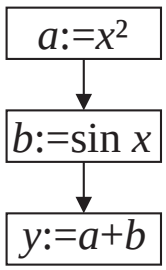
Operatorennetzes zur *zentralen Verarbeitung* durch einen einzigen Arbeitsoperator, die ALU, und in dem damit verbundenen Übergang von *verteilter* zu *zentraler Speicherung*. Aus den *lokalen* Operandenspeichern, die über ein KR-Netz *verteilt* sind, werden **Speicherplätze** des Hauptspeichers. Wir erinnern uns, dass die gedankliche Zusammenfassung der Speicherplätze eines Operatorennetzes zu einer einzigen Speichereinheit der Idee des *endlichen Automaten* zugrunde liegt (vgl. Kap.8.2.3 [8.10]). Mit der fundamentalen Bedeutung der Zentralisierung für die Rechentechnik, insbesondere für die Programmierungstechnik werden wir uns in Kap. 13.7 beschäftigen. Im Augenblick interessieren wir uns für ihre Konsequenzen hinsichtlich der Funktionsweise des Prozessors, den wir entwerfen wollen.

Die Beschreibung einer Kompositoperation mittels eines Operandenflussplanes verliert ihren eigentlichen Sinn, da es auf der Ebene der Programmierung des Prozessors kein Netz aus mehreren Operatoren gibt, zwischen denen die Operanden
8 fließen könnten. Vielmehr muss dem Prozessor mitgeteilt werden, welche *Aktion* die ALU als nächste auszuführen hat, also welche Operation mit welchen Operanden. Ein Programm, das eine Folge von Aktionen vorschreibt, nennen wir **Aktionsfolgeprogramm**. Es besteht aus einer Folge von **Befehlen**, je ein Befehl für jede Aktion.

Die graphische Darstellung eines Aktionsfolgeprogramms nennen wir **Aktionsfolgeplan**. Ein Aktionsfolge*programm* ist also die maschinenverständliche sprachliche Artikulierung eines Aktionsfolge*plans*, m.a.W. es ist ein maschinenverständli-



(a)



(b)

Bild 13.4 Graphische Darstellung der Berechnung der Funktion $y = x^2 + \sin x$. (a) - Datenflussplan; (b) - Aktionsfolgeplan. ^2 ist als “hoch 2" zu lesen.

9 cher imperativer Algorithmus. Diesem Sachverhalt entspricht der Sprachgebrauch der Informatiker, wonach Aktionsfolgeprogramme als **imperative Programme** bezeichnet werden[5]. Bild 13.4b zeigt den Aktionsfolgeplan zur Berechnung der Funktion $y = x^2 + \sin x$. Die Reihenfolge der ersten beiden Befehle kann umgekehrt werden. Zum Vergleich ist in Bild 13.4a der entsprechende Operandenflussplan

dargestellt. Man beachte die unterschiedliche Bedeutung der Pfeile. Ein Pfeil des Operandenflussplans stellt den Übergabeweg eines Operanden dar; im Aktionsfolgeplan zeigt er auf die nächste Aktion.

Ein charakteristisches Problem der Aktionsfolgeprogrammierung (der imperativen Programmierung) ist der Operandentransport. Infolge der Zentralisierung der Speicherung müssen die Operanden der Bausteinoperationen (der Operationen der ALU) von und zum zentralen Speicher transportiert werden. Ein Steueroperator, der Bestandteil des Prozessors ist, hat dafür zu sorgen, dass die Befehle in der richtigen Reihenfolge ausgeführt und die dabei notwendigen Operandentransporte ausgeführt werden. Wenn wir darangehen einen Prozessor zu entwerfen, müssen wir also unser Augenmerk darauf richten, dass der Steueroperator eine sequenzielle Aktionsfolge steuert und dass er bei jeder Aktion folgende *Aktionsschritte* auszuführen hat:

1. Holen des aktuellen Befehls,
2. Konditionieren der ALU (Einstellung auf die konkrete Operation),
3. Versorgen der ALU mit den aktuellen Operanden und Berechnen des Resultats.
4. Abspeichern des Resultats.
5. Berechnung der nächsten Befehlsadresse

Diese Folge wird aus der abstrakten Sicht des Computerentwurfs **von-neumannsches Operationsprinzip** und aus der konkreten Sicht der internen Semantik (der Prozesse im Computer) **zentrale Steuerschleife** genannt. Der fünfte Aktionsschritt ist der Vollständigkeit halber hinzugefügt. Er wird weiter unten besprochen. In Kap.16.5 [16.14] wird ein sechster Aktionsschritt eingeführt, welcher der Behandlung von Unterbrechungen dient.

Es mag überraschen, dass die zentrale Steuerschleife keinen speziellen Aktionsschritt für die eigentliche Operationsausführung enthält, für die Zuordnung des Resultatwertes zu den Argumentwerten. Der Grund ist folgender. Die Zuordnung wird von der ALU ausgeführt und diese ist eine Kombinationsschaltung, besitzt also keinen Speicher. Der Zuordnungsprozess ist ein Übergangsprozess in der ALU und enthält als solcher keinen Zeitpunkt der kausaldiskreten Prozessbeschreibung (siehe Kap.9.4 [9.2]). Die Zuordnung erfolgt, sobald die Operanden am ALU-Eingang liegen und gehört zum Aktionsschritt 3.

Damit der Prozessor einen Befehl holen und ausführen kann, müssen ihm die **Adressen** des Befehls, der Operanden und des Resultats sowie die auszuführende Operation mitgeteilt werden. Diese Angaben müssen in jedem Befehl enthalten sein, entweder explizit oder implizit (aus den expliziten Angaben ableitbar). Dabei wird vorausgesetzt, dass der Steueroperator auf die Speicherplätze eines Arbeitsspeichers über Adressen zugreifen kann. Außerdem muss jeder Befehl ein Codewort für die auszuführende ALU-Operation enthalten, den sog. **Operationscode** (**OC**).

---

5 Mit der Einführung des Begriffs des imperativen Algorithmus in Kap.7.2 [7.10] sollte der Begriff des imperativen Programms vorbereitet werden.

An dieser Stelle sei eine Bemerkung zur Speicherorganisation eingeschoben. Ein Computer hantiert mit zwei Objektklassen, mit Programmen (sprachlichen Operatoren) und mit Daten (Operanden technischer sprachlicher Operatoren)[6], sodass es naheliegt, ihn mit zwei Speichern auszurüsten, einem Programmspeicher und einem Datenspeicher. Der ungarisch-amerikanische Mathematiker und Physiker JOHN VON NEUMANN hat vorgeschlagen, Befehle und Daten in einem einzigen Speicher aufzubewahren und einheitlich zu behandeln. Genauer gesagt hat von Neumann mit seiner Autorität diesem, später nach ihm benannten Speicherprinzip zur allgemeinen Anerkennung und zum technischen Durchbruch verholfen. Ursprünglich stammte die Idee von J. PRESPER ECKERT und JOHN W. MAUCHLY[7].

Die gemeinsame Speicherung von Befehlen und Daten hat zur Folge, dass in einem Befehl anstelle einer Operandenadresse eine Befehlsadresse auftreten kann, sodass eventuell die *Be*arbeitung eines Befehls veranlasst wird (nicht die *Ab*arbeitung, d.h. Ausführung). Das bedeutet eine Relativierung der begrifflichen Unterscheidung zwischen Programmen und Daten oder zwischen Operatoren und Operanden. Wir hatten uns schon früher davon überzeugt, dass die Unterscheidung nicht konsequent durchführbar ist, und es sinnvoll sein kann, einen gemeinsamen Oberbegriff zu bilden, wie es im Lambda-Kalkül gehandhabt wird. Diese generalisierende Abstraktion liegt auch dem von-neumannschen Vorschlag zugrunde.

Wir setzen den unterbrochenen Gedankengang fort und ziehen folgenden Schluss. Damit die genannten Aktionsschritte der zentralen Steuerschleife ausgeführt werden können, muss die ALU in ein KR-Netz eingebettet werden, in dem die erforderlichen Befehls- und Datentransporte stattfinden. Dieses Netz nennen wir **RALU** (**R**egister und **ALU**). Die Weichen der RALU sind von einem Steueroperator zu steuern. Die gedankliche Vereinigung der RALU mit dem Steueroperator führt zum Begriff des **Prozessors** (siehe Bild 13.7)[8].

Bevor wir damit beginnen, die Schaltung eines Prozessors zu entwerfen, überlegen wir uns, wie die Sprache etwa auszusehen hat, die der Prozessor verstehen (interpretieren), d.h. in Steuersignale umsetzen soll. Die Sprache muss die Möglichkeit bieten, maschinenlesbare Aktionsfolgen, also maschinenlesbare *imperative Algorithmen*, d.h. *imperative Programme* zu artikulieren. Eine solche Sprache heißt **imperative Sprache**.

---

6 In einem verallgemeinerten Sinn werden als Daten häufig beliebige (auch beliebig lange) Bitketten bezeichnet, die in Rechnern oder Rechnernetzen transportiert werden.

7 In diesem Zusammenhang ist der Artikel [Bauer 98] sehr aufschlussreich.

8 Die Komponierung des Prozessors aus RALU und Steueroperator entspricht nicht unbedingt den Darstellungen in der Literatur; die Struktur eines Prozessors kann komplizierter sein. Wir begnügen uns mit der sehr einfachen in Bild 13.7 dargestellten Struktur. Sie reicht aus, um die Arbeitsweise eines Prozessors im Prinzip zu verstehen.

13.5.2 Maschinensprache

Damit der zu entwerfende Prozessor einen imperativen Algorithmus ausführen kann, muss dieser in einer Sprache geschrieben sein, die der Prozessor "versteht". Wir nennen sie **Prozessorsprache**. Das lässt sich dadurch erreichen, dass der Aufbau der Befehle, aus denen der Algorithmus besteht, standardisiert und der Prozessor mit einem speziellen Register für die Aufnahme eines standardisierten Befehls ausgerüstet wird. Dieses Register heißt **Befehlsregister**, abgekürzt **BR**. Die Standardisierung schreibt vor, in welcher Reihenfolge die Bestandteile eines Befehls, also der Operationscode und die Adressen zu einem Binärwort (i.d.R. identisch mit dem *Rechnerwort*) zu verketten sind, man spricht von *Befehlsformatierung*. Dem **Befehlsformat** muss der Aufbau (das "Format") des Befehlsregisters genau entsprechen.

Diese *Formatentsprechung* ermöglicht durch sequenzielles Laden der Befehle eines Programms in das Befehlsregister die *direkte* Abarbeitung; "direkt" bedeutet hier: ohne vorherige Übersetzung in eine andere Sprache oder sonstige Bearbeitung. Durch die Formatentsprechung wird die Sprache an die Schaltung "angekoppelt". Man kann auch hier, ähnlich wie im Falle des Wort-Leitung- und Leitung-Wort-Zuordners, von *Schnittstelle* oder *Interface* zwischen Schaltung und Sprache, zwischen Hardware und Software sprechen.

