Vorlesung

Rechnerstrukturen

Erster Teil

Wintersemester 2012/13

Gernot A. Fink
Technische Universität Dortmund
Fakultät für Informatik
44221 Dortmund
Gernot.Fink@udo.edu

Dieses Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung oder Verbreitung über den Kreis der Teilnehmerinnen bzw. Teilnehmer der Vorlesung hinaus ohne Zustimmung des Autors ist unzulässig.

Version vom 22. August 2012

Vorwort zur Ausgabe für das WS 2011/12

Die Vorlesung Rechnerstrukturen an der Fakultät für Informatik der TU Dortmund wird auch im Wintersemester 2012/12 durch das bewährte Veranstalterteam angeboten. Den zweiten Teil der Vorlesung bestreitet Prof. Peter Marwedel. Ich selbst werde wieder den ersten Teil der Veranstaltung übernehmen.

Die Grundlagen für die Lehrmaterialien zum ersten Vorlesungsteil wurden von Jun.-Prof. Thomas Jansen gelegt, dem ich für deren Zurverfügungstellung an dieser Stelle herzlich danken möchte. Aus dem von Herrn Jansen in der Zeit vom Wintersemester 2003/04 bis zum Wintersemester 2008/2009 entwickelten Vorlesungsskript entstand der vorliegende Text durch leichte Kürzungen im thematischen Überlappungsbereich mit dem zweiten Vorlesungsteil, unbedeutende Korrekturen, und geringfügige Ergänzungen.

Dortmund, August 2012 *Gernot A. Fink*

Inhaltsverzeichnis

1 Einleitung **7**

2 Hinweise zum Studium **10**

3 Repräsentation von Daten **11**
3.1 Repräsentation von Texten 13
3.2 Repräsentation von ganzen Zahlen 16
3.3 Repräsentation von rationalen Zahlen 19
3.4 Repräsentation anderer Daten 23

4 Boolesche Funktionen und Schaltnetze **24**
4.1 Einfache Repräsentationen boolescher Funktionen 26
4.2 Repräsentation boolescher Funktionen mit OBDDs 30
4.3 Schaltnetze 42
4.4 Rechner-Arithmetik 49
4.5 Optimierung von Schaltnetzen 68
4.6 Hazards 80
4.7 Programmierbare Bausteine 82

5 Sequenzielle Schaltungen **90**
5.1 Modellbildung 94
5.2 Synchrone Schaltwerke 97
5.3 Speicher 117

1 Einleitung

Wenn man sich als potenzielle Studentin oder potenzieller Student mit dem Studienfach Informatik auseinander setzt und versucht, sich umfassend darüber zu informieren, erhält man (hoffentlich) oft den warnenden Hinweis, dass „Computerfreak“ zu sein absolut keine Garantie für ein erfolgreiches Informatikstudium ist: Informatik ist weit mehr als nur „das mit den Computern“. Obwohl diese Feststellung selbstverständlich richtig ist, muss andererseits doch zugegeben werden, dass Informatik sich wesentlich *auch* mit Computern beschäftigt.

„Computer“ – das ist ein weites Feld; darum tut man auf jeden Fall gut daran, zumindest den Versuch zu machen, zu einer angemessenen Gliederung zu kommen, um sich so dem Thema systematisch zu nähern. Eine sehr nahe liegende Einteilung unterscheidet dabei Hardware, also die real in physikalischen Gegenständen realisierten Bestandteile eines Computers, und Software, die Realisierung von Funktionen und Funktionalität mittels Programmen, die auf dieser Hardware aufsetzen. Man muss sich darüber im klaren sein, dass diese Einteilung in gewisser Weise willkürlich ist und sicher mit gewissem Recht kritisiert werden kann. Sie ist jedenfalls sehr vom Stand der Technik abhängig und damit starkem Wandel unterworfen. Beim Entwurf und bei der Realisierung eines Computers ist man in weiten Teilen frei in der Entscheidung, welche Funktionen als Hardware und welche als Software realisiert werden sollen. Trotzdem ist diese Einteilung grundsätzlich praktikabel und vernünftig; sie schlägt sich zum Beispiel an der Fakultät für Informatik der Technischen Universität Dortmund in der Bachelorprüfungsordnung für den Studiengang Informatik nieder: Die Vorlesung „Datenstrukturen, Algorithmen und Programmierung“ beschäftigt sich wesentlich mit „Software“, die Vorlesung „Rechnerstrukturen“ deckt den Bereich „Hardware“ ab. Beide Teile sind für angehende Informatikerinnen und Informatiker unverzichtbare Eckpfeiler des Wissens, unabhängig davon, in welchem Bereich man später seine Schwerpunkte setzen möchte.

Eine andere gängige Unterscheidung beschäftigt sich mit Teilgebieten der Informatik, löst sich also in gewisser Weise vom Computer als Dreh- und Angelpunkt, und spricht von praktischer, technischer, theoretischer und angewandter Informatik (und vielleicht auch noch von weiteren Teilgebieten). So gesehen gehört die Vorlesung „Rechnerstrukturen“ eher zur technischen Informatik; aber Aspekte aus anderen Teilgebieten, zum Beispiel der theoretischen Informatik, lassen sich natürlich nicht ohne weiteres ausschließen – und die Einteilung ist ohnehin manchmal eher hinderlich als nützlich. Wir bemühen uns darum hier lieber um eine Sichtweise, die sich am konkreten Objekt orientiert: also wieder zurück zum Computer.

Wenn man einen handelsüblichen Computer betrachtet und zu „verstehen“ versucht, so kann man vernünftige Beschreibungen auf verschiedenen Abstraktionsebenen durchführen. Eine vielleicht nahe liegende, jedenfalls recht fundamentale Beschreibung setzt auf den sehr elementaren Bausteinen auf und beschäftigt sich zentral mit Transistoren. Auf dieser Ebene erfolgt die Beschreibung wesentlich in der Sprache der Elektrotechnik – und wir weisen eine Beschreibung auf dieser Ebene auch in das Fach Elektrotechnik. Zweifellos noch fundamentaler kann man auch einen einzelnen Transistor als schon relativ kompliziertes Konstrukt ansehen, noch weiter „hinabsteigen“ und auf physikalische Einzelheiten innerhalb von Transistoren und Schaltungen eingehen. Dann ist man im Grunde auf der Beschreibungsebene der Physik – und auch das ist sicher für eine Grundvorlesung in der Informatik nicht der passende Zugang. Wir entscheiden uns für einen Zugang, der schon wesentlich von technischen Realisierungen abstrahiert. Als unterste Schicht unserer Beschreibungen wählen wir die so genannte digital-logische Ebene, die auf logischen Bausteinen aufsetzt, welche eine wohldefinierte, primitive Grundfunktionalität zur Verfügung stellen. Darauf basierend kann man zu komplexen Schaltungen kommen, die (aus unserer Sicht) aus diesen Grundbausteinen aufgebaut sind. So kommt man zur so genannten Mikroarchitekturebene, auf der man arithmetisch-logische Einheiten findet, über Zentralprozessoren und Datenbusse spricht. Diese Ebene ist Bestandteil des zweiten Teils der Vorlesung und darum nicht Bestandteil dieses Skriptteils.

Nachdem wir uns kurz klar gemacht haben, warum moderne Computer allesamt Digitalrechner[1] sind, werden wir uns in Kapitel 3 mit der Frage auseinander setzen, ob das denn eine wesentliche Einschränkung ist und wie es uns bei so eingeschränkten Möglichkeiten gelingen kann, uns interessierende Daten im Rechner abzuspeichern. Konkret beschäftigen wir uns mit verschiedenen Repräsentationsmöglichkeiten für Texte und Zahlen. Solche Codierungen können prinzipiell beliebig definiert werden, wir beschäftigen uns aber mit wichtigen etablierten Standards, die wir kennen lernen müssen. Danach kommen wir zu grundlegenden Konzepten, die man einerseits mathematisch-logisch fassen kann (boolesche Algebra und boolesche Funktionen), andererseits aber natürlich auch praktisch handhaben möchte. Darum geht es uns in Kapitel 4 zum einen um die Darstellung boolescher Funktionen (natürlich auch im Rechner), andererseits aber auch um ihre Realisierung in Schaltnetzen. Dabei beschäftigen wir uns mit verschiedenen Einzelaspekten wie zum

[1] Wir werden hier in diesem Skript schon um eine saubere Fachsprache bemüht sein. Eine gewisse sprachliche Flexibilität, vor allem im Umgang mit englischsprachigen Begriffen, kann aber ohne Zweifel nur nützlich sein. Es sollte darum niemanden verwirren, wenn verschiedene (zum Beispiel englische und deutsche) Begriffe synonym nebeneinander verwendet werden. Ein Beispiel ist die Benutzung von „Computer“ und „Rechner“.