Eventuell muss zwischen Prozessorsprache und Maschinensprache unterschieden werden. Wir vereinbaren: *Eine Programmiersprache, die ein direktes Interface mit einem Computer besitzt, heißt **Maschinensprache** des Computers. Ein Programm, das in einer Maschinensprache geschrieben ist, heißt **Maschinenprogramm***. Die Maschinensprache eines Computers, der einen einzigen Prozessor enthält, ist mit der Prozessorsprache identisch. Soweit von Einprozessorrechnern die Rede ist, können die Wörter *Maschinensprache* und *Prozessorsprache* als Synonyme verwendet werden.

Neben dem Befehlsformat muss auch das **Programmformat** festgelegt werden, d.h. die genaue Anordnung der Befehle innerhalb eines Programms. Wenn ein **Operationsrepertoire** (die Menge der zur Verfügung stehenden Operationscodes), ein Befehlsformat und ein Programmformat vorgegeben sind, ist damit die *Syntax* einer Maschinensprache festgelegt. Man beachte, dass mit der Syntax auch die interne Semantik der Maschinensprache festgelegt ist. Denn aus der Schaltung der Maschine ergibt sich zwangsläufig die interne Semantik eines Befehls bzw. eines Programms, d.h. der Prozess, der bei der Ausführung (Interpretation) des Befehls bzw. Programms in der Maschine abläuft.

Im vorangehenden Kapitel hatten wir uns überlegt, welche Informationen ein Befehl enthalten muss, damit der Prozessor obige Aktionsschritte ausführen kann. Auf dieser Grundlage wollen wir eine Maschinensprache entwerfen.[9] Dazu legen wir fest:

---

9 Es sei an die Bemerkung am Ende des Kapitels 13.1 erinnert, dass die Darlegungen dieses

1. Ein Befehl besteht aus 5 Feldern bestimmter Länge. In die Felder werden der Reihe nach die Befehlsadresse (BA), der Operationscode (OC), die Adressen der Operanden (A1 und A2) und die Adresse des Resultats (A3) eingetragen (siehe die oberste Zeile in Bild 13.5).
2. Ein Programm ist eine Tabelle, deren Zeilen je einen Befehl und deren Spalten jeweils die gleichen Elemente der Befehle enthalten.
3. Das Codewort der Addition ist `ADD`, das der Subtraktion `SUB`.

Damit ist eine Tabellensprache für eine **Vier-Adress-Maschine** definiert, d.h. für einen Rechner, der pro Aktion mit bis zu 4 Adressen hantieren kann. Bild 13.5 zeigt ein Programm, das in dieser Sprache geschrieben ist für die Berechnung des Wertes von *r* nach der Formel (Ergibtanweisung)

$$r := (a - b) + c \tag{13.1}$$

Dabei sind den Variablen in (13.1) Speicheradressen zugewiesen, beispielsweise der Variablen *a* die Adresse 3010 (siehe den Kommentar in Bild 13.5).

Das Programm besteht aus zwei Zeilen. Um das Lesen zu erleichtern, ist in der Kopfzeile (sie gehört nicht zum Programm) angegeben, welche Befehlskomponenten

```
 BA     OC     A1      A2      A3
_____________________________________
3000    SUB   3010    3011    3012;
3001    ADD   3012    3013    3014;
```

Kommentar:

```
3000 - Adresse des ersten Befehls, Startadresse
3001   Adresse des zweiten Befehls
3010 - Adresse von a
3011 - Adresse von b
3012 - Adresse von d
3013 - Adresse von c
3014 - Adresse von r
```

Bild 13.5 Maschinenprogramm einer Vier-Adress-Maschine zur Berechnung des durch (13.1) festgelegten Ausdrucks. BA - Befehlsadresse, OC - Operationscode, A1, A2 - Operandenadressen, A3 - Resultatadresse.

Kapitels auf den Fachmann eventuell einen antiquierten Eindruck machen. Wenn im Weiteren verschiedene Maschinensprachen vorgeschlagen werden, kommt es darauf an, die Prinzipien, nach welchen Prozessoren arbeiten und programmiert werden, deutlich erkennbar zu machen. Ob der Prozessor, den wir erfinden werden, in genau der beschriebenen Form tatsächlich existiert, ist nebensächlich.

in den einzelnen Spalten eingetragen sind. Auch die Abstände zwischen den Feldern sind wegen der besseren Lesbarkeit eingefügt, obwohl sie nicht zum Programmtext gehören. Die Adressen sind als Dezimalzahlen angegeben. Das setzt voraus, dass der Rechner, der das Programm ausführen soll, über einen Codeumsetzer verfügt, der Dezimalzahlen in Dualzahlen umcodiert. Auch die Operationscodes müssen umcodiert, d.h. in die entsprechenden vorgeschriebenen Bitketten überführt werden.

Die Abarbeitung des Programms beginnt mit dem "Holen" des ersten Befehls d.h. mit dessen Transport aus der HS-Zelle (Speicherplatz des Hauptspeichers) mit der Adresse 3000 in das Befehlsregister BR[10]. Aus dem Kommentar ergibt sich, dass in der ersten Aktion die Differenz *a-b* berechnet werden soll. Zu diesem Zweck müssen zunächst die Werte von *a* und *b* aus den Speicherzellen 3010 bzw. 3011 geholt und in zwei weitere reservierte Register geladen werden, in ein Datenregister (DR) und in ein Register, das wir aus später erkennbaren Gründen *Akkumulator* (AC) nennen. Von da werden sie der ALU zugeführt, die vorher auf Subtraktion konditioniert werden muss. Schließlich wird das Ausgabewort der ALU unter der HS-Adresse 3012 abgespeichert.

Die Ausführung des zweiten Befehls verläuft analog. Über den Speicherplatz mit der Adresse 3012 wird der Wert der Differenz *a-b* an die zweite Aktion übergeben. Diesem Speicherplatz entspricht keine der drei Variablen in (13.1). Die dort gespei- 11
cherte Variable stellt eine *Hilfsvariable* dar; wir bezeichnen sie mit *d*. Im Auftreten der Hilfsvariablen *d* spiegelt sich das Grundprinzip des imperativen Programmierens wider. Der Leser erinnere sich an die Definition des imperativen Algorithmus in Kap.7.2 [7.10]. Dort war anhand des gleichen Rechenbeispiels zuerst der Begriff der *Aktion* als Operation an explizit angegebenen Operanden eingeführt worden und anschließend der Begriff des *imperativen Algorithmus* als Sequenz von Imperativsätzen, wobei je ein Imperativsatz eine Aktion vorschreibt. Man beachte, dass die beiden Befehle in Bild 13.5 die beiden Berechnungsschritte darstellen, in die wir den Ausdruck *a-b+c* in Kap. 7.2 [7.11] zerlegt hatten.

Unser Programm kann jedem Argumentwertetripel (*a,b,c*) einen Funktionswert *r* zuordnen, es ist ein formaler Operator, der die durch (13.1) festgelegte Funktion berechnet. Die gewünschten Argumentwerte müssen vor dem Start des Programms unter den betreffenden Adressen eingespeichert werden (s.u.). Die beschriebene Vorgehensweise ist ziemlich umständlich und die Frage ist berechtigt, warum man ein Programm schreiben soll, um eine Subtraktion und eine Addition auszuführen. Mit einem Taschenrechner ist die Berechnung einfacher und ökonomischer, jedenfalls wenn man nur wenige Funktionswerte berechnen will. Muss man die Operation

---

10 Die folgenden Überlegungen bilden die Grundlage für den Entwurf des Prozessors einer Zwei-Adress-Maschine, dessen Schaltung in Bild 13.7 gezeigt ist. Denjenigen Lesern, die gerne Denksportaufgaben lösen, wird vorgeschlagen, sich selber eine Schaltung auszudenken. Wer davon Abstand nehmen will, kann vorblättern und den Prozess der Befehlsausführung anhand des Bildes 13.7 verfolgen.

jedoch hundert- oder gar tausendmal ausführen, so kann sich das Programmieren schon lohnen, zumal es viele Mittel gibt, das Programmieren zu vereinfachen.

Eine bedeutende Erleichterung, die bereits der Taschenrechner gewährt, besteht in der Möglichkeit, mit dem Ergebnis der Subtraktion sofort weiterzurechnen, ohne das Zwischenresultat explizit abzuspeichern. Die Methode, die das ermöglicht, ist uns bekannt; sie heißt *Rückkopplung*. Im vorliegenden Fall besteht sie in der Rückführung des Ausganges der ALU auf ihren Eingang. Das Register in der Rückkopplungsschleife ist der bereits genannte **Akkumulator**. Man beachte, dass sich durch den Akkumulator die explizite Angabe von Operanden teilweise erübrigt. Genau genommen liegt hier eine erste Abweichung vom Prinzip der konsequenten Aktionsfolgeprogrammierung vor.

Der Akkumulator bringt mehrere Vorteile. Er verkürzt das Befehlsformat um eine Operandenadresse, er spart den Hauptspeicherplatz für das Zwischenergebnis ein und er verkürzt die Ausführungszeit, da zwei Zugriffe auf den HS entfallen. Er bringt aber auch einen Nachteil. Es ist ein spezieller Befehl für den Transport von Operandenwerten vom HS zum AC notwendig, der immer dann zum Einsatz kommt, wenn die nächste Aktion nicht mit dem im AC aufbewahrten, sondern einem anderen Wert ausgeführt werden soll. Dadurch kann das Programm länger werden.

Das Beispiel zeigt anschaulich, wie eng Sprache und Hardware miteinander verquickt sind. Beide müssen gemeinsam entworfen werden. Dabei ist ein Kompromiss zwischen mehreren Zielen zu schließen:

- niedriger Hardwareaufwand,
- niedriger Programmieraufwand,
- kurze Ausführungszeiten,
- geringer Speicherplatzbedarf.

Das Suchen nach dem optimalen Kompromiss zieht sich durch die gesamte Hardware-, Software- und Sprachentwicklung.

Neben dem Akkumulator hat sich eine zweite Idee zur Verkürzung des Befehlsformats durchgesetzt, die Einführung eines **Befehlszählers** (**BZ**). Sie geht davon aus, dass die Befehle eines Programms der Reihe nach im HS gespeichert werden, sodass der nächste Befehl die nächst höhere Adresse besitzt (vorausgesetzt, dass ein Befehl nicht mehr als eine Speicherzelle beansprucht). Wenn vor dem Start eines Programms dafür gesorgt wird, dass die Adresse des ersten Befehls in den Befehlszähler geladen wird, können die folgenden Befehlsadressen durch laufende Inkrementierung des Inhaltes des Befehlszählers berechnet werden, sodass sich ihre Angabe in den weiteren Befehlen erübrigt. Die Adresse des ersten Befehls, die sogenannte **Startadresse**, wird i.d.R. automatisch vom Computer (genauer vom sog. Betriebssystem; siehe Kap.19.5) beim “Laden” des Programms in einen freien Speicherbereich festgelegt".

Damit enthält ein Befehl neben dem Operationscode nur noch höchstens zwei Adressen, und aus unserer Vier-Adress- wird eine Zwei-Adress-Maschinensprache. In ihr ist das Programm von Bild 13.6 geschrieben, das die durch (13.1) festgelegte

Funktion berechnet. TVS ist der Operationscode für den oben erläuterten **T**ransport **V**om **S**peicher zum AC. Später werden wir auch einen TNS-Befehl benötigen für den **T**ransport **N**ach dem **S**peicher. Der END-Befehl beendet die Abarbeitung des Programms. Der besseren Lesbarkeit halber sind anstelle der Adressen die Variablenbe-

```
BEGIN
   TVS   <a>
   SUB   <b>
   ADD   <c>   <r>
END
```

Bild 13.6 Programm einer Zwei-Adress-Maschine zur Berechnung von *r* gemäß (13.1). Die spitzen Klammern zeigen an, dass im Programm nicht der Bezeichner, sondern die betreffende Adresse zu stehen hat.

zeichner in spitzen Klammern eingetragen. Beispielsweise stellt `<a>` die Adresse dar, unter der die Werte der Variablen *a* abgespeichert werden.

Nach Ausführung des ersten Befehls (Transport vom Speicher) steht im Akkumulator der Wert von *a*, nach der Subtraktion der Wert der Differenz *a-b* und nach der Addition der Wert von *r*, der dann dem HS übergeben wird. Für die Angabe des Speicherplatzes für *r* steht das zweite Adressfeld des Additionsbefehls zur Verfügung. Nach Ausführung des `ADD`-Befehls steht der Wert von *r* sowohl im HS unter der Adresse <r>, als auch im AC, denn ein Register wird durch Auslesen seines Inhalts nicht “geleert”.

13.5.3 Arbeitsweise der RALU

Vielleicht wird mancher Leser schon ungeduldig angesichts aller möglichen scheinbar kaum zur Sache gehörenden Überlegungen, und es wird Zeit, dass wir aufhören, um den universellen Rechner wie die Katze um den heißen Brei herumzuschleichen. Der Brei ist genügend abgekühlt. Die Aufgabe “Entwerfe eine Zwei-Adressmaschine” können wir “in den Mund nehmen” (artikulieren), ohne Gefahr zu laufen, uns zu verbrennen, d.h. an ihr zu scheitern. Die Aufgabe ist inzwischen so scharf umrissen, dass wohl jeder, dem Denksport Spaß macht, nach einigem Probieren eine Schaltung zusammengebastelt haben wird, die das Programm von Bild 13.6 ausführen kann und die der in Abb.13.7 dargestellten Schaltung mehr oder weniger ähnlich ist.

Wer nicht die Muße hat, *seine* Prozessorschaltung zu erfinden, kann die Arbeitsweise des Prozessors anhand der Schaltung von Bild 13.7 nachvollziehen. Doch auch dies kann er unterlassen; für das weitere Verständnis ist es nicht erforderlich. Er muss sich jedoch merken, was genau der Prozessor “tut”. Wir wiederholen es noch einmal. *Er konditioniert die ALU und versorgt sie mit Operanden (was zur Ausführung der*

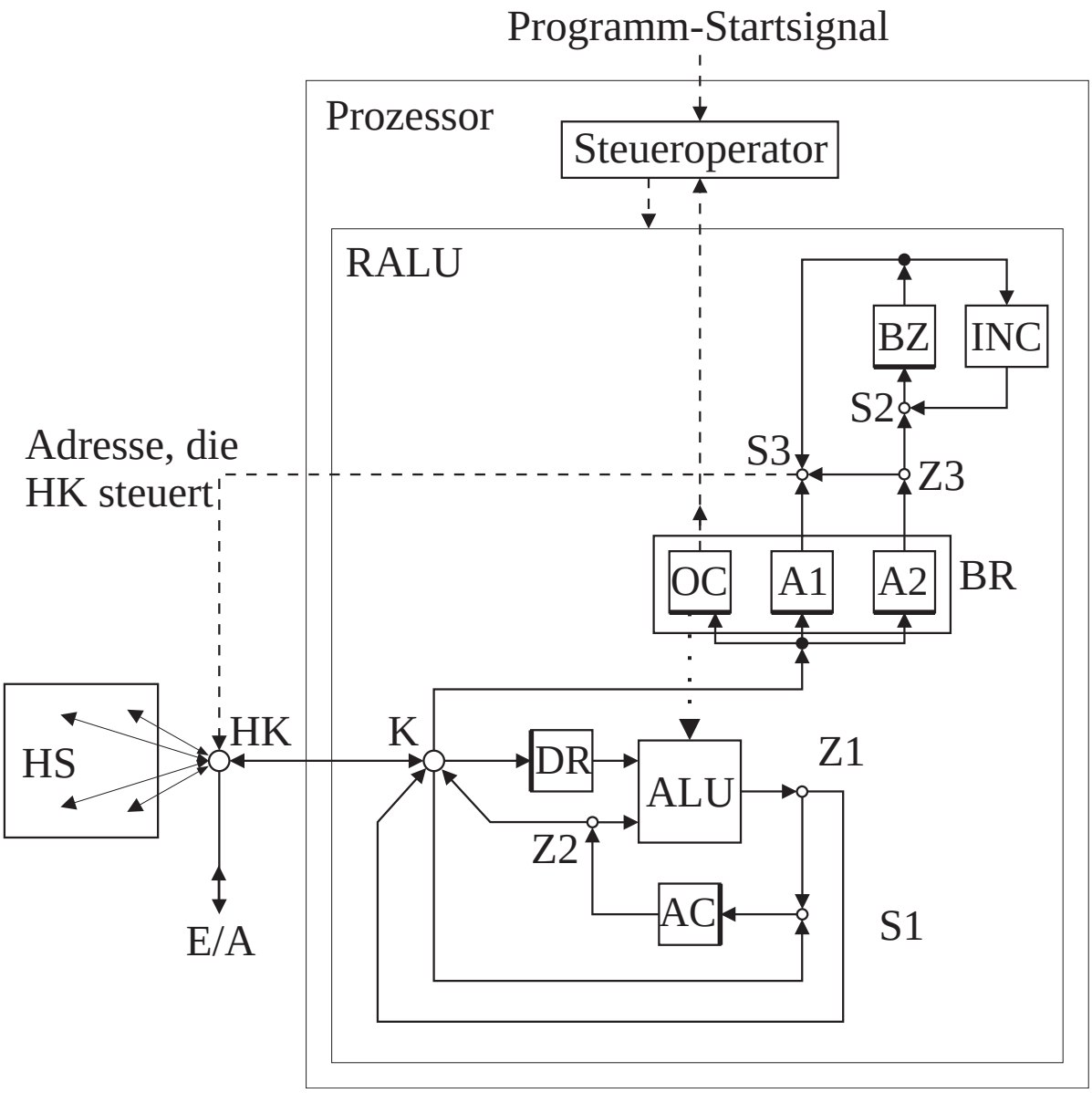


Bild 13.7 Prinzipschaltung eines Einprozessorrechners mit dekomponierter RALU.
Bezeichnungen:

HS - Hauptspeicher
ALU - Arithmetisch-Logische Einheit
RALU - Register+ALU
AC - Akkumulator
DR - Datenregister
BR - Befehlsregister
OC - Register für Operationscode
A - Register für Adresse
BZ - Befehlszähler
INC - Inkrementierer
K - Kommutator
HK - Halbkommutator
S - Sammelweiche
Z - Zweiwegeweiche
E/A - Ein-/Ausgabe

ALU-Operation führt); er bewahrt das Resultat im AC auf und legt es, falls verlangt, im HS ab; und er holt sich den nächsten Befehl.

Für diejenigen Leser, welche die Vorgänge im Prozessor nachzuvollziehen möchten, soll Bild 13.7 näher erläutert werden. Es zeigt die Schaltung eines Einprozessorrechners, dessen RALU in ein KR-Netz dekomponiert ist, das zwei Kombinationsschaltungen (ALU, INC) und vier Register (BZ, BR, AC, DR) enthält. Der HS und - wie wir später sehen werden - auch der Steueroperator lassen sich in KR-Netze dekomponieren. Der ganze Rechner ist also ein KR-Operator. Der Hauptspeicher interessiert hier lediglich unter dem Aspekt der Kommunikation mit dem Prozessor. Sie erfolgt über einen Kommutator (K) [12.2] und einen Halbkommutator (HK; siehe Bild 12.4a). Diese Kommunikationsstruktur ist derjenigen von Bild 12.4b sehr ähnlich. Die Kommunikation mit der Außenwelt erfolgt über den HK. Ein- und Ausgabeeinheiten sind nicht eingezeichnet. Die durchgezogenen Pfeile stellen die Übergabewege der Befehle, der Operanden und der Adressen dar; aus der Sicht des Elektronikers stellen sie elektrische Leitungen (leitende Bahnen auf einem Chip) dar bzw. Leitungsbündel, falls die den Daten entsprechenden Bitketten parallel übertragen werden. Der Adressweg S3-HK ist ausnahmsweise gestrichelt gezeichnet, weil die Adresse die Rolle eines Steuersignals des Halbkommutators HK spielt, sie ist der "Schlüssel", der das "Schloss" des adressierten Speicherplatzes öffnet. Die Wege der Steuersignale zu den Weichen und zum Kommutator K sind nicht einzeln dargestellt, sondern werden gemeinsam durch den gestrichelten Pfeil vom Steueroperator zur RALU symbolisiert. Der Pfeil vom OC-Feld des BR zur ALU ist punktiert gezeichnet, um anzudeuten, dass die ALU durch Übergabe des Operationscodes über den punktierten Pfeil (und Umcodierung in das entsprechende Steuersignal) konditioniert werden *kann*, dass die ALU ihr Steuersignal aber auch vom Steueroperator erhalten kann (der entsprechende Signalweg ist nicht eingezeichnet).

Die Ausführung des im Befehlsregister (BR) befindlichen Befehls beginnt mit der Übergabe (Meldung) des Operationscodes (OC) an den Steueroperator, der daraufhin die Steuersignale in der erforderlichen zeitlichen Reihenfolge und mit den erforderlichen zeitlichen Abständen generiert. Die Signalfolge kann für jeden Befehl ein für allemal festgelegt werden, da bekannt ist, in welcher Reihenfolge die Steueroperationen auszuführen sind, wie viel Zeit die Übergangsprozesse in den Schaltungen beanspruchen und welche zeitlichen Abstände demzufolge zwischen den Steuersignalen einzuhalten sind, um zu gewährleisten, dass die Datenübergaben (Spannungsübergaben zwischen den elektronischen Schaltungen) fehlerfrei erfolgen. Der Steueroperator steuert also die einzelnen Befehlsausführungen autonom (ohne Einfluss von Meldungen). Durch die OC-Meldungen werden die Ausführungen der Befehle eines Programms gestartet, und durch ein Programm-Startsignal wird die Ausführung eines Programms gestartet. Die Startadresse des Programms muss vor oder mit dem Startsignal in den Befehlszähler (BZ) eingetragen werden. In Kap.13.5.5 werden wir uns überlegen, wie sich der Steueroperator schaltungsmäßig realisieren lässt.