Beispiel der Optimierung von Schaltnetzen und ungewünschtem Schaltnetzverhalten. Ein wichtiger Schritt über Schaltnetze hinaus ist der Übergang zu sequenziellen Schaltungen, bei denen die Schaltung Kreise enthalten kann. Damit kommen wir in Kapitel 5 zu Schaltwerken, für die wir einerseits ein wichtiges theoretisches Modell, so genannte Automaten, kennen lernen wollen. Andererseits wollen wir aber auch wichtige Bausteine, die mit Hilfe solcher Schaltwerke realisierbar sind, besprechen; ein zentrales Beispiel in diesem Bereich ist die Realisierung von Speicherbausteinen.
Es gibt eine große Anzahl von Büchern, in denen die Gegenstände dieser Vorlesung behandelt werden. Zu einem wissenschaftlichen Studium gehört selbstverständlich die Lektüre wissenschaftlicher Literatur. Darum ist allen Hörerinnen und Hörern dringend ans Herz gelegt, in solchen Büchern zu lesen. Dass keines dieser Bücher perfekt zur Vorlesung passt, davon zeugt die Existenz dieses Skripts zur Vorlesung. Darum wird an dieser Stelle auch keine spezielle Einzelempfehlung ausgesprochen. Die kleine Liste hier genannter Bücher stellt ja schon eine Auswahl und eine Empfehlung dar. Allen hier genannten Texten ist gemeinsam, dass sie in der Universitätsbibliothek der Technischen Universität Dortmund vorhanden sind. Auf jeden Fall lohnt sich auch ein Gang in die Bereichsbibliothek Informatik und ein Stöbern in den Büchern, die unter der Signatur 3620 eingestellt sind.

- W. Coy: *Aufbau und Arbeitsweise von Rechenanlagen.* 2. Auflage, 1992.
- J. P. Hayes: *Computer Architecture and Organization.* 4. Auflage, 2003.
- J. L. Hennessy und D. A. Patterson: *Computer Organization and Design. The Hardware Software Interface.* 3. Auflage, 2004.
- W. Oberschelp und G. Vossen: *Rechneraufbau und Rechnerstrukturen.* 10. Auflage, 2006.
- A. S. Tanenbaum und J. Goodman: *Computerarchitektur.* 4. Auflage, 2002.
- A. S. Tanenbaum: *Computerarchitektur.* 5. Auflage, 2006.

2 Hinweise zum Studium

Die Vorlesung „Rechnerstrukturen“ vermittelt grundlegende Kenntnisse und Fähigkeiten, die zum Informatik-Studium unerlässlich sind. An Ihrem Ende steht eine schriftliche Fachprüfung, deren erfolgreiches Absolvieren das Erlangen dieser Kenntnisse und Fähigkeiten formal dokumentiert. Dieses Skript ist eine Komponente, die zum Bestehen dieser Fachprüfung beitragen soll. Es ist jedoch ausdrücklich kein Lehrbuch und sollte nicht als solches missverstanden werden. Insbesondere ist es sicher nicht zum Selbststudium geeignet. Weitere wichtige Komponenten sind die Vorlesung selbst sowie die zugehörigen Übungen. Dass die Vorlesungsfolien zur Verfügung gestellt werden, ist ein Service, der bei der notwendigen Nachbereitung der Vorlesung helfen soll, außerdem kann dank dieser Folien weitgehend auf eigene Mitschriften verzichtet werden, so dass der Vorlesung aufmerksamer gefolgt werden kann. Auch die Kombination von Skript und Vorlesungsfolien kann den Besuch der Vorlesung aber nicht ersetzen. Die Übungen zur Vorlesung schließlich sollen helfen, den ganz wichtigen Schritt von der passiven Rezeption zur aktiven Anwendung des Gelernten zu meistern. Dieser Schritt ist von zentraler Bedeutung für die schriftliche Fachprüfung. Die Übungsaufgaben sind auf den Inhalt der Vorlesung abgestimmt und bereiten auf verschiedene Weise auf die abschließende Prüfung vor. Sie können ihren Zweck aber nur erfüllen, wenn Übungsteilnehmerinnen und Übungsteilnehmer aktiv an den Übungen mitwirken. Das beginnt mit der regelmäßigen Bearbeitung der Übungsaufgaben, der Abgabe der Übungsaufgaben, setzt sich bei der Präsentation der eigenen Lösungen in der Übungsgruppe fort und findet in der Diskussion alternativer Lösungsvorschläge seinen Höhepunkt. In einer gut funktionierenden Übungsgruppe ist eigentlich kein Übungsgruppenleiter erforderlich. Wenn die Übungsgruppe zum „Frontalunterricht“ verkommt, sollte jede Teilnehmerin und jeder Teilnehmer sich fragen, was man selbst ändern kann, um zu einer besseren Zusammenarbeit zu kommen.

Für Informatik ist Teamarbeit in vielerlei Hinsicht von zentraler Bedeutung. Schon in der O-Phase sind hoffentlich Verbindungen geknüpft worden, die zu fruchtbaren Arbeitsgemeinschaften führen, die im besten Fall das ganze Studium hindurch tragen. Es ist auf jeden Fall sinnvoll, den Inhalt von gemeinsam gehörten Vorlesungen zum Gegenstand von Gesprächen zu machen. Sprechen über das eigene Fach hilft, das Gelernte besser aufzunehmen und zu verarbeiten und bereitet ideal darauf vor, Informatikerin oder Informatiker zu werden.

3 Repräsentation von Daten

Moderne Computer sind binäre Digitalrechner; das liegt daran, dass es technisch einfach möglich ist, genau zwei verschiedene Zustände zu realisieren und ein Bauelement schnell zwischen ihnen wechseln zu lassen. Analogrechner hingegen sind notwendig nicht völlig exakt; Digitalrechner, die auf mehr als zwei verschiedenen Grundzuständen aufbauen, gibt es praktisch nicht, weil die direkte Realisierung von mehr als zwei Grundzuständen technisch schwieriger, unzuverlässiger, teurer und langsamer ist. Wir werden im Kapitel 4 noch ein wenig genauer darauf eingehen.

Die Einschränkung auf nur zwei Grundzustände bedeutet, dass wir zunächst einmal keine Zahlen, ja nicht einmal Ziffern direkt repräsentieren können. Wir wissen aber alle schon (zumindest implizit) aus der Grundschule, dass das keine wesentliche Einschränkung ist. Wir sind ja durchaus in der Lage, unendlich viele natürliche Zahlen mit nur zehn verschiedenen Ziffern zu repräsentieren: Wir bilden aus den Ziffern $\{0, 1, \ldots, 9\}$ Zeichenketten. Die Wahl der Ziffern $\{0, 1, \ldots, 9\}$ ist dabei in gewisser Weise willkürlich und „zufällig"; das gleiche Prinzip funktioniert auch mit fast jeder anderen Anzahl von Ziffern – mehr als eine Ziffer muss es aber schon sein. Zu jeder natürlichen Zahl $b \in \mathbb{N} \setminus \{1\}$ und jeder natürlichen Zahl $n \in \mathbb{N}_0$ gibt es eine eindeutige Darstellung von n als

$$n = \sum_{i \geq 0} n_i \cdot b^i \tag{1}$$

mit $n_0, n_1, \dots \in \mathbb{N}_0$. Wir nennen b Basis und n_i Koeffizienten oder auch Ziffern. Es sei n_l der größte von Null verschiedene Koeffizient in der Summe (1), also $l = \max\{i \mid n_i \neq 0\}$. Dann schreiben wir auch

$$n = (n_l n_{l-1} n_{l-2} \cdots n_1 n_0)_b$$

und sprechen von der *b-adischen Darstellung* von n. Ist $b > 10$, so kann es „Ziffern" $n_i > 9$ geben. Häufig verwendet man dann Buchstaben als zusätzliche „Ziffern" (also A an Stelle von 10, B an Stelle von 11 usw.), so dass man eine kompakte und einfache Darstellung erhält. Bei sehr großen Basen wird das allerdings nicht durchgeführt: man denke zum Beispiel an die Basis 60, die bei der Darstellung von Sekunden und Minuten eine besondere Rolle spielt. Von besonderem Interesse für uns sind die Basen 10, 2 und 16; wir wollen uns darum zur Veranschaulichung als Beispiel die Darstellung der Zahl 748 zu diesen drei Basen ansehen.

$$\begin{aligned} 748 &= (748)_{10} \\ 748 &= (1011101100)_2 \\ 748 &= (2\text{EC})_{16} \end{aligned}$$

Stimmt das denn auch? Man überzeugt sich leicht davon, dass

$$8 \cdot 10^0 + 4 \cdot 10^1 + 7 \cdot 10^2 = 8 + 40 + 700 = 748$$

und darum die erste Zeile richtig ist. Außerdem ist

$$0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 1 \cdot 2^5 + 1 \cdot 2^6 + 1 \cdot 2^7 + 0 \cdot 2^8 + 1 \cdot 2^9 = 4 + 8 + 32 + 64 + 128 + 512 = 748$$

und die zweite Zeile stimmt auch. Für die dritte Zeile könnte man darauf verweisen, dass

$$12 \cdot 16^0 + 14 \cdot 16^1 + 2 \cdot 16^2 = 12 + 224 + 512 = 748$$

ist und damit die dritte Zeile rechtfertigen. Es geht aber auch noch etwas anders. Weil $16 = 2^4$ gilt, erhält man eine Ziffer zur Basis 16 aus vier Ziffern zur Basis 2 (natürlich von rechts nach links gelesen). Man kann also einen Blick auf Tabelle 1 werfen und die dritte Zeile direkt aus der zweiten gewinnen. Dieser Zusammenhang zwischen den Basen 2 und 16 ist es übrigens, der die Basis 16 für uns so interessant macht: man kann Zahlen zur Basis 16 kompakter als zur Basis 2 schreiben, im Gegensatz zur Basis 10 gibt es dabei so direkte Zusammenhänge, dass die Konvertierung mit einiger Übung mühelos im Kopf gelingt.