1 Aktionsschritt		2 Transfer-Nr.	3 Daten	4 Datenweg
1	Befehl holen	1	Befehlsadresse	BZ - S3 - HK
		2	ADD-Befehl	HS - HK - K - BR
2	ALU konditionieren	3	OC der Addition	OC - ALU
3	Operanden an ALU	4	Adresse von c	A1 - S3 - HK
		5	c	HS - HK - K - DR - ALU
4	Resultat speichern	6	Adresse von r	A2 - Z3 - S3 - HK
		7	r	ALU - Z1 - K - HK - HS
5	Adresse des nächsten Befehls berechnen	8	Befehlsadresse	BZ - INC - S2 - BZ

13.8 Registertransfers beim Holen und Ausführen des Additionsbefehls des Programms von Bild 13.6. Bezeichnungen siehe Bild 13.7. In der Spalte "Daten" ist dasjenige Datum (Adressen, Operationscode, Operand) angegeben, das transferiert wird.

Eine weitere Unterstützung beim Nachvollziehen der Programmabarbeitung gibt die Tabelle von Bild 13.8. In ihr sind alle Datenübergaben in zeitlicher Reihenfolge aufgelistet, die während der Ausführung des Additionsbefehls aus dem Programm von Bild 13.6 stattfinden.

Die Spalte 1 enthält die Aktionsschritte [10]. In Spalte 2 sind die einzelnen Übergaben durchnummeriert. Eine Übergabe wird auch als **Transfer** bezeichnet. Wenn in jedem Arbeitstakt der RALU (in jedem Takt des Steueroperators) ein Transfer ausgeführt wird, gibt die Zahl in Spalte 2 die Taktnummer an. In Spalte 3 sind die Daten (Operanden bzw. Adressen) angegeben, die transferiert werden, und in Spalte 4 ihre Wege. Nach dem Laden des ADD-Befehls in das Befehlsregister steht im Feld A1 die Adresse von *c* und im Feld A2 die Adresse von *r*. Im AC steht das Resultat der vorangegangenen Subtraktion, also der Wert der Differenz *a*-*b*. Zur Illustration sollen die beiden Transfers des dritten Aktionsschrittes ausführlicher beschrieben werden. In diesem Schritt wird die ALU mit Operanden versorgt und das Resultat berechnet [10].

Transfer 4. Der Steueroperator stellt die Sammelweiche S3 so, dass die im Adressfeld A1 befindliche Adresse von *c* dem Halbkommutator HK übergeben wird. Dadurch öffnet sich das "Schloss" des Speicherplatzes mit der Adresse <c>.
Transfer 5. Der Steueroperator stellt HK und K auf Durchgang HS-DR und Z2 in Richtung ALU. Dadurch wird *c* vom HS nach DR transferiert und damit an den oberen Eingang der ALU gelegt. Unmittelbar danach (d.h. nach Ablauf des Übergangsprozesses in der ALU) erscheint am Ausgang der ALU das Resultat *r*, denn am unteren Eingang der ALU liegt die Differenz *a-b*. Weiterer Kommentare wird es kaum bedürfen, um die Abarbeitung des Programms von Bild 13.6 in allen Einzelheiten nachvollziehen zu können.

Eines der Ziele des Entwurfs von Prozessor und Maschinensprache, die gegen Ende des Kapitel 13.5.2 genannt worden waren, ist eine möglichst schnelle Befehlsausführung. Die Ausführungszeit einer Operation kann eventuell dadurch verkürzt werden, dass Datentransfers gleichzeitig ausgeführt werden. Voraussetzung dafür ist, dass die betreffenden Datenwege sich nicht berühren. Geht man unter diesem Gesichtspunkt die Transfers von Bild 13.8 noch einmal durch, erkennt man, dass Transfer 8 mit allen Transfers außer dem ersten gleichzeitig ausgeführt werden kann. Die Adresse des nächsten Befehl kann also berechnet werden, während der laufende Befehl ausgeführt wird. Weiterhin erkennt man, dass die Transfers 3, 4 und 5 in einem einzigen Takt ausgeführt werden können, wenn gewährleistet ist, dass die Steuersignale die Tore solange geöffnet halten, wie es für die fehlerfreie Übergabe der Daten (Spannungen) erforderlich ist. Der aufmerksame Leser wird erkennen, dass die drei Transfers gleichzeitig ausgeführt werden *müssen* oder dass zusätzliche Register erforderlich sind. Ganz anlog verhält es sich mit den Transfers 6 und 7. Die Anzahl der Takte lässt sich also auf 4 herabsetzen. Die Minimierung der Taktzahl pro Befehlsausführung ist eine wichtige Aufgabe der Entwicklungsingenieure.

Der nachfolgende END-Befehl stoppt den Steueroperator (genauer den Taktgenerator des Steueroperators). In bestimmten Fällen wird die Adresse des Befehls, der als nächster auszuführen ist, nicht durch Inkrementieren bestimmt, sondern durch das Programm selbst angegeben. Darauf wird im nächsten Abschnitt eingegangen.

Während der Abarbeitung eines Programms wiederholen sich ständig die fünf (sich evtl. überlappenden) Aktionsschritte der Befehlsausführung: Holen des aktuellen Befehls, Konditionieren der ALU, Versorgen der ALU mit den aktuellen Operanden, Abspeichern des Resultats und Berechnen der Adresse des nächsten Befehls. Diese Folge hatten wir *zentrale Steuerschleife* genannt.

Durch den detaillierten Nachvollzug der Operationsausführung erhält man eine genaue Vorstellung davon, was dieser kleine Kobold, Prozessor genannt, der dabei ist, die Welt zu verändern, tatsächlich macht. Es ist stets das gleiche stereotype Hin- und Herschieben von Bitketten zwischen denselben Registern. Unterwegs können die Ketten transformiert werden, und zwar stets von derselben Kombinationsschaltung, der ALU. Die Übergabe einer Bitkette zwischen zwei Registern wird häufig

Registertransfer genannt, unabhängig davon, ob der Übergabeweg über eine Kombinationsschaltung führt oder nicht.

Der Prozess, den der Leser vielleicht soeben nachvollzogen hat, läuft in der Welt viele Milliarden Male pro Sekunde ab und mischt sich in fast Alles ein, was wir Menschen tun und lassen. Er bringt die Vielfalt der Leistungsmöglichkeiten der technischen Informationsverarbeitung und er bringt "künstliche Intelligenz" hervor. Dieser Übergang von Quantität in Qualität erinnert an den Bau der Materie, an das ständige Herumkreisen der Elektronen auf den gleichen Bahnen um die Atomkerne, die ihrerseits alle aus den gleichen Bausteinen, aus Protonen und Neutronen bestehen. Und dieses stereotype Kreisen bringt die Vielfalt der Welt hervor.

Die Struktur der Materie und die Funktionsweise des Prozessors sind nur zwei Beispiele für das Phänomen, dass durch Wechselwirkung vieler (oft gleicher oder ähnlicher) Objekte (Prozesse) ein neues, kompliziertes Objekt (ein neuer, komplizierter Prozess) mit neuen Eigenschaften entsteht, eventuell eine Hierarchie immer komplizierterer Objekte. Zur Kennzeichnung der komplizierten Struktur derartiger Objekte hat sich das Wort *Komplexität* und für das Hervortreten neuer Eigenschaften das Wort *Emergenz* eingebürgert. Der Begriff der Komplexität wird uns in Kap.21 beschäftigen.

12 Der Leser beachte folgenden wichtigen und vielleicht überraschenden Umstand. Die Signale, die der Steueroperator des Prozessors in Bild 13.7 generiert, sind völlig andere Signale als diejenigen, die ein Steueroperator generiert, der einen realen, nach den Regeln der USB-Methode komponierten Kompositoperator (ein ON) steuert. Angenommen, das Operatorennetz enthält einen Subtrahierer und einen Addierer und es soll der Wert $(a\text{-}b)+c$ berechnet werden. Dann hat der Steueroperator Steuersignale zu generieren, durch welche die Werte von a und b dem Subtrahierer und dessen Ausgabewert und der Wert von c dem Addierer zugeführt werden. Diese Steuersignale haben wenig mit denjenigen zu tun, die der Steueroperator des Prozessors zu generieren hat, damit der Wert von $(a\text{-}b)+c$ berechnet wird (siehe dazu auch Kap.13.7 und die Diskussion zu Bild 19.4).

Das scheint zu bedeuten, dass sich mit Hilfe des Von-Neumann-Rechners das Komponieren von Kompositoperatoren nach der USB-Methode *nicht* von der Hardware auf die Software übertragen lässt. Unsere Idee, sprachliche Kompositoperatoren nach der USB-Methode zu komponieren, d.h. als Operationsvorschrift zu artikulieren, um sie dann vom Computer ausführen zu lassen, scheint zum Scheitern verurteilt zu sein, es sei denn, es gelingt, Operandenflussprogramme in die Maschinensprache zu übersetzen. Um das zu erreichen, müssen wir unsere Maschinensprache offensichtlich erweitern. Wir wollen uns überlegen, welche Erweiterung unbedingt erforderlich ist.

13.5.4 Sprungbefehl

Damit ein Operandenflussplan in eine Maschinensprache übersetzt werden kann, muss diese die Möglichkeit bieten, die Flussknoten des Operatorennetzes *imperativ*,

d.h. durch eine Folge von Befehlen auszudrücken. Insbesondere muss es eine Entsprechung zur Zweigeweiche geben, also einen Befehl, der die Verzweigung der Aktionsfolge steuert, m.a.W. der entscheidet, welcher von zwei möglichen Befehlen als nächster ausgeführt wird. Einem solchen Befehl sind wir bereits im *bedingten Sprung* der unbeschränkten Registermaschine begegnet (siehe Kap.8.4.3).

Die graphische Darstellung eines imperativen Programms, das einen Sprungbefehl enthält, muss offenbar ein *verzweigter* Aktionsfolgeplan sein, denn ein *linearer* Aktionsfolgeplan kann ohne zusätzliche Vereinbarungen keinen bedingten Sprung der Aktionsfolge darstellen. Bild 13.9 zeigt einen verzweigten Aktionsfolgeplan, der die Folge der Aktionen angibt, die bei der sequenziellen Berechnung der durch (8.1) definierten Funktion auszuführen sind (pot-Operator und sin-Operator sind nicht dekomponiert). Dabei ist eine in der Praxis gängige Darstellungform gemäß DIN (Deutsches Institut für Normung) gewählt, der sog. **Programmablaufplan** (**PAP**)[11]. Die Parallelogramme stellen die Dateneingabe bzw. Datenausgabe dar. Jedem rechteckigen Kästchen entspricht eine Aktion. Das rhombische Kästchen stellt eine Zweigeweiche im Aktionsfolgeplan dar. Es entspricht einem Prädikatoperator, der entscheidet, welcher Weg gewählt wird. Wenn das Prädikat [$x \leq 0$] erfüllt ist, wird der linke, mit t (von true) bezeichnete, andernfalls der rechte, mit f (von false) bezeichnete Weg eingeschlagen. Das rhombische Kästchen entspricht in Bild 8.1 der Zweigeweiche vor dem Sinusoperator und schließt den Steueroperator der Weiche ein.