$(0)_{16} = (0000)_2$	$(1)_{16} = (0001)_2$	$(2)_{16} = (0010)_2$	$(3)_{16} = (0011)_2$
$(4)_{16} = (0100)_2$	$(5)_{16} = (0101)_2$	$(6)_{16} = (0110)_2$	$(7)_{16} = (0111)_2$
$(8)_{16} = (1000)_2$	$(9)_{16} = (1001)_2$	$(\text{A})_{16} = (1010)_2$	$(\text{B})_{16} = (1011)_2$
$(\text{C})_{16} = (1100)_2$	$(\text{D})_{16} = (1101)_2$	$(\text{E})_{16} = (1110)_2$	$(\text{F})_{16} = (1111)_2$

Tabelle 1: Umrechnung von Hexadezimalziffern in Binärzahlen

Manchmal ist es übersichtlicher, Repräsentationen gleicher Länge zu haben. Darum erlauben wir uns gelegentlich die Freiheit, für unsere Zwecke zu kurze Zahlen (eigentlich Repräsentationen von Zahlen) mit führenden Nullen aufzufüllen, wie wir das in Tabelle 1 schon gemacht haben.

Repräsentationen zu den genannten drei für uns relevanten Basen bekommen jeweils eigene Namen. Wir sprechen von der *Dezimaldarstellung* (Basis 10),

Binärdarstellung (Basis 2) und *Hexadezimaldarstellung* (Basis 16). Eine Ziffer der Binärdarstellung nennt man häufig auch ein *Bit*, eine Verkürzung von *bi*nary digi*t*.
Der Wechsel von einer Basis b zur Basis 10 ist jeweils nicht schwierig. Wir haben das schon implizit bei der Rechtfertigung unseres Beispiels ausgenutzt. Für den Wechsel von der Basis 10 zu einer Basis b gibt es verschiedene Möglichkeiten. Am einfachsten ist vielleicht der folgende Algorithmus 1, der zu einer Zahl $n \in \mathbb{N}$ und einer Basis $b \in \mathbb{N}\setminus\{1\}$ die Darstellung $(n_l n_{l-1} \cdots n_0)_b$ berechnet.

Algorithmus 1 (Basenwechsel **10 → *b***).

1. $l := -1$
2. So lange $n > 0$ gilt:
3. $\quad l := l + 1$
4. $\quad n_l := n - b \cdot \lfloor n/b \rfloor$
5. $\quad n := \lfloor n/b \rfloor$

3.1 Repräsentation von Texten

Dass uns zwei Ziffern reichen, um Zahlen aufzuschreiben, ist ja schon ganz schön. Aber eigentlich wollten wir uns doch überlegen, wie man basierend auf Bits beliebige Daten im Rechner repräsentieren kann. Weil wir gewohnt sind, vieles in Worte zu fassen, wollen wir uns zunächst auf die Repräsentation von Texten und darum auf die Repräsentation von Buchstaben konzentrieren. Es ist ganz klar, dass man im Grunde eine völlig beliebige Codierung wählen kann; also eine eindeutige Festlegung, die jedem zu repräsentierenden Zeichen eine Binärzahl zuweist. Man kann alternativ auch eine Binärzahl fester Länge verwenden (und wieder mit führenden Nullen auffüllen, wo das nötig ist), weil das technisch vieles einfacher macht. Es gibt aber auch Codierungen, die nicht längenkonstant sind, der Morsecode und die Huffman-Codierung sind die wohl bekanntesten Beispiele; sie sind für uns hier aber nicht relevant. Wenn wir Texte im Rechner repräsentieren wollen, so genügt es jedenfalls, jedem Buchstaben eine solche Codierung fest zuzuordnen. Weil es 26 Buchstaben gibt, reichen 5 Bits schon aus; damit lassen sich 32 verschiedene Zeichen codieren. Vielleicht sollten wir einfach alphabetisch vorgehen, also dem Buchstaben A die Folge 00000 zuordnen, dem Buchstaben B die Folge 00001 und so weiter.
`DIESALPHABETISTABERVIELLEICHTDOCHETWASZUKLEIN`
Offensichtlich wäre es schön, zusätzlich Kleinbuchstaben, Wortzwischenräume und auch Satzzeichen codieren zu können. Vermutlich wären ja auch

noch Ziffern nützlich... Tatsächlich müssen wir uns eigentlich keine eigenen Gedanken zu diesem Thema machen. Es gibt etablierte Standards, an die sich zu halten sicher Sinn macht. Das vermutlich bekannteste Beispiel ist der *ASCII Code* (American Standard Code for Information Interchange), der 1963 von der American Standards Organization verabschiedet wurde. Der ASCII Code verwendet sieben Bits zur Codierung eines Zeichens, eine Übersicht findet man in Tabelle 2. Die Codes 0000000 bis 0011111 sowie 1111111 stehen dabei für Steuerzeichen, die wir im Moment gar nicht weiter kennen lernen wollen. Die zwei- bis dreibuchstabigen Tabelleneinträge sind dabei in der Regel Abkürzungen, so steht etwa LF (0001010) für Linefeed, CR (0001101) für Carriage Return und DEL (1111111) für Delete. Es ist nicht ganz eindeutig, ob der Code 0100000 für ein Steuerzeichen oder ein Textzeichen steht, jedenfalls repräsentiert er einen Wortzwischenraum.

	000	001	010	011	100	101	110	111
0000...	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
0001...	BS	HT	LF	VT	FF	CR	SO	SI
0010...	DLE	DC1	XON	DC3	XOF	NAK	SYN	ETB
0011...	CAN	EM	SUB	ESC	FS	GS	RS	US
0100...		!	"	#	$	%	&	'
0101...	(	)	*	+	,	-	.	/
0110...	0	1	2	3	4	5	6	7
0111...	8	9	:	;	<	=	>	?
1000...	@	A	B	C	D	E	F	G
1001...	H	I	J	K	L	M	N	O
1010...	P	Q	R	S	T	U	V	W
1011...	X	Y	Z	[	\	]	^	_
1100...	`	a	b	c	d	e	f	g
1101...	h	i	j	k	l	m	n	o
1110...	p	q	r	s	t	u	v	w
1111...	x	y	z	{	|	}	~	DEL

Tabelle 2: ASCII Code

Natuerlich ist mit dem ASCII Code nicht das letzte Wort in Sachen Codierung gesprochen. Ausserhalb der Vereinigten Staaten von Amerika koennte man mit der Festlegung durchaus unzufrieden sein. Ist offensichtlich, warum das so ist?

Häufig besteht eine Speichereinheit in einem Computer aus acht Bit, die zu einem sogenannten *Byte* zusammengefasst sind. Es bietet sich darum an, den ASCII Code um ein achtes (führendes) Bit zu erweitern und damit 128

weitere Buchstaben codieren zu können. Die zumindest bei uns bekannteste Erweiterung ist ISO-8859-1 (oder auch ISO Latin1, dabei steht ISO für International Organization for Standardization), die nationale Sonderzeichen für die meisten westeuropäischen Sprachen enthält (zum Beispiel Deutsch, Französisch und Spanisch), also unter anderem Umlaute und ein Pfundzeichen £.

Die Einschränkung „Sonderzeichen für *die meisten westeuropäischen* Sprachen" macht schon deutlich, dass auch ein Byte in Zeiten zunehmender Internationalisierung nicht all zu viel ist. Es wundert darum sicher niemanden, dass der zur Zeit aktuelle Standard noch mehr Bits zur Codierung eines Zeichens verwendet, ursprünglich nämlich 16.

Gelegentlich nennt man die Zusammenfassung von 16 Bits ein *Wort*; dabei bezeichnet Wort aber in der Regel die „natürliche" Datengröße für einen Rechner, was offensichtlich von der Rechnerarchitektur abhängt. So kann für Computer ein Wort auch aus 32 oder 64 Bits bestehen.

Der Name des aktuellen Codes ist *Unicode*, der Standard wird vom internationalen Unicode Konsortium verwaltet. Unicode hat den Vorteil, plattformunabhängig zu sein und zunehmend Verbreitung zu finden. Weil man sich für eine Codierung mit 16 Bits entschieden hat, gibt es insgesamt 65 536 verschiedene sogenannte *Codepoints*. Man kann das für viel halten, es gibt aber mehr als 200 000 verschiedene Schriftzeichen weltweit und in Sprachen wie zum Beispiel Japanisch kommen auch noch neue hinzu. Der Raum für Zeichen ist also wiederum knapp und die Einführung neuer Zeichen durchaus ein Politikum. Unicode besteht aber nicht nur aus der erwähnten 16 Bit Codierung von Zeichen und Symbolen. Unicode unterstützt in der aktuellen Version 5.1.0 (August 2008) drei verschiedene Codierungsformate (Unicode Transformation Format), namentlich UTF-8, UTF-16 und UTF-32, die auf 8, 16, bzw. 32 Bits basieren. Dabei stimmen grundsätzlich jeweils die „größeren" Codierungen mit den „kleineren" Codierungen überein, man kann sich also jeweils vorstellen, dass vorne zusätzliche Bits hinzukommen. Weil Kompatibilität natürlich wichtig ist, stimmen die ersten 128 Zeichen mit dem ASCII Code überein. Der UTF-16 Codepoint für A ist also 0000 0000 0100 0001 (der ASCII Code ist 100 0001), der Codepoint für das Euro-Symbol (€), das ja auch in ISO Latin1 noch fehlt, ist 0010 0000 1010 1100. Von Dortmund aus betrachtet mag es interessant sein, dass mit der Version 5.1.0 im Jahr 2008 ein Zeichen für das große ß eingeführt wurde. Unicode geht aber über einen reinen Zeichencode hinaus. Zusatzinformationen wie zum Beispiel die Schreibrichtung werden ebenfalls festgelegt, außerdem ist es ausdrücklich vorgesehen, vorhandene Zeichen zu neuen Zeichen zu kombinieren, ohne dass diesen Kombinationen dann eigene Codepoints zugeordnet werden müssen.

Das geht aber über unsere Ausgangsfrage, wie man Texte mit Hilfe von Bits codieren kann, deutlich hinaus.

3.2 Repräsentation von ganzen Zahlen

Wenn es um natürliche Zahlen geht, kennen wir schon eine gute Darstellung, die nur mit Bits auskommt: die Darstellung als Binärzahl. Man legt sich auf eine feste Anzahl von Bits fest (8, 16, 32, ...) und kann dann die Zahlen von 0 bis 255, 65 535 bzw. 4 294 967 295 repräsentieren. Aber natürliche Zahlen reichen uns selbstverständlich nicht aus. Und etwas interessanter wird es schon, wenn man auch negative ganze Zahlen repräsentieren will. Es gibt vier gebräuchliche Methoden, das zu tun. Dabei wird stets eine feste Anzahl von zu verwendenden Bits l vorausgesetzt.

Die erste Methode ist die *Vorzeichenbetragsmethode*. Man verwendet eines der Bits als Vorzeichen (in der Regel benutzt man dazu das am weitesten links gelegene Bit, das häufig MSB (most significant bit) genannt wird) und versteht den Wert 0 als positives und den Wert 1 als negatives Vorzeichen. Man kann sich das leicht merken, wenn man daran denkt, dass man die Zahl aus Vorzeichenbit s und Betrag b als $(-1)^s \cdot b$ erhält. Bei Verwendung von 16 Bits können wir also die Zahlen $-32\,767, -32\,766, \dots, 32\,767$ darstellen. Wer genau mitzählt, wird bemerken, dass wir eine Zahl weniger repräsentieren können, als es verschiedene Bitmuster der Länge 16 gibt. Das liegt daran, dass die Darstellung der 0 nicht eindeutig ist, sie kann als 1000 0000 0000 0000 und als 0000 0000 0000 0000 dargestellt werden. Diese Zweideutigkeit ist stärker noch als die „Verschwendung" einer Zahl ein Ärgernis dieser Darstellungsform. Als vorteilhaft kann man die Symmetrie der Darstellung empfinden: wenn $n \in \mathbb{N}$ die größte darstellbare Zahl ist, so ist $-n$ die kleinste darstellbare Zahl. Wir erkennen, dass ein Vorzeichenwechsel in dieser Darstellung besonders einfach ist. Es genügt, das erste Bit zu invertieren (also aus einem Nullbit ein Einsbit bzw. einem Einsbit ein Nullbit zu machen). Dafür ist es nicht ganz so einfach zu erkennen, welche von zwei Zahlen eigentlich die größere ist.

Denken wir einen Moment an nicht-negative Zahlen in Standardbinärcodierung zurück. Will man von zwei Zahlen die größere bestimmen, so genügt es, die beiden Zahlen ziffernweise von links nach rechts zu vergleichen. Die Zahl, die als erste an einer Stelle, an der die andere Zahl ein Nullbit hat, ein Einsbit hat, ist die größere. Das liegt daran, dass die andere Zahl diesen Größenvorsprung auf den restlichen Stellen nicht mehr einholen kann, es gilt ja $2^i > \sum_{j=0}^{i-1} 2^j$ für alle $i \geq 0$.

Bei der Vorzeichenbetragsmethode funktioniert dieses Vorgehen aber nicht; das Vorzeichen muss zusätzlich berücksichtigt werden: Eine positive Zahl ist immer größer als eine negative Zahl, zwei positive Zahlen können verglichen werden wie zwei nicht-negative Zahlen, bei zwei negativen Zahlen dreht sich die Reihenfolge aber gerade um. Weil Fallunterscheidungen in Hardware nicht gut realisiert werden können, ist die Vorzeichenbetragsmethode ungünstig, wenn Zahlen häufig miteinander verglichen werden sollen.
Die zweite erwähnenswerte Repräsentation ist die *Darstellung mit Bias*, die auch *Exzessdarstellung* genannt wird. Man wählt eine feste Verschiebung (Bias) $b \in \mathbb{N}$ für alle Zahlen. Soll die Zahl $z \in \mathbb{Z}$ repräsentiert werden, so stellt man $z + b$ als Binärzahl dar. Das setzt voraus, dass $z + b \geq 0$ gilt. Man kann also genau die Zahlen $-b$, $-b + 1$, $\ldots$, $2^l - 1 - b$ darstellen. Meistens wählt man den Bias als $b = 2^{l-1}$ oder auch $b = 2^{l-1} - 1$. Dann hat das am weitesten links gelegene Bit fast die Funktion eines Vorzeichenbits; es ist allerdings 0 für negative und 1 für positive Zahlen. Die Darstellung der 0 ist bei dieser Methode eindeutig, ihre Darstellung hat an vorderster Stelle eine 0, wenn $b \leq 2^{l-1} - 1$ gewählt wird und eine 1, wenn $b \geq 2^{l-1}$ ist. Ein Vorzeichenwechsel ist hier nicht so einfach. Will man statt z nun $-z$ repräsentieren, muss man an Stelle von $z + b$ nun $-z + b$ darstellen; man muss also $2z$ abziehen, was nicht so ganz einfach ist. Wenn Vorzeichenwechsel eine häufig benutzte Operation ist, sollte man folglich nicht die Exzessdarstellung benutzen. Zum Ausgleich dafür ist der Vergleich zweier Zahlen nun besonders einfach. Weil ja nur nicht-negative Zahlen repräsentiert sind und die natürliche Ordnung auf den Repräsentationen der Ordnung der Zahlen entspricht, können wir die Repräsentationen einfach so vergleichen, wie wir das für nicht-negative Zahlen besprochen haben.
Als dritte Methode erwähnen wir die *Einerkomplementdarstellung*. Man repräsentiert positive Zahlen einfach mit ihrer Binärdarstellung, für negative Zahlen wird die Binärdarstellung stellenweise invertiert, aus jedem Nullbit wird also eine Eins und umgekehrt (das nennt man Einerkomplement einer Zahl). Damit kann man also die Zahlen $-2^{l-1} + 1$, $-2^{l-1} + 2$, $\ldots$, $2^{l-1} - 1$ darstellen. Die 0 hat wieder zwei verschiedene Repräsentationen, nämlich $000 \cdots 000$ und $111 \cdots 111$, was nicht ganz so schön ist. Ein Vorzeichenwechsel kann hier durch ein Invertieren aller Bits erreicht werden; das ist fast so einfach wie bei der Vorzeichenbetragsmethode. Wir haben hier allerdings auch die gleichen Schwierigkeiten beim Vergleich zweier Zahlen.
Die vierte und letzte hier vorgestellte Methode schließlich ist die *Zweierkomplementdarstellung*. Das Einerkomplement einer Zahl erhält man, indem man die Binärdarstellung invertiert. Das Zweierkomplement ergibt sich, wenn man auf das Einerkomplement noch 1 addiert. Die Darstellungen der positiven Zahlen sind also in Einerkomplementdarstellung und Zweierkomple-

mentdarstellung gleich, die 0 hat die eindeutige Zweierkomplementdarstellung $000\cdots000$, die Repräsentation $111\cdots111$ stellt ja schließlich die -1 dar. Wir können also die Zahlen -2^{l-1}, $-2^{l-1}+1$, …, $2^{l-1}-1$ in dieser Darstellung mit l Bits repräsentieren. Wenn ganze Zahlen repräsentiert werden sollen, wird meistens die Zweierkomplementdarstellung verwendet. Ein Vorzeichenwechsel ist noch etwas schwieriger als bei der Einerkomplementdarstellung: Nach dem Invertieren aller Bits muss noch eine 1 addiert werden. Das ist aber auf jeden Fall immer noch deutlich einfacher als ein Vorzeichenwechsel in der Exzessdarstellung. Der Vergleich zweier Zahlen ist hier aber auch nicht leichter als bei der Einerkomplementdarstellung. Warum in der Praxis trotzdem die Zweierkomplementdarstellung sehr häufig benutzt wird, werden wir erst nachvollziehen können, wenn wir im Abschnitt 4.4 über das Rechnen mit Zahlen sprechen.