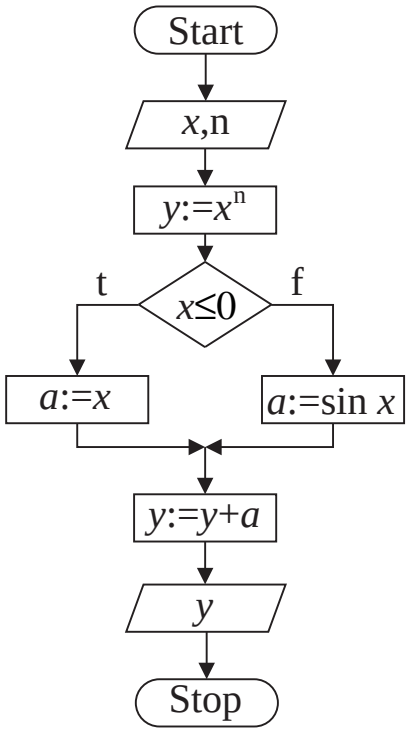


Bild 13.9 Aktionsfolgeplan (Programmablaufplan) für die Berechnung der durch (8.1) definierten Funktion. Der Rhombus stellt eine steuerbare Verzweigung der Aktionsfolge dar.

Zur Ergibtanweisung `y:=y+a` ist eine Bemerkung am Platze. Das Auftreten des Bezeichners `y` auf beiden Seiten der Anweisung könnte einen funktional denkenden Leser irritieren. Doch ist die Ergibtanweisung erlaubt und sinnvoll. Ihre interne Semantik ist das Überschreiben einer Speicherzelle. Um das zu erkennen, muss man sich vergegenwärtigen, dass in dem binär notierten Maschinenprogramm anstelle von `y` die Adresse der Speicherzelle

11 Siehe z.B. [Duden 89].

für den Wert von y steht. Das y auf der rechten Seite der Ergibtanweisung bezeichnet den Inhalt dieser Speicherzelle *vor* der Operationsausführung, das y auf der linken Seite den Inhalt *nach* der Operationsausführung.

Die graphische (zweidimensionale) Operationsvorschrift von Bild 13.9 soll nun in eine eindimensionale ("lineare") Befehlsfolge überführt werden. Wir sprechen von *Linearisierung*. Sie führt zwangsläufig dazu, dass bei der Abarbeitung unter bestimmten Bedingungen innerhalb der Befehlsfolge gesprungen werden muss. Der einfachste Befehl, der dies ermöglicht, heißt **bedingter Sprungbefehl**[12]. Mit seiner Hilfe lässt sich Bild 13.9 in eine Folge von Befehlen, also in einen imperativen Algorithmus überführen (linearisieren). Bild 13.10 gibt einen solchen Algorithmus an. Der dritte Befehl ist folgendermaßen zu lesen: "Springe, falls x nicht größer als 0 ist, nach Befehl 5". In Bild 13.9 entspricht diesem Befehl der Rhombus. Der Befehl setzt die Nummerierung der Befehle voraus, um das Sprungziel "adressieren" zu können.

```
1    y := x^n
2    a := x
3    SPNG x 5
4    a := sin(x)
5    y := y+a
```

Bild 13.10 Linearisierung von Bild 13.9

Wir wollen nun unsere Zwei-Adress-Maschine befähigen, Sprünge in einem Maschinenprogramm auszuführen. Dazu vereinbaren wir, dass in das Adressfeld A2 des Befehlsregisters die **Sprungadresse** eingetragen wird; das ist die Adresse desjenigen Befehls, der *angesprungen* wird, zu dem "verzweigt" werden soll oder, wie man auch sagt, dem die "*Steuerung übergeben*" werden soll (die Steuerung der RALU)[13]. In das Adressfeld A1 wird die Adresse derjenigen Variablen eingetragen, die im Weichenprädikat auftritt. Der Operationscode SPNG zusammen mit dem Bezeichner x legt nach dieser Vereinbarung das Prädikat [x≤0] fest. Wenn es erfüllt ist, soll zum Befehl mit den Nummer 5 gesprungen werden, andernfalls wird Befehl 4 ausgeführt. Hardwaremäßig wird das Springen durch die bisher nicht benutzte Verbindung von Z3 nach S2 in Bild 13.7 ermöglicht. Über sie kann der Inhalt von A2 in den Befehlszähler BZ eingetragen werden. Durch die Eintragung wird die automatische Befehlszählung durch Inkrementieren unterbrochen; nach dem Sprung wird sie fortgesetzt.

Unsere Vereinbarungen hinsichtlich des Sprungbefehls sind willkürlich. Es sind andere Vereinbarungen möglich. Sie müssen in der Sprachdefinition festgelegt sein. Es kann jedes Relationszeichen verwendet werden. Auch die Festlegung, ob bei

12 Es gibt auch komfortablere "Verzweigebefehle".

13 Offenbar liegt hier die Wurzel der Bezeichnung *Steuerfluss* als Synonym von *Aktionsfolge*. Mit der Bezeichnung "Steuerfluss" ist der "Fluss" der Übergabe der Steuerung von Befehl zu Befehl gemeint, m.a.W. die Folge der Befehlsausführungen, der *Aktionen*.

erfülltem oder bei nicht erfülltem Prädikat gesprungen werden soll, ist dem Sprachentwickler überlassen.

Steuerprädikate müssen von der ALU entschieden werden. Dazu muss sie durch den Operationscode (in unserem Beispiel die binär verschlüsselte Buchstabenfolge SPNG) auf die festgelegte Vergleichsoperation konditioniert werden, und ihr müssen die zu vergleichenden Größen (z.B. x und 0) zugeführt werden. Das resultierende Ausgabebit der ALU steuert die Weichen Z3 und S2 (die Steuerleitung ist in Bild 13.7 nicht eingezeichnet).

In Kap.13.7 werden wir uns davon überzeugen, dass unsere so erweiterte Maschinensprache *universell* ist. Sie erinnert sehr an die "Programmiersprache" der "Unbeschränkten Registermaschine" (URM) aus Kap.8.4.3. Offensichtlich ist die URM-Sprache in unmittelbarer Anlehnung an gängige Maschinensprachen definiert worden. Der Sprungbefehl der URM ist etwas komfortabler als unser Sprungbefehl, denn er stellt für die Programmierung der Sprungbedingung zwei Adressen zur Verfügung.

13.5.5 Matrixsteuerwerk

Nach dieser wichtigen Schaltungs- und Spracherweiterung wenden wir uns dem Entwurf des Steueroperators der RALU zu. Der erste Entwurfsschritt (wir führen ihn nicht im Detail aus) besteht darin, die Liste der Aktionsschritte der Addition von Bild 13.8 auf sämtliche Befehle zu erweitern. Dann ist für jeden Befehl bekannt, welche Datenwege in welcher Reihenfolge bei den verschiedenen Befehlsausführungen benutzt werden. Damit ist auch die Reihenfolge bekannt, in der die Tore geöffnet werden müssen. Genau dieses Wissen ist erforderlich, um eine Schaltung zu entwerfen, welche die notwendigen Steuersignalfolgen generiert.

In Kap.12.3.4 sind Entwurfsmethoden und mikroelektronische Realisierungen von Steueroperatoren mittels ROM besprochen worden. Ein Steueroperator dieser Art wurde dort als *Matrixsteuerwerk* und der ROM als *Steuermatrix* bezeichnet. Es wurde zwischen *rückgekoppelter* Steuerung, die aufgrund von Meldungen aus dem Arbeitsoperator (z.B. einer Waschmaschine) erfolgt, und *rückkopplungsfreier* Steuerung unterschieden, die keine Meldungen berücksichtigt. Letztere ist möglich, wenn bei der Steuerung keine Alternativen zu entscheiden sind und wenn die Dauer der Bausteinoperationen von vornherein festliegt, wenn also keine Ende-Meldungen der Bausteinoperatoren erforderlich sind. Das trifft für den Prozessor zu, denn die Zeitdauer, die ein Registertransfer benötigt, ist bekannt, und damit sind die Zeitpunkte bekannt, zu denen die einzelnen Schritte einer Aktion (einer Befehlsausführung) beginnen können. Die "Zeitmessung" (richtiger die Taktzählung) beginnt mit dem Start der Befehlsausführung.

Ein Steueroperator, der keine Meldungen zu berücksichtigen braucht, kann nach dem Funktionsgenerator-Prinzip aufgebaut werden (siehe Kap.12.3.3). Dabei werden (in Analogie zur Waschmaschinensteuerung) in die Adressiermatrix (vgl. Bild 12.2) die Zeitpunkte (Taktnummern) eingetragen (eingeprägt), in jede Zeile der

Matrix eine Taktnummer. In die Zeilen der Speichermatrix wird das jeweilige Steuerwort (Steuersignaltupel) eingetragen (eingeprägt). Das Steuerwort enthält für jeden wählbaren Übergabeweg ein Bit, z.B. für die Steuerung von S3 in Bild 13.7 drei Bit.

Hat der Prozessor aber einen Sprungbefehl mit Sprungbedingung zu interpretieren, hängt die aktuelle Aktion von einer Meldung über die Erfüllung der Sprungbedingung ab (entsprechend der Meldungen des Temperaturfühlers der Waschmaschine). Die Adressmatrix (und die zugrundeliegende Entscheidungstabelle) muss also neben der Taktspalte eine oder mehrere Spalten für Meldungen enthalten. Eine Zeile, die im Adressteil eine Meldung enthält, muss im Speicherteil die "Sprungadresse", d.h. die Taktnummer enthalten, mit der fortzufahren ist. Diese wird über eine Rückkopplungsschleife mit Taktverzögerer auf den Eingang der Adressmatrix zurückgegeben (in Bild 12.2 gestrichelt angedeutet).

Diese Methode kann in *jeder* Spalte angewendet werden, sodass kein Befehlszähler (Taktzähler), sondern nur ein *Taktgenerator* erforderlich ist, d.h. ein Impulsgenerator, der das Weiterspringen von Zeile zu Zeile auslöst. Damit ist grob skizziert, wie ein Matrixsteuerwerk arbeitet.

Damit der Prozessor das Programm von Bild 13.6 abarbeiten kann, muss er für jeden Befehl über eine spezielle Steuermatrix verfügen. Die Steuermatrizen (die ROMs) lassen sich zu einer Matrix vereinigen, indem sie z.B. "untereinander" angeordnet werden. Um die Ausführung eines Befehls zu starten, muss die Anfangszeile der betreffenden Matrix angewählt werden, indem der *OC-Entschlüsseler* (der Decodierer des Operationscodes) den Operationscode in die Nummer der Matrixzeile umcodiert, an der die betreffende Operation beginnt. Das *Operationsrepertoire* bleibt auf die Operationen der ALU beschränkt. Eine Multiplikation muss bereits vom Nutzer programmiert werden. Wir wollen uns überlegen, wie dem abgeholfen werden kann.