Zur Veranschaulichung stellen wir die Zahlen $-8, -7, \ldots, 8$ in allen vier Repräsentationen in Tabelle 3 dar, dabei verwenden wir jeweils $l = 4$. Natürlich sind für $l = 4$ nicht alle diese Zahlen in allen Darstellungsformen auch tatsächlich darstellbar: es sind 17 Zahlen, aber nur 16 verschiedene Bitmuster.

	VZ-Betrag	Bias $b = 8$	Bias $b = 7$	1er-Kompl.	2er-Kompl.
-8	–	0000	–	–	1000
-7	1111	0001	0000	1000	1001
-6	1110	0010	0001	1001	1010
-5	1101	0011	0010	1010	1011
-4	1100	0100	0011	1011	1100
-3	1011	0101	0100	1100	1101
-2	1010	0110	0101	1101	1110
-1	1001	0111	0110	1110	1111
0	0000, 1000	1000	0111	0000, 1111	0000
1	0001	1001	1000	0001	0001
2	0010	1010	1001	0010	0010
3	0011	1011	1010	0011	0011
4	0100	1100	1011	0100	0100
5	0101	1101	1100	0101	0101
6	0110	1110	1101	0110	0110
7	0111	1111	1110	0111	0111
8	–	–	1111	–	–

Tabelle 3: Repräsentation der Zahlen $-8, -7, \ldots, 8$

3.3 Repräsentation von rationalen Zahlen

Die Überschrift dieses Abschnittes ist bewusst bescheiden gewählt. Sie geht anscheinend an unseren Wünschen etwas vorbei: zumindest in der Schule haben wir meistens mit reellen Zahlen gerechnet. Es ist aber ganz klar, dass wir nur endlich viele verschiedene Zahlen in unserem Rechner speichern können. Wir beschränken uns darum (genau wie unser Taschenrechner das macht) auf die Darstellung einiger rationaler Zahlen.
Es gibt zwei grundsätzlich verschiedene Arten, Zahlen $z \in \mathbb{Q}$ darzustellen: Man kann sich entschließen festzulegen, wie viele Stellen man vor und hinter dem Komma darstellen möchte, also dem Komma eine feste Stelle in der Repräsentation zuweisen. Wir sprechen bei dieser Darstellung aus nahe liegenden Gründen von *Festkommazahlen*. Möchte man etwa zur Basis 2 nicht-negative Zahlen $z \in \mathbb{Q}_0^+$ mit l Stellen vor dem Komma und m Stellen nach dem Komma darstellen, so erhält man

$$z = \sum_{i=-m}^{l} z_i \cdot 2^i,$$

wobei sich bei festen l und m natürlich nicht jede beliebige Zahl so darstellen lässt. Wir kennen das Problem alle schon von der Dezimaldarstellung: so hat zum Beispiel die rationale Zahl 1/3 keine Dezimaldarstellung endlicher Länge. Geht man zur Basis 3 über, ändert sich das offensichtlich; 1/3 ist $(0{,}1)_3$. Man muss also damit leben, dass einige rationale Zahlen bei fest gewählter Darstellung nicht exakt dargestellt werden können. Für den Spezialfall $m = 0$ haben wir diese Darstellung schon ausführlich besprochen. Darum wollen wir hier nicht weiter darauf eingehen.
Interessanter sind Repräsentationen, bei denen die Position des Kommas nicht feststeht, wir sprechen dann von *Gleitkommazahlen* (auch manchmal etwas unschön als *Fließkommazahlen* bezeichnet; das ist erkennbar zu dicht am englischen *floating point number*). Das Prinzip ist uns wiederum allen bestens bekannt, jedenfalls konfrontiert uns unser Taschenrechner gelegentlich damit, wenn wir besonders große oder besonders kleine Zahlen als Ergebnis erhalten. Man kann eine Zahl $r \in \mathbb{R}^+$ „normalisiert“ als $m \cdot 10^e$ darstellen mit einem Exponenten $e \in \mathbb{Z}$ und einer Mantisse $m \in \mathbb{R}$ mit $1 \leq m < 10$. Dass 10 für uns hier keine gute Basis der Zahlendarstellung ist, sollte inzwischen klar geworden sein. Darum überrascht es sicher niemanden, dass wir uns für die Darstellung $r = m \cdot 2^e$ mit $e \in \mathbb{Z}$ und $m \in \mathbb{R}$ mit $1 \leq m < 2$ entscheiden. Es versteht sich von selbst, dass wir uns für konkrete Repräsentationen von Mantisse m und Exponent e entscheiden müssen. Außerdem ist natürlich auch wieder über negative Zahlen zu sprechen.

Selbstverständlich sind bei diesen Entscheidungen wieder sehr viele verschiedene Methoden vorstellbar. Wir haben uns schon im Abschnitt über Texte (Abschnitt 3.1) klar gemacht, dass es sinnvoll ist, sich an Standards zu halten. Im Bereich der Gleitkommazahlen ist das IEEE 754-1985. Wir wollen hier nicht auf alle Feinheiten eingehen und uns mit dem Kennenlernen der Grundzüge zufrieden geben. Der Standard ist für verschiedene Anzahlen von Bits je Zahl definiert, die „normale" Länge sind 32 Bits, genauer sind Darstellungen mit 64 bzw. 80 Bits. Das Prinzip ist aber jeweils das gleiche.
Das erste Bit ist für das Abspeichern eines Vorzeichens reserviert, dabei steht wiederum eine 1 für ein negatives Vorzeichen. Es handelt sich also gewissermaßen um eine Vorzeichenbetragsdarstellung. Danach folgt die Darstellung des Exponenten in Darstellung mit Bias, dabei wird der Bias $b = 2^{l-1} - 1$ gewählt, wenn l Bits zur Darstellung des Exponenten verwendet werden. Werden für eine Zahl 32 Bits verwendet (einfache Genauigkeit), so wird der Exponent mit 8 Bits repräsentiert (also $b = 127$), bei 64 Bits Gesamtlänge (doppelte Genauigkeit) sind es 11 Bits für den Exponenten, bei 80 Bits Gesamtlänge schließlich sind es 15 Bits für den Exponenten. Die restlichen Bits dienen der Darstellung der Mantisse, es sind also 23 Bits bei Gesamtlänge 32, 52 Bits bei Gesamtlänge 64 und 64 Bits bei Gesamtlänge 80. Die erste Ziffer der Mantisse m ist ja stets 1 (es gilt ja $1 \leq m < 2$) und wird darum nicht explizit mit abgespeichert. Gespeichert werden dann die Nachkommastellen in Standardbinärcodierung, also (bei Gesamtlänge 32) $m_1 m_2 \cdots m_{23}$ und es gilt $m = 1 + \sum_{i=1}^{23} m_i \cdot 2^{-i}$.
Welche Zahl wird also zum Beispiel durch

$$1\,1000\,0111\,011\,1001\,0000\,0000\,0000\,0000$$

dargestellt? Weil das Vorzeichenbit 1 ist, wird eine negative Zahl repräsentiert. Die Repräsentation des Exponenten 1000 0111 können wir zunächst als Standardbinärcodierung interpretieren, dann wird also 135 dargestellt. Davon ziehen wir den Bias $2^{8-1} - 1 = 127$ ab, so dass wir 8 als Exponenten erkennen. Die Mantisse 011 1001 0000 ... ergänzen wir zunächst um die implizite Eins, wir haben also 1,0111001 und sehen, dass $1 + (1/4) + (1/8) + (1/16) + (1/128)$ repräsentiert wird. Insgesamt wird also

$$-2^8 \cdot \frac{128 + 32 + 16 + 8 + 1}{128} = -370$$

dargestellt.
Wir wollen uns jetzt noch umgekehrt überlegen, wie man zu einer Zahl ihre Repräsentation findet. Schauen wir uns die Zahl 5,25 an und suchen ihre

Repräsentation mit Codierungslänge 32. Weil die Zahl positiv ist, können wir schon 0 als Vorzeichenbit notieren. Wir benötigen nun zunächst eine Darstellung als Summe von Zweierpotenzen und sehen direkt, dass

$$5{,}25 = 4 + 1 + \frac{1}{4} = 2^2 + 2^0 + 2^{-2}$$

gilt. Natürlich kann es sein, dass es keine exakte Darstellung als Summe von Zweierpotenzen gibt. Weil wir das Thema Runden hier aussparen, kümmern wir uns um die Details in diesem Fall nicht und merken uns nur, dass man dann eben eine möglichst ähnliche Zahl repräsentiert. Kommen wir zurück zur 5,25. Wir haben eine Summe von Zweierpotenzen, brauchen aber eine Darstellung der Form $2^e \cdot (1 + \cdots)$ (mit einem passend zu wählenden Exponenten e) und machen uns darum klar, dass

$$5{,}25 = 4 + 1 + \frac{1}{4} = 2^2 + 2^0 + 2^{-2} = 2^2 \cdot \left(1 + 2^{-2} + 2^{-4}\right)$$

gilt. Nun sind wir schon fast am Ziel. Der Exponent 2 wird in Exzessdarstellung als 129 repräsentiert und es gilt $129 = (1000\,0001)_2$. Darum ist