13.5.6 Operatorenkomponierung mittels Firmware

Die Verwirklichung einer im Grunde einfachen Idee befreit den Nutzer von der Aufgabe, das Multiplizieren und andere häufig auszuführende Operationen selber programmieren zu müssen. Die Idee überträgt diese Aufgabe dem Computerbauer. Sie wird durch die vorangehenden Überlegungen nahegelegt, sodass mancher Leser sich vielleicht schon gefragt hat, warum man die ALU nicht durch einen (geeignet adaptierten) Taschenrechner ersetzt. Eben darin besteht die Idee. Das Operationsrepertoire wird auf die wichtigsten Operationen wie Multiplizieren, Dividieren, Wurzelziehen u.ä.m. erweitert. Für jede Operation wird eine Steuermatrix entworfen und implementiert. Eine bestimmte Operation wird durch ihren Operationscode gestartet (ensprechend dem Drücken der Funktionstaste des Taschenrechners).

Es ergibt sich eine zweistufige Steuerhierarchie. Der übergeordnete Steueroperator organisiert die Abarbeitung von Maschinenprogrammen und die Kommunikation mit dem HS, jedoch nicht die Abarbeitung der einzelnen Befehle. Diese überträgt er

dem untergeordneten Steueroperator (“Taschenrechner-sop”), indem er ihm den aktuellen Operationscode und die Operanden übergibt. Gegebenenfalls können Operandenwerte unmittelbar über den Akkumulator an die nächste Operationsausführung übergeben werden. Die Erweiterung des Operationsrepertoires der Maschinensprache (Prozessorsprache) verlangt - ähnlich wie die Erweiterung um den Sprungbefehl - eine Hardwareerweiterung um einige Register und Weichen. Zur Bereitstellung der Eingabeoperanden reicht evtl. ein einziges Datenregister (DR in Bild 13.7) nicht aus. Beim Entwurf des untergeordneten Steueroperators ist vom KR-Netz des jeweiligen Kompositoperators auszugehen, im Falle der Multiplikation z.B. von der in Bild 13.3 dargestellten Schaltung.

Die entscheidende Erweiterung des “Taschenrechnerkonzeptes” betrifft das Matrixsteuerwerk, also die Firmware. Um das zu unterstreichen, unterscheiden wir hinsichtlich der Komponierung realer Operatoren zwischen Komponierung durch “*echte Hardware*” und Komponierung durch *Firmware*. Bild 19.1 zeigt die Prinzipschaltung eines Prozessors, der über ein Matrixsteuerwerk zur Operatorenkomponierung verfügt.

13.5.7 Mikroprozessor

Die Integration des Matrixsteuerwerks und der vom ihm gesteuerten Schaltung, der RALU, auf einem Chip liefert ein mikroelektronisches Bauelement von grob gesagt 1 cm^2 Größe, den **Mikroprozessor**. Der HS eines PC besteht i.d.R. aus

Bezeichnung	Jahr	Registerlänge [Bit] Datenreg./Adr.-Reg.	Taktfrequenz [Mhz]
8080	1976	8/16	bis 4
8086/186	1981	16/20	bis 10
80286	1984	16/24	bis 25
80386	1986	32/32	bis 40
80486	1989	32/32	bis 66
Pentium	1993	64/32	bis 233
Pentium II	1997	64/32	bis 450
Pentium III	1999	64/32	600
verschiedene Anbieter	2000		1000

Bild 13.11 Mikroprozessoren der Firma Intel.

mehreren Chips und nimmt eine größere Fläche auf der *Steckkarte* (so heißen die Einschübe eines PC) ein als der Prozessor. Ein wichtiger technischer Parameter eines Mikroprozessors ist neben der Größe und dem Energieverbrauch seine Taktfrequenz.

Sie bestimmt entscheidend die Rechengeschwindigkeit. Der Wettbewerb zwischen den Herstellern bewirkt gemeinsam mit den Wünschen der Nutzer eine ständige Erhöhung der Taktfrequenz. In Bild 13.11[14] sind für mehrer Mikroprozessoren der Firma Intel die für den Nutzer wichtigsten technische Daten und das Erscheinungsjahr aufgeführt.

Die Länge der Adressregister (der Adress-Felder im Befehlsregister) bestimmt die Größe des adressierbaren Speichers. Bei einer Länge von 32 Bit ist die Anzahl der adressierbaren Speicherplätze gleich 2^{32}, sie liegt nach (9.4a) also zwischen 10^9 und 10^{10}. Die Taktfrequenz ist durch die Zeitdauer begrenzt, die ein Registertransfer beansprucht. Diese hängt von den Schaltzeiten der Transistoren ab, also davon, wie schnell sich die erforderlichen Schaltspannungen aufbauen. Damit hängt die Taktfrequenz letzten Endes von der Zeitdauer ab, welche die Ladungsträger brauchen, um die halbleitenden Schichten der Tansistoren zu durchqueren. Letztendlich geht es also darum, Halbleiter mit möglichst hoher Beweglichkeit der Ladungsträger zu verwenden und möglichst dünne *p*- und *n*-leitende Schichten zu produzieren. Demzufolge wird weltweit nach neuen Halbleitermaterialien gesucht und die Dotierungstechnologie wird ständig vervollkommnet.

Die geringen Ausmaße von Mikroprozessoren und Arbeitsspeichern, die hohe Taktfrequenz und die Möglichkeit ihrer Massenproduktion (vgl.Kap.12.2) sind entscheidende Voraussetzungen dafür, dass der Computer beginnt, in sämtliche Bereiche menschlicher Tätigkeit einzudringen. Der Hardwareentwurf kann offensichtlich als “gelungen” beurteilt werden.

Nicht so günstig sieht es mit dem Softwareentwurf aus, treffender: mit dem Entwurf und der Implementierung von Programmiersprachen. Der Anwender wünscht sich eine einfache und dennoch ausdrucksstarke Sprache. Aber welche Freiheiten haben die Informatiker beim Entwerfen von Sprachen? Mit dem detaillierten Entwurf des Prozessors ist bereits diejenige Sprache festgelegt, die der Prozessor direkt, d.h. ohne vorherige Anpassung an die Prozessorstruktur (ohne Übersetzung) verarbeiten kann. Diese Sprache hatten wir in Kap.13.5.2 *Maschinensprache*[15] genannt.

Die Maschinensprache unserer Zwei-Adress-Maschine ist im wesentlichen durch den Aufbau des Befehlsregisters in Bild 13.7 und dessen erlaubte Inhalte definiert. Ihre Verwendung zur Beschreibung komplizierter Funktionen kann recht aufwendig werden. Die Suche nach besseren Programmiersprachen ist mit dem Entwurf des Prozessors “vorprogrammiert”. Sie dauert bis heute an. In Kap.18 wird ein Überblick über die Entwicklung gegeben. Eine sehr lebendige Darstellung der Softwareent-

14 Die Tabelle ist mit einigen Änderungen und Ergänzungen [Märtin 94] entnommen.

15 Genau genommen hatten wir sie *Prozessorsprache* genannt. Da aber in diesem Kapitel nur vom Ein-Prozessor-Rechner die Rede ist, braucht zwischen Prozessor- und Maschinensprache nicht unterschieden zu werden.

wicklung, die Hand in Hand mit der Hardwareentwicklung der Mikroprozessoren vom Intel 8080 bis zum Pentium verlief, findet der Leser in dem Buch “Der Weg nach vorn” von BILL GATES [Gates 97], einem der Hauptakteure auf dem Gebiet der Massenproduktion und Vermarktung von PC-Software.

13.6 Von-Neumann-Rechner

Durch Ankopplung eines Arbeitsspeichers an einen Prozessor entsteht ein *Einprozessorrechner*. Wenn Programme und Daten gleichberechtigt in einem einzigen Speicher abgespeichert werden, nennt man den Rechner **Von-Neumann-Rechner**, weil JOHN VON NEUMANN diesem Strukturprinzip zum Durchbruch verholfen hat. Der Arbeitsspeicher des Von-Neumann-Rechners wird auch *Hauptspeicher* (abgekürzt HS) genannt. Wenn für Programme und Daten getrennte Speicher existieren, wird von **Havard-Computer** gesprochen. Beide Rechnertypen sind Prozessorrechner. Heutzutage wird die getrennte Speicherung praktisch nur noch hinsichtlich des Cache-Speichers (siehe Kap.19.2.1 und Bild 19.1) angewendet.

Die Verbindung vom Prozessor zum Hauptspeicher und wieder zurück zum Prozessor kann als Rückkopplungsschleife aufgefasst werden, in welcher Daten “kreisen”. Die sich ergebende globale Struktur ist diejenige des Automaten von Bild 8.5. Dabei entspricht dem inneren Zustand z des Automaten der gesamte Inhalt des HS. In konsequenter Weiterführung dieser Analogie wird dem Prozessor bei jeder Befehlsausführung (in jedem “Automatentakt”, d.h. jedes mal, wenn das Tor vom Speicher zum Prozessor geöffnet wird) der gesamte Speicherinhalt zur Verfügung gestellt, aus dem sich der Prozessor den aktuellen Befehl und die aktuellen Daten heraussucht. Sodann führt der Prozessor den Befehl aus und generiert den neuen Zustand z', indem er den Inhalt des durch die Resultatadresse gekennzeichneten Speicherplatzes mit dem Ergebnis der Befehlsausführung überschreibt.

Die Vorstellung, dass der Prozessor in jedem Takt mit dem gesamten Speicherinhalt hantiert, aber nur einen kleinen Bruchteil benutzt bzw. verändert, ist durchaus erlaubt. Sie ist verbunden mit der Vorstellung, dass der nichtveränderte Teil gewissermaßen durch den Prozessor “durchgezogen” wird. In Kap.8.4.5 hatten wir dafür den Begriff des *Durchschleppens* eingeführt.

Die Organisation des Befehls- und Datenflusses im Von-Neumann-Rechner führt zu einem ernsthaften Problem, denn es gibt nur eine einzige Verbindung zwischen Prozessor und Speicher, die Verbindung K-HK in Bild 13.7. Über diese eine Verbindung wickelt sich die gesamte Kommunikation zwischen Prozessor und HS ab. Über sie werden sämtliche Befehle und Daten transportiert und zwar in beiden Richtungen und selbstverständlich streng sequenziell.

Dieses Leitungsstück zusammen dem Kommutator und dem Halbkommutator stellt einen sehr einfachen Datenbus dar (vgl. Kap.12.3.2). Sein Durchlassvermögen kann die Arbeitsgeschwindigkeit des Von-Neumann-Rechners evtl. erheblich herab-

setzen. Das Leitungsstück wirkt wie ein Straßenstück, das nur einspurig befahrbar ist und vor dem sich bei stärkerem Verkehr die Fahrzeuge in beiden Richtungen stauen. Die Verbindung zwischen Hauptspeicher und Prozessor wird sehr anschaulich **von-neumannscher Flaschenhals** genannt. Seine Erweiterung oder noch besser Eliminierung beschäftigt die Computerbauer seit Jahrzehnten. Der Entwurf und die Realisierung des Von-Neumann-Rechners stellt das Ergebnis der ersten großen Etappe des Weges “vom Rechnen zum Erkennen” dar.