$$0\,1000\,0001\,010\,1000\,0000\,0000\,0000\,0000$$

die Repräsentation von 5,25.
Man kann sich jetzt überlegen, welche Zahlen bei welcher Genauigkeit darstellbar sind, was also die größte darstellbare Zahl ist und welches die kleinste positive darstellbare Zahl ist. Wir führen das hier jetzt nicht vor und laden die Leserinnen und Leser dazu ein, sich diese Gedanken zur Übung ganz konkret selber zu machen.
Es gibt noch einige Besonderheiten, die auf jeden Fall erwähnenswert sind. Der Exponent könnte in der gewählten Darstellung ja eigentlich zwischen $-(2^{l-1} - 1)$ und 2^{l-1} liegen (also zum Beispiel zwischen -127 und 128 bei Gesamtlänge 32 und $l = 8$), jeweils einschließlich. Man schränkt diesen Bereich aber noch etwas ein und lässt die kleinste mögliche Zahl und die größte mögliche Zahl nicht zu. Die jetzt noch minimal und maximal darstellbaren Exponenten nennt man e_{min} und e_{max}. Für Gesamtlänge 32 haben wir also $e_{\text{min}} = -126$ und $e_{\text{max}} = 127$. Für „normale Zahlen“ gilt also für den Exponenten e auf jeden Fall $e_{\text{min}} \leq e \leq e_{\text{max}}$. Exponenten, die außerhalb dieses zulässigen Bereiches liegen, signalisieren „besondere Zahlen“. Ist der Exponent $e = e_{\text{max}} + 1$ und die dargestellte Mantisse Null, so wird (je nach Vorzeichenbit) entweder $+\infty$ oder $-\infty$ dargestellt. Diese „Zahlen“ ergeben sich zum Beispiel bei Rechnung $1/0$ bzw. $-1/0$. Ist der Exponent zwar $e = e_{\text{max}}+1$ aber ist die dargestellte Mantisse von Null verschieden, so wird „NaN“ (not a

number) dargestellt. Diese „Zahl" ergibt sich zum Beispiel bei $\sqrt{-1}$. Ist der Exponent $e = e_{\min} - 1$ und die dargestellte Mantisse Null, so repräsentiert die Zahl 0 – es war hoffentlich schon vorher klar, dass 0 nicht als „normale Zahl" repräsentierbar ist in dieser Darstellung. Der Wert des Vorzeichenbits erlaubt hier die Darstellung von $+0$ und -0. Ist schließlich der Exponent $e = e_{\min} - 1$ aber die dargestellte Mantisse von Null verschieden, so wird die Zahl $2^{e_{\min}} \cdot \sum_{i=1}^{l_m} m_i \cdot 2^{-i}$ dargestellt. Das erlaubt jetzt die Darstellung von noch kleineren Zahlen, als sie mit der normierten Darstellung eigentlich möglich sind. Das erfordert bei der Realisierung durchaus erheblichen zusätzlichen Aufwand. Was soll damit also eigentlich erreicht werden? Wir können uns das an einem Beispiel klar machen.
Nehmen wir an, dass wir zwei Variablen x und y haben und an dem Kehrwert der Differenz interessiert sind, also $1/(x - y)$ suchen. Es wäre offensichtlich keine gute Idee, direkt `z:=1/(x-y)` zu schreiben, weil das für $x = y$ sofort zu Problemen führt. Aber wie wäre denn die folgende Lösung?

```
If x ≠ y Then z := 1/(x-y)
```

Wir vergessen für einen Moment die Darstellung von „zu kleinen" Zahlen und betrachten folgendes konkretes Beispiel. Die Darstellung von x und y seien wie folgt gegeben.

	VZ	Exponent	Mantisse
x	0	0000 0001	000 0000 0000 0000 0000 0001
y	0	0000 0001	000 0000 0000 0000 0000 0000

Offensichtlich sind Mantisse von x und y verschieden, also fällt der Test „`x ≠ y`" positiv aus. Was sind eigentlich die Werte von x und y? Wir zählen nach und sehen, dass jede Zahl mit 32 Bits dargestellt ist. Damit ist der Bias $b = 127$, der Exponent ist gelesen als Binärzahl sowohl für x als auch für y die 1, also ist $e = -126$. Die Mantisse von y ist $m_y = 0$, wir haben also $y = (1 + m_y) \cdot 2^e = 2^{-126}$. Bei x ist die Mantisse $m_x = 2^{-23}$, wir haben also $x = (1 + m_x) \cdot 2^e = 2^{-126} + 2^{-149}$. Wenn $x - y$ berechnet wird, so ergibt sich 2^{-149}, was nicht darstellbar ist und darum zur 0 abgerundet wird. Darum ist die obige Zeile offenbar auch nicht sicher. Wir erlauben jetzt aber die erweiterte Darstellung. Dann ist 2^{-149} als

VZ	Exponent	Mantisse
0	0000 0000	000 0000 0000 0000 0000 0001

darstellbar: der Exponent ist vereinbarungsgemäß $e_{\min} - 1 = -127$, die Mantisse ist weiterhin 2^{-23}, wir stellen also die Zahl $2^{-23} \cdot 2^{-126} = 2^{-149}$ dar. Die Division kann also durchgeführt werden. Wir merken nur am Rande an, dass „`If x-y ≠ 0 Then z := 1/(x-y)`" auf jeden Fall besser wäre.

3.4 Repräsentation anderer Daten

Texte und Zahlen repräsentieren zu können ist sicher gut; vermutlich wollen wir aber auch noch andere Daten im Rechner abspeichern. Je nach Temperament und Vorliebe können einem dabei verschiedene Dinge zuerst einfallen: wir haben noch nicht über Bilder, Töne oder Musik sowie Filme gesprochen. Alle diese komplexen Themen passen auch nicht so recht zur Vorlesung „Rechnerstrukturen“, bei der es ja um die Grundlagen des Rechneraufbaus geht. Solche komplexen Daten werden oft nicht direkt von der Hardware eines Rechners unterstützt, im Gegensatz vor allem zu Zahlen.

Was einem aber eigentlich zunächst noch einfallen sollte, sind Programme. Um ein Programm ausführen zu können, muss es in aller Regel im Hauptspeicher des Rechners repräsentiert werden. Dazu werden Bitmuster benutzt, ein Bitmuster fester Länge (meistens 8 Bits) repräsentiert einen Maschinenbefehl. Zu einigen Maschinenbefehlen gehören zusätzliche Parameter, die als Argumente des Maschinenbefehls direkt dahinter im Speicher abgelegt werden. Dabei wird auch dort jeweils eine feste Anzahl von Bits (8 Bits, 16 Bits oder auch 32 Bits) verwendet. Es ergibt sich jetzt offensichtlich das Problem, das der Wert von beispielsweise acht festen Bits im Speicher verschiedene Bedeutung haben kann. Es kann sich ja um eine Zahl in irgendeiner Darstellung, den ASCII Code eines Zeichens oder auch einen Maschinenbefehl handeln. Manchmal verwendet man einige Bits als Typbits, um den Inhalt einer Speicherzelle sicher einem bestimmten Typ zuordnen zu können.

Gelegentlich bietet es sich an, mehrere (einfache) Daten logisch zu einem Datensatz zusammenzufassen. Vielleicht wollen wir Name, Geburtsdatum und Matrikelnummer zusammen abspeichern. In der Regel werden die Daten eines solchen Datensatzes direkt hintereinander in den Speicher gelegt, wobei jedes Datum einfachen Typs an einer eigenen Speicherzelle beginnt. Es bietet sich an, sich die einzelnen Speicherzellen, die jeweils acht Bits umfassen, als in Worte organisiert zu denken, wobei ein Wort je nach Rechner aus zwei oder vier Bytes besteht. Bei solchen Datensätzen möchte man dann gelegentlich, dass jedes Datum einfachen Typs in einem eigenen Wort steht. Belegt ein Datum weniger Speicher, bleiben die restlichen Bytes unbenutzt.

Bei Texten und bei Folgen von Daten gleichen Typs stellt sich manchmal die Frage, wie man erkennt, wann die Datenfolge (oder Zeichenfolge) zu Ende ist. Es gibt drei etablierte Methoden, diese Frage zu beantworten. Man kann entweder eine Anzahl fest vereinbaren; dann stellt sich das Problem eigentlich gar nicht. Man kann die Länge der Folge vor die eigentliche Folge schreiben, so dass explizit mit gespeichert wird, wie viele Elemente zu der Folge gehören. Oder man beendet die Folge mit einem speziellen Folgeendezeichen, das dann natürlich seinerseits nicht Bestandteil der Folge sein darf.