13.7 Netzparadigma und imperatives Paradigma

In diesem Kapitel soll der begriffliche Hintergrund der Unterscheidung zwischen *Datenfluss*programmierung und *Aktionsfolge*programmierung aufgezeigt werden. Außerdem wird nachgewiesen, dass der Von-Neumann-Rechner alle rekursiven Funktionen und nur diese berechnen kann, womit die Aussage (8) aus Kap.13.3 bestätigt wird, dass die von einem Prozessorcomputer berechenbaren Funktionen rekursive Funktionen sind.

Der Schritt von der Hardware in die Software ist mit einem für die Informatik charakteristischen Wechsel des Denk- und Modellierungsparadigmas verknüpft, dem Übergang aus der Welt der *Operatorennetze* in die Welt der *Algorithmen*, speziell der *imperativen* Algorithmen. Der Paradigmenwechsel ist letzten Endes durch die Zentralisierung der Speicherung verursacht und führt zu einem Verlust an Übersichtlichkeit und Durchschaubarkeit; während die *Sichtbarkeit* der Operandenflüsse in der Beschreibung von Prozessen durch Operatorennetze unmittelbar gegeben ist, geht sie in der Beschreibung durch Algorithmen weitgehend verloren.

Wir haben es mit zwei fundamentalen **Modellierungsparadigmen** zu tun, die wir das **Netzparadigma** und das **imperative Paradigma** nennen. Verallgemeinert könnte man auch von **raum-zeitlichem** und **rein zeitlichem Paradigma** sprechen. Damit knüpfen wir an Kap.8.2.3 [8.11] an. Dort hatten wir bei der Behandlung verschiedener Methoden des kausaldiskreten Modellierens zwischen *raum-zeitlicher* und *rein zeitlicher* Prozessbeschreibung unterschieden. Das heißt nicht, dass für ein Modell in Form eines imperativen Algorithmus oder eines Aktionsfolgeprogramms der Raum nicht existiert und dass es keinerlei räumliche Beziehungen gibt. Doch hat der physische Raum, in welchem Daten im Computer übergeben und gespeichert werden, nichts mit dem Raum zu tun, in dem das *modellierte* Original existiert. Demgegenüber kann der gedachte Raum, in welchem ein Operatorennetz existiert und funktioniert, sehr viel mit dem Raum des Originals zu tun haben. Das “Verschwinden” des “Originalraumes” (des räumlichen Diskursbereiches) aus dem als imperatives Programm formulierten sprachlichen Modells bedeutet aber nicht sein völliges Verschwinden aus dem Modell. Implizit ist der Originalraum auch in einem imperativen Modell enthalten und zwar in den Bezeichnern der Operatoren und Operanden. Durch sie kann der Nutzer (Programmierer) den Raum in ein imperatives Programm

hineininterpretieren. Denn die externe Semantik der Bezeichner kann durchaus räumliche Aspekte besitzen.

Die beiden Paradigmen des Denkens (Modellierens) können alternativ angewendet werden, wenn raumzeitliche Prozesse sprachlich modelliert werden sollen, sei es zum Zwecke ihrer theoretischen Untersuchung, sei es zum Zwecke ihrer Steuerung. Der Kern dieser Wahlfreiheit liegt darin begründet, dass man Prozesse, die mit irgendwelchen Flüssen (Stoff-, Energie-, Informationsflüssen) verbunden sind, aus zwei unterschiedlichen Sichten betrachten kann, entweder aus der Sicht eines mit dem Fluss sich *mitbewegenden* Punktes oder aus der Sicht eines im Raum *fixierten* Punktes ("Standpunktes").

In der Welt der Operatorennetze lässt sich die Menge der verkoppelten Operatoren 14
als Menge kooperierender Akteure auffassen, die sich gegenseitig Daten zur weiteren Verarbeitung zuspielen. Die Akteure denken wir uns ortsfest, die Daten fließend. Für einen Beobachter lassen sich die Daten auf dem Wege ihrer Bearbeitung ohne große Schwierigkeiten verfolgen, vorausgesetzt, das Netz hat eine übersichtliche Struktur. Operationsvorschriften (Programme) für Operatorennetze legen den Datenfluss durch das Netz explizit fest, es sind *Datenflussprogramme*. Der Programmierer eines Datenflussprogramms schwimmt gewissermaßen mit dem Fluss mit.

In der Welt der imperativen Algorithmen gibt es nur einen einzigen Akteur, beispielsweise man selber, wenn man etwa die Aktionsfolge eines Kochrezeptes oder die Schritte eines Rechenverfahrens, z.B. des schriftlichen Multiplizierens, der Reihe nach ausführt. Die Operandenwege treten in der Vorschrift nicht explizit auf. Der Akteur muss sich die Operanden (Zutaten), die er gerade benötigt, selber beschaffen. Wenn der Akteur ein Prozessor ist, muss ihm per Programm mitgeteilt werden, wo er die Operanden der aktuellen Aktion findet und wo er das Resultat abspeichern soll. Dafür waren in der Maschinensprache (Kap.13.5.2) die Transportoperationen TVS und TNS eingeführt worden.

Die Rolle des Prozessors eines Von-Neumann-Rechners ist die eines zentralen Akteurs. Sie erzwingt eine Art der Programmierung, die sich von der Beschreibung des Operandenflusses durch ein Operatorennetz wesentlich unterscheidet. Ein Programm für den Prozessor legt keinen Datenfluss, sondern eine Aktionsfolge, d.h. eine Folge von Operationen mit explizit angegebenen Operanden fest. Maschinenprogramme sind *Aktionsfolgeprogramme*. Der "Standpunkt" des Programmierers eines Aktionsfolgeprogramms ist fixiert. Er sieht (mit den Augen des Prozessors) die Daten der Reihe nach "durch sich hindurchfließen".

Betrachtet man daraufhin noch einmal Bild 13.7, erkennt man Folgendes. In das Befehlsregister (BR) werden der Reihe nach die Befehle eines Aktionsfolgeprogramms[16] eingetragen. Nach jedem Befehlseintrag generiert der Steueroperator der

16 Es sei an das Synonym Steuerfluss für Aktionsfolge und Steuerflussprogrammierung für Aktionsfolgeprogrammierung erinnert.

RALU eine Steuerimpulsfolge, die den Datenfluss durch die RALU steuert. Das Befehlsregister bildet die Schnittstelle zwischen imperativem Algorithmus (Maschinenprogramm) und Operatorennetz (RALU und Hauptspeicher). Es sei noch einmal folgender Sachverhalt unterstrichen. Die Maschinensprachen gängiger Computer sind imperative Sprachen. Programme, die mit ihrer Hilfe geschrieben werden können, sind imperative Programme. Der Grund hierfür ist die zentrale Speicherung; die Folge ist das Format der Hardware-Software-Schnittstelle, also der einheitliche Aufbau des Befehlsregisters hardwareseitig und der Befehle softwareseitig mit festen Feldern für den Operationscode und die Adressen.

Im Hinblick auf die beiden Arten des Programmierens werden die genannten *Modellier*paradigmen, das Netzparadigma und das imperative Paradigma, zu *Programmier*paradigmen. Doch besitzen sie, wie gesagt, viel allgemeinere Bedeutung. Das sei an zwei Beispielen aus ganz anderen Bereichen veranschaulicht. Damit soll gleichzeitig der wesentliche Kern und die allgemeine (abstrakte) Bedeutung der Unterscheidung zwischen Operatorennetz und imperativem Algorithmus bzw. zwischen Datenfluss und Aktionsfolge verdeutlicht werden.

15 Das erste Beispiel entnehmen wir dem Straßenverkehr. Wir vergleichen die Sicht eines Autofahrers auf den Verkehr mit derjenigen eines Verkehrspolizisten. Der Autofahrer bewegt sich im Fluss des Verkehrs von Ort zu Ort. Er beobachtet ständig den Fluss und ordnet sich so ein, dass er durch seine Bewegung im Straßen*netz* sein Ziel erreicht. Dabei wird er den zu erwartenden Verkehrsfluss bis hin zum Ziel berücksichtigen. Seine Sicht ist *netz-* bzw. *datenflussorientiert*, wenn man die Autos als Daten bezeichnet. Die netzorientierte Sicht wird noch deutlicher, wenn man an einen Benutzer eines öffentlichen Verkehrs*netzes* denkt.

Demgegenüber ist der Verkehrspolizist auf einer bestimmten Kreuzung fest postiert und hat zu entscheiden, welches Auto im gegenwärtigen Zeitpunkt die Kreuzung passieren soll (welche Aktion ausgeführt werden soll). Die weiteren Wege der Autos (die Ziele ihrer Fahrer) interessieren ihn nicht. Seine Sicht mit Blick auf die Kreuzung (nicht auf das Straßennetz) ist *aktionsfolgeorientiert*. Er selber ist der zentrale Akteur. Das Beispiel hinkt insofern, als der Polizist den Auto*fluss* im benachbarten Straßen*netz* im Auge haben muss und der Autofahrer hinsichtlich seines Wagens eine *Folge* von Steuer*aktionen* ausführt.

16 Das zweite Beispiel entnehmen wir der Physik. Es betrifft die mathematische Beschreibung von Strömungsfeldern durch hydrodynamische Gleichungen. Dabei kann man sich als theoretischer Physiker (als mathematischer Modellierer von Strömungen) entweder sozusagen in ein Flüssigkeitsteilchen als Vehikel hineinsetzen, mit ihm mitschwimmen und seine Bahn und Geschwindigkeit "*datenflussorientiert*" beobachten bzw. berechnen (LAGRANGE'sche Herangehensweise), oder man kann an einem festen Punkt Aufstellung nehmen und die Strömungsparameter an diesem Punkt "*aktionsfolgeorientiert*" beobachten bzw. berechnen (EULER'sche Herangehensweise).

Beide Herangehensweisen, beide Sichten müssen letztendlich zu den gleichen Resultaten führen, im Strömungsbeispiel zu den gleichen Modellaussagen, im Verkehrsbeispiel zu dem gleichen angestrebten Endzustand, in dem alle Autofahrer ihre Ziele erreicht haben. Auf unser Problem übertragen heißt dies, dass sich ein und dieselbe Funktion sowohl datenfluss- als auch aktionsfolgeorientiert beschreiben und berechnen lässt. Die Folge ist die Herausbildung von zwei *Programmierparadigmen* oder *Programmierwelten*, wie zuweilen gesagt wird.

Mit den Konsequenzen der Spaltung der Programmierungstechnik in zwei vom Ansatz her unterschiedliche Bereiche werden wir konfrontiert, wenn wir versuchen, die Universalität des Von-Neumann-Rechners nachzuweisen. Zu diesem Zwecke ist zu zeigen, dass sich jedes Datenflussprogramm in ein Aktionsfolgeprogramm überführen lässt, denn jede rekursive Funktion ist eine USB-Funktion und lässt sich durch einen Datenflussplan beschreiben. Der Von-Neumann-Rechner kann also nur universell sein, d.h. jede rekursive Funktion berechnen, wenn sich jeder Datenflussplan in einen Aktionsfolgeplan überführen lässt.

Um den Beweis zu erbringen, kann man sich auf Kap.8.4.3 berufen. Unter der Annahme, dass sich der Speicher beliebig erweitern lässt, stellt der Von-Neumann-Rechner eine unbegrenzte Registermaschine (URM) dar, wobei die Register der URM den Registern der RALU und den adressierbaren Speicherplätzen des HS entsprechen. Vergleicht man die Befehle der Registermaschine mit denen der in Kap.13.5. definierten Maschinensprache, stellt man fest, dass das Befehlsrepertoire der Maschinensprache - es wird **Befehlssatz** genannt - den Befehlssatz der Registermaschine in sich einschließt. Das ist nicht verwunderlich, denn die fiktive Registermaschine ist im Grunde zu dem Zweck erdacht worden, die Universalität des Von-Neumann-Rechners nachzuweisen.

Damit könnten wir uns zufrieden geben, vorausgesetzt, wir vertrauen auf die Schlussfolgerung der Algorithmentheoretiker, dass die Registermaschine genau alle rekursiven Funktionen berechnen kann. Der Beweis dafür ist in Kap.8.4.3 nicht geführt worden. Wir wollen ihn auf unsere Weise nachholen.

Dazu genügt es zu zeigen, dass sich mit den Mitteln der Maschinensprache die Komponierungsmittel der USB-Methode, also die Datenübergabe von Aktion zu Aktion (an die Stelle der Bausteinoperationen treten jetzt Aktionen), die Gabel, die Vereinung und die Zweigeweiche darstellen (programmieren) lassen. In Kap.8.1 [8.1] hatten wir uns davon überzeugt, dass auf die Sammelweiche ohne Verlust der Universalität verzichtet werden kann, vorausgesetzt, es finden keine parallelen Prozesse in dem Netz statt, sodass keine Sammelweiche als Sequenzierer fungieren muss. Wir betrachten die Komponierungsmittel der Reihe nach.

1. Eine **Datenübergabe** zwischen zwei Aktionen[17], sagen wir eine Übergabe von Aktion A an Aktion B, lässt sich dadurch programmieren, dass die Resultatadresse

17 Mit anderen Worten ein Pfeil in einem PAP; vgl. Bild 13.9.

des Befehls zu Aktion A in den Befehl zu Aktion B als Operandenadresse eingetragen wird. Wenn zwischen den beiden Befehlen andere Befehle liegen, evtl. ein längeres Programmstück, muss gesichert sein, dass der Prozessor den Speicherplatz, in den er das übergebene Datum eingetragen hat, bis zu dessen Verwendung nicht überschreibt. Für die Sicherung des Speicherplatzinhaltes hat der Programmierer Sorge zu tragen. Hier haben wir es mit einem sehr einfachen Fall einer - zuweilen recht unangenehmen - Begleiterscheinung der Aktionsfolgeprogrammierung zu tun, die in ihrer Verallgemeinerung als **Seiteneffekt** (Effekte “an der Seite” oder “vonseiten” anderer Programmteile) bezeichnet wird.

2. Die Programmierung von **Vereinungen** ist durch das Befehlsformat gesichert, da es die Angabe zweistelliger Operationen samt ihrer beiden Operanden ermöglicht. Bei der Abarbeitung werden die beiden Operanden zu einem Operandenpaar *vereinigt*, gerade so, wie die Operanden (Summanden) des Additionsoperators in Bild 8.1 in dem Flussknoten vor dem Additionsoperator vereinigt werden.

3. Gabeln lassen sich dadurch programmieren, dass zwei Befehle ein und dieselbe Operandenadresse enthalten. In Analogie zur einfachen Datenübergabe muss garantiert sein, dass die Inhalte der beteiligten Speicherplätze nicht vorzeitig durch andere Befehlsausführungen überschrieben werden.

4. Der **Zweigeweiche** scheint der bedingte Sprungbefehl zu entsprechen. Genau genommen ist das jedoch nicht der Fall, denn der Sprungbefehl verzweigt keinen *Daten*fluss, sondern einen *Steuer*fluss (eine Aktionsfolge), also die Übergabe der Steuerung an den nächsten Befehl. Dennoch lässt sich eine Zweigeweiche mit Hilfe eines bedingten Sprungbefehls beschreiben. Dabei obliegt es dem Programmierer zu sichern, dass der Sprungbefehl die richtige (verlangte) *Daten*flussverzweigung bewerkstelligt, d.h. dass in den Befehlen, zu denen der Sprungbefehl verzweigt, die richtigen Operandenadressen eingetragen sind. Dass es dabei leicht zu Programmierfehlern kommen kann, insbesondere bei stark verzweigten Programmen, ist verständlich.

Damit haben wir uns überzeugt, dass sich sämtliche Komponierungsmittel der USB-Methode und damit auch der Datenflussprogrammierung in Aktionsfolgeprogrammen darstellen lassen. Die Überlegungen zeigen gleichzeitig, dass auch umgekehrt die Komponierungsmittel der Aktionsfolgeprogrammierung (also die Wiederholung von Operanden- bzw. Resultatadressen und der Sprungbefehl) in Datenflussprogrammen dargestellt werden können. Damit kommen wir zu folgendem Ergebnis: *Aktionsfolgepläne lassen sich in Datenflusspläne überführen und umgekehrt.*[18] Aus der wechselseitigen Überführbarkeit ergibt sich der

18 Bedeutend kürzer hätte der Nachweis mit Hilfe der *Durchschleppmethode* erbracht werden können.

Satz: Die Klasse der USB-Funktionen, d.h. derjenigen Funktionen, die sich nach Datenflussprogrammen berechnen lassen, ist mit der Klasse der imperativ berechenbaren Funktionen, d.h. derjenigen Funktionen, die sich nach Aktionsfolgeprogrammen berechnen lassen, identisch. Aktionsfolgeprogramme (Aktionsfolgepläne) berechnen alo rekursive Funktionen. 17

Da sich *jedes* Maschinenprogramm (jedes Aktionsfolgeprogramm) des Von-Neumann-Rechners in ein Datenflussprogramm überführen lässt und demzufolge eine USB-Funktion berechnet, ist jede mittels Von-Neumann-Rechner berechnete Funktion eine rekursive Funktionen. Da sich andrerseits jedes Datenflussprogramm in ein Aktionsfolgeprogramm und jedes Aktionsfolgeprogramm in ein Maschinenprogramm des Von-Neumann-Rechners überführen lässt, kann für jede rekursive Funktion ein Maschinenprogramm des Von-Neumann-Rechners geschrieben werden. Der Von-Neumann-Rechner kann also *jede* rekursive Funktion berechnen kann. Damit ergibt sich die Schlussfolgerung, dass die Klasse der durch den Von-Neumann-Rechner berechenbaren Funktionen mit der Klasse der rekursiven Funktionen identisch ist.

Diese Aussage hatten wir in Kap.13.3 unmittelbar aus dem *Berechenbarkeits-Äquivalenzsatz* [5] hergeleitet (vgl. Aussage (8) am Ende von Kap.13.). Die dortige Herleitung basierte auf dem Umstand, dass der Von-Neumann-Rechner ein KR-Netz darstellt oder in ein solches überführbar ist. Die jetzige Herleitung basiert auf dem Umstand, dass alle Komponierungsmittel der USB-Methode mit Hilfe der Maschinensprache beschreibbar sind. Da dies auch für die Sprache der unbeschränkten Registermaschine URM gilt, wie man unschwer erkennt, ist damit die Richtigkeit der Behauptung aus Kap.8.4.3 nachgewiesen, dass die Klasse der URM-berechenbaren Funktionen mit der Klasse der rekursiven Funktionen identisch ist. Außerdem folgt, dass *jedes Aktionsfolgeprogramm* ***wohlstrukturierbar*** ist[19], denn die Überführung eines Datenflussprogramms in ein Aktionsfolgeprogramm zerstört die Wohlstruktur nicht. Dass sich Operandenflusspläne wohlstrukturieren lassen, wissen wir aus Kap.8.4.5. [8.28]

Die oben getroffene Aussage, dass für jede rekursive Funktion ein Maschinenprogramm für ihre Berechnung geschrieben werden kann, ist gleichbedeutend mit folgendem

Satz: Der Maschinenkalkül eines Prozessorcomputers ist universell.

Der Begriff des Maschinenkalküls war in Kap.8.6 eingeführt worden. Der **Maschinenkalkül** eines Computers ist seine Maschinensprache zusammen mit ihrer internen Semantik (den semantischen Regeln, nach denen der Computer die Befehle der

19 Der ausführliche Beweis ist in [Böhm 66] erbracht worden.

Maschinensprache ausführt). Wir hatten einen Kalkül *universell* genannt, wenn in ihm jede rekursive Funktion berechnet werden kann.

Abschließend sollen einige besonders wichtige der eingeführten Begriffe zusammengestellt und die in der Literatur gängigen Bezeichnungen hinzugefügt werden.

- Die graphische Darstellung eines Kompositoperators mittels der Symbole der USB-Methode ohne Angaben zur Steuerung von Flussknoten heißt *Operandenflussgraph*, im Falle eines sprachlichen Operators auch *Datenflussgraph*.
- Ein Operandenflussgraph/Datenflussgraph, der mit allen erforderlichen Angaben zur Steuerung von Flussknoten beschriftet ist, heißt *Operandenflussplan/Datenflussplan*.
- Ein maschinenlesbarer Operandenflussplan/Datenflussplan heißt *Operandenflussprogramm/Datenflussprogramm*. Operandenflussprogramme sind *linearisierte* Operandenflusspläne, es sei denn der Rechner versteht die USB-Sprache oder eine andere, äquivalente zweidimensionale Sprache.
- Die graphische Darstellung einer Aktionsfolge mittels der Symbole der USB-Methode ohne Angaben zur Steuerung von Flussknoten heißt *Aktionsfolgegraph*.
- Ein Aktionsfolgegraph, der mit allen erforderlichen Angaben zur Steuerung der steuerbaren Flussknoten beschriftet ist, heißt *Aktionsfolgeplan*, in der Informatikliteratur meistens als *Steuerflussplan* bezeichnet.
- Ein nach DIN-Norm[20] dargestellter Aktionsfolgeplan heißt *Programmablaufplan* (abgekürzt PAP; siehe z.B. Bild 13.9).
- Ein maschinenlesbarer (i.d.R. also linearisierter) Aktionsfolgeplan heißt *Aktionsfolgeprogramm* oder *imperatives Programm*.
- Ein *Maschinenprogramm* ist eine Folge von Maschinenbefehlen. Ein *Maschinenbefehl* ist die Beschreibung einer Computeraktion. Ein terminierendes Maschinenprogramm ist ein imperativer Algorithmus.

20 DIN - Deutsches Institut für Normung.