4 Boolesche Funktionen und Schaltnetze

Nachdem wir uns im Abschnitt 3 vergewissert haben, dass wir in der Lage sind, wesentliche Daten in binärer Form zu repräsentieren, wollen wir hier einen ersten Schritt in Richtung eines „Rechners“ wagen. Wir wollen zunächst untersuchen, wie man einfache mathematische und logische Funktionen in einem Digitalrechner realisieren kann. Dabei gilt unser Augenmerk wie in der Einleitung ausgeführt nicht der physikalischen oder elektrotechnischen Realisierung. Uns interessiert stattdessen der logische Aspekt der Angelegenheit, wir wollen verstehen, wie man basierend auf einfachen Grundfunktionen effizient komplizierte Maschinen zusammensetzen kann.
Als Basis dient uns im Grunde die klassische *Aussagenlogik*, bei der wir nur genau zwei Wahrheitswerte unterscheiden, die man in der Regel „wahr“ und „falsch“ nennt, für die wir aber aus nahe liegenden Gründen die Symbole „1“ und „0“ verwenden. Die Repräsentation der Wahrheitswerte ist damit schon einmal gesichert. Eine Aussage ist ein Satz, dem (prinzipiell) eindeutig ein Wahrheitswert zugeordnet werden kann, der also eindeutig „wahr“ oder „falsch“ ist. Aus einfachen Sätzen lassen sich mit Hilfe von Verknüpfungen neue Sätze zusammensetzen. Wir beschränken uns zunächst auf drei Verknüpfungen, die wir „und“, „oder“, und „nicht“ nennen. Die Verknüpfung „nicht“ (auch *Negation* genannt), für die wir das Zeichen „$\neg$“ verwenden, wird mit einer einzelnen Aussage x verknüpft. Es ist also $\neg x$ wieder eine Aussage. Dabei hat $\neg x$ gerade nicht den Wahrheitswert von x, ist also „wahr“, wenn x „falsch“ ist und umgekehrt. Die Verknüpfung „und“ (auch *Konjunktion* genannt), für die wir das Zeichen „$\wedge$“ verwenden, wird mit zwei Aussagen x und y verbunden. Es ist also $x \wedge y$ wieder eine Aussage. Der Wahrheitswert von $x \wedge y$ ist „wahr“, wenn x und y beide „wahr“ sind und „falsch“, wenn wenigstens eine von beiden „falsch“ ist. Die Verknüpfung „oder“ (auch *Disjunktion* genannt), für die wir das Zeichen „$\vee$“ verwenden, wird auch mit zwei Aussagen x und y verbunden, es ist also $x \vee y$ eine neue Aussage. Dabei ist der Wahrheitswert von $x \vee y$ genau dann „falsch“, wenn x und y beide „falsch“ sind und „wahr“ sonst.
Die so definierte Aussagenlogik ist eine *boolesche Algebra*, benannt nach George Boole (1815–1864), einem englischen Mathematiker. Wir wollen hier gar nicht so genau ausrollen, was das bedeutet[2], wollen aber gleich ein weiteres, ganz eng verwandtes Beispiel für eine boolesche Algebra definieren, das für uns von großer Wichtigkeit ist.

---

[2]Falls es doch jemanden brennend interessiert: Eine boolesche Algebra ist ein Verband mit zwei Verknüpfungen, die distributiv sind, in dem es bezüglich beider Verknüpfungen neutrale Elemente gibt und in dem es zu jedem Element ein inverses Element gibt.

Definition 2. *Wir nennen* $(B, \cup, \cap, ^-)$ *mit* $B := \{0, 1\}$ *und für alle* $x, y \in B$ $x \cup y := \max\{x, y\}$, $x \cap y := \min\{x, y\}$ *und* $\overline{x} := 1 - x$ *boolesche Algebra.*

Man überzeugt sich leicht, dass bei Interpretation von „wahr“ als 1 und „falsch“ als 0 sowie $\vee$ als $\cup$, $\wedge$ als $\cap$ und $\neg$ als $^-$ die boolesche Algebra gerade der oben skizzierten Aussagenlogik entspricht.
Es wird sich beim Umgang mit unserer booleschen Algebra als nützlich erweisen, eine Reihe von Rechenregeln zur Verfügung zu haben. Wir notieren den folgenden Satz entgegen guter wissenschaftlicher Praxis ohne Beweis und führen nur für die dritte Aussage exemplarisch aus, wie ein Beweis vollständig geführt werden kann. Es kann sicher nicht schaden, wenn Leserinnen und Leser sich jeweils für sich von der Korrektheit überzeugen und eigene Beweise führen.

Satz 3. *In der booleschen Algebra* $(B, \cup, \cap, ^-)$ *aus Definition 2 gelten die folgenden Rechengesetze für alle* $x, y, z \in B$*.*

Kommutativität: $x \cup y = y \cup x$, $x \cap y = y \cap x$

Assoziativität: $(x \cup y) \cup z = x \cup (y \cup z)$, $(x \cap y) \cap z = x \cap (y \cap z)$

Absorption: $(x \cup y) \cap x = x$, $(x \cap y) \cup x = x$

Distributivität: $x \cap (y \cup z) = (x \cap y) \cup (x \cap z)$, $x \cup (y \cap z) = (x \cup y) \cap (x \cup z)$

Komplementarität: $x \cup (y \cap \overline{y}) = x$, $x \cap (y \cup \overline{y}) = x$

Idempotenz: $x = x \cup x = x \cap x = \overline{\overline{x}}$

Resolution: $(x \cup y) \cap (\overline{x} \cup y) = y$, $(x \cap y) \cup (\overline{x} \cap y) = y$

de Morgansche Regel: $\overline{x \cup y} = \overline{x} \cap \overline{y}$, $\overline{x \cap y} = \overline{x} \cup \overline{y}$

Neutralelemente: $x \cup 0 = x$, $x \cap 1 = x$

Nullelemente: $x \cup 1 = 1$, $x \cap 0 = 0$

Weil B ja nur zwei Elemente enthält, lässt sich jede Aussage auf ganz elementare Art und Weise durch vollständige Fallunterscheidung beweisen. Am übersichtlichsten kann man das in Form einer Tabelle machen, wobei man die Tabelle auch *Wahrheitstafel* nennt. Wie angekündigt wollen wir uns das für das Beispiel der Absorption (erste Teilaussage) ansehen.

x	y	$x \cup y$	linke Seite $(x \cup y) \cap x$	rechte Seite x
0	0	0	0	0
0	1	1	0	0
1	0	1	1	1
1	1	1	1	1

Die Gleichheit der linken und rechten Seite der Gleichung für alle möglichen Belegungen der Variablen, also in allen Zeilen der Tabelle, beweist die Aussage.
Wir haben schon gesehen, dass die boolesche Algebra aus Definition 2 und die vorher skizzierte Aussagenlogik äquivalent sind. Es gelten natürlich die gleichen Rechenregeln. Wir werden uns im Folgenden vor allem der Zeichen $\vee$, $\wedge$ und $\neg$ bedienen und diese mit den Werten 0 und 1 verknüpfen. Bei komplizierteren Ausdrücken ist es manchmal übersichtlicher, das Negationssymbol $\neg$ durch eine lange Überstreichung zu ersetzen. Dabei handelt es sich nur um eine andere Schreibweise.

4.1 Einfache Repräsentationen boolescher Funktionen

Die Wahl der Verknüpfung $\vee$, $\wedge$ und $\neg$ war in gewisser Weise willkürlich. Im Grunde kann man sich beliebige Funktionen ausdenken, die einer Anzahl von booleschen Variablen einen Wert zuweisen. Wir wollen das formalisieren, dabei benutzen wir weiterhin die Schreibweise $B = \{0, 1\}$. Für $n \in \mathbb{N}$ bezeichnet B^n die Menge aller n-stelligen Tupel mit Einträgen aus B. Es ist zum Beispiel also $B^3 = \{(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1)\}$.

Definition 4. *Es seien $n, m \in \mathbb{N}$. Eine Funktion $f\colon B^n \to B^m$ heißt* boolesche Funktion*.*

Manchmal wird der Begriff der booleschen Funktion auch enger gefasst. Die in Definition 4 definierten Funktionen werden dann verallgemeinerte boolesche Funktionen genannt, der Begriff boolesche Funktion wird für Funktionen $f\colon B^n \to B$ reserviert. Auch wir werden uns meistens mit diesem Spezialfall beschäftigen. Offensichtlich kann man eine boolesche Funktion $f\colon B^n \to B^m$ durch eine Folge von booleschen Funktionen $f_1, \ldots, f_m$ mit $f_i\colon B^n \to B$ für alle $i \in \{1, \ldots, m\}$ ersetzen. Wir erkennen jetzt $\wedge$, $\vee$ und $\neg$ als spezielle boolesche Funktionen.
Wie viele verschiedene boolesche Funktionen $f\colon B^n \to B^m$ gibt es eigentlich? Die Antwort ist nicht schwer zu finden. Weil B^n nur endlich viele Elemente enthält, können wir eine boolesche Funktion $f\colon B^n \to B^m$ durch eine

x	y	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Funktion	Name	Symbol
f_1	Nullfunktion	0
f_2	Und (AND)	$\wedge$
f_4	Projektion	x
f_6	Projektion	y
f_7	Exklusives Oder (XOR)	$\oplus$
f_8	Oder (OR)	$\vee$
f_9	Nicht Oder (NOR)	
f_{10}	Äquivalenz	$\Leftrightarrow$
f_{13}	Negation	$\neg$
f_{14}	Implikation	$\Rightarrow$
f_{15}	Nicht Und (NAND)	
f_{16}	Einsfunktion	1

Tabelle 4: Alle booleschen Funktionen $f\colon B^2 \to B$

Wertetabelle mit $|B^n|$ Zeilen darstellen. Wir wissen, dass $|B^n| = 2^n$ gilt. In jeder Zeile gibt es nun $|B^m| = 2^m$ verschiedene mögliche Einträge. Wenn sich zwei solche Wertetabellen in mindestens einer Zeile unterscheiden, so sind die beiden zugehörigen Funktionen verschieden – vorausgesetzt die Zeilen sind in gleicher Reihenfolge aufgeführt. Es gibt also genau so viele verschiedene Funktionen wie es verschiedene Wertetabellen gibt. Und das sind, wie wir uns gerade implizit überlegt haben, genau $(2^m)^{2^n} = 2^{m \cdot 2^n}$. Es gibt also insbesondere $2^{2^2} = 2^4 = 16$ verschiedene Funktionen $f\colon B^2 \to B$. Tabelle 4 gibt sie alle zusammen mit ihrem Namen und ihrem speziellen Symbol (falls sie eines haben) wieder.

Wir haben schon eine Methode kennen gelernt, boolesche Funktionen darzustellen: man kann eine Wertetabelle angeben. Diese Darstellung ist aber unangenehm groß und wir wünschen uns kompaktere Repräsentationen. Nahe liegend ist es, sich auf eine feste Ordnung der Zeilen der Wertetabelle zu einigen und dann nur noch die Funktionswerte aufzulisten. Diese Darstellung nennen wir *Wertevektor*. Ein solcher Wertevektor enthält aber immer noch 2^n verschiedene Funktionswerte für eine boolesche Funktion $f\colon B^n \to B$.

Wir haben in Tabelle 4 die Zeilen so angeordnet, dass die Zeilen von 0 bis $2^n - 1$ nummeriert sind, wenn man die Belegung der Variablen als Binärzahl liest. Wir ordnen diese Zahl jeder Zeile zu und nennen diese Zahl *Index* der

Zeile. Bei Zeilen, in denen der Funktionswert 1 steht, nennen wir den Index *einschlägig*, andere Indizes nennen wir *nicht-einschlägig*. Betrachten wir nun eine Zeile mit einem einschlägigen Index i. Wir können eine (andere) Funktion angeben, die nur genau in dieser Zeile den Wert 1 annimmt und sonst den Funktionswert 0 hat. Diese Funktion nennen wir *Minterm*. Wir können einen solchen Minterm mit Hilfe von Negationen und Und-Verknüpfungen ausdrücken: wir verknüpfen jede Variable, die in der Zeile mit 1 belegt ist und die Negation jeder Variablen, die in der Zeile mit 0 belegt ist, mit Und. Selbstverständlich wird ein Minterm zu einem nicht-einschlägigen Index auf die gleiche Art gebildet.
Wir wollen das an einem Beispiel veranschaulichen und sehen uns die Funktion $f_{\mathrm{bsp}}\colon B^3 \to B$ an, die durch den Wertevektor $(1, 0, 0, 0, 1, 1, 1, 1)$ gegeben ist. Ihr Minterm m_5 ist einschlägig, da in der Zeile 6 ihrer Wertetabelle der Funktionswert 1 eingetragen ist[3], es ist nämlich $f_{\mathrm{bsp}}(1, 0, 1) = 1$. Der zugehörige Minterm ist durch $m_5(x_1, x_2, x_3) = x_1 \wedge \overline{x_2} \wedge x_3$ gegeben.
Wenn wir alle Minterme zu einschlägigen Indizes einer Funktion $f\colon B^n \to B$ mit $\vee$ verknüpfen, so erhalten wir wiederum eine Funktion. Diese Funktion nimmt genau dann den Wert 1 an, wenn mindestens einer der Minterme den Wert 1 annimmt. Folglich stimmt diese Funktion genau mit f überein. Dabei haben wir nur die Verknüpfungen $\vee$, $\wedge$ und $\neg$ benutzt. Wir wollen der Tatsache, dass man mit einem System von booleschen Funktionen jede andere boolesche Funktion darstellen kann, einen besonderen Namen geben und unsere Erkenntnis festhalten.

Definition 5. *Eine Menge $\mathcal{F}$ boolescher Funktionen heißt* funktional vollständig*, wenn sich jede boolesche Funktion durch Einsetzen und Komposition von Funktionen aus $\mathcal{F}$ darstellen lässt.*

Satz 6. $\{\vee, \wedge, \neg\}$ *ist funktional vollständig.*

Wir sehen, dass schon eine Menge von nur drei Funktionen funktional vollständig sein kann. Geht es vielleicht sogar mit noch weniger Funktionen?
Zunächst einmal können wir erleichtert feststellen, dass wir nun den Nachweis der funktionalen Vollständigkeit einfacher führen können. Weil wir wissen, dass $\{\vee, \wedge, \neg\}$ funktional vollständig sind, genügt es von einer anderen Funktionenmenge $\mathcal{F}$ nachzuweisen, dass sie die Funktionen $\{\vee, \wedge, \neg\}$ darstellen kann, um sie als funktional vollständig nachzuweisen. Daraus folgt direkt, dass sowohl $\{\vee, \neg\}$ als auch $\{\wedge, \neg\}$ funktional vollständig sind: durch Anwendung der de Morganschen Regeln ($x \vee y = \overline{\overline{x} \wedge \overline{y}}$) folgt das sofort. Sind diese Mengen nun minimal? Klar ist, dass man in diesen Mengen der

[3] Zu m_5 gehört die sechste Zeile, da zu m_0 die erste Zeile gehört.

Größe 2 auf keine der beiden Funktionen verzichten kann. Aber es könnte ja andere Funktionen geben, die alleine ausreichen? Und tatsächlich kann man eine solche Funktion finden, die eine minimale funktional vollständige Menge von Funktionen bildet.

Satz 7. $\{NAND\}$ *ist funktional vollständig.*

Beweis. Es genügt, $\neg$ und $\vee$ mit Hilfe von NAND darzustellen.
$\neg x = \text{NAND}(x, x)$ $\quad$ $x \vee y = \text{NAND}\left(\text{NAND}(x, x), \text{NAND}(y, y)\right)$ □

Wir wollen noch einmal auf die Darstellung von Funktionen mit Hilfe der Minterme der einschlägigen Indizes zurückkommen. Wir hatten argumentiert, dass wir für nicht-einschlägige Indizes keine Minterme hinzunehmen und wir für jeden einschlägigen Index mindestens einen Minterm haben, der den Funktionswert 1 annimmt. Das ist durchaus richtig, tatsächlich stimmt aber eine noch etwas schärfere Aussage: Für jeden einschlägigen Index gibt es *genau einen* Minterm, der den Wert 1 annimmt. Es spielt darum keine Rolle, ob wir das (normale inklusive) Oder ($\vee$) oder das exklusive Oder ($\oplus$) zur Verknüpfung benutzen. Wir können also jede boolesche Funktion auch als XOR-Verknüpfung der Minterme ihrer einschlägigen Indizes darstellen. Das bedeutet, dass auch $\{\oplus, \wedge, \neg\}$ funktional vollständig ist.
Bei der Definition von Mintermen haben wir besonderen Wert auf die Indizes gelegt, bei denen die Funktion f den Funktionswert 1 annimmt. Zu solchen Variablenbelegungen haben wir die Minterme betrachtet (also genau die Minterme zu einschlägigen Indizes), die Disjunktion aller dieser Minterme ist wieder f. Analog kann man mehr auf den Funktionswert 0 schauen.
Ist $m_i(x_1, \ldots, x_n)$ der i-te Minterm zu einer Funktion $f\colon B^n \to B$, so nennen wir $M_i(x_1, \ldots, x_n) := \neg m_i(x_1, \ldots, x_n)$ den i-ten *Maxterm*. Man kann eine Funktion auch mit Hilfe der Maxterme darstellen; dazu bildet man die Konjunktion aller Maxterme zu nicht-einschlägigen Indizes. Wer das jetzt nicht sofort einsieht, sollte sich die Zeit nehmen, sich das klar zu machen. Wir haben jetzt also drei verschiedene Darstellungen für boolesche Funktionen, denen wir auch jeweils einen Namen geben wollen.

Definition 8. *Sei $f\colon B^n \to B$ eine boolesche Funktion. Die Darstellung von f als Disjunktion aller ihrer Minterme zu einschlägigen Indizes nennen wir* disjunktive Normalform (DNF)*. Die Darstellung von f als XOR-Verknüpfung aller ihrer Minterme zu einschlägigen Indizes nennen wir* Ringsummen-Normalform (RNF)*. Die Darstellung von f als Konjunktion aller ihrer Maxterme zu nicht-einschlägigen Indizes nennen wir* konjunktive Normalform (KNF)*.*

Kommen wir noch einmal zu unserer Beispielfunktion f_{bsp} zurück. Wir sehen uns ihre Darstellung in den drei Normalformen an. Um Platz zu sparen, führen wir dafür folgende verkürzende Schreibweise ein, die wir auch weiter benutzen wollen. Wir lassen Klammern so weit wie möglich fallen und vereinbaren, dass im Zweifel zuerst $\wedge$ anzuwenden ist. Außerdem lassen wir den Operator $\wedge$ einfach fallen; bei direkt hintereinander geschriebenen Variablen ist also jeweils gedanklich eine Konjunktion einzufügen. Wichtig ist, dass $\overline{x_1}\,\overline{x_2}$ und $\overline{x_1 x_2}$ verschieden sind. Es steht nämlich $\overline{x_1}\,\overline{x_2}$ für $(\neg x_1) \wedge (\neg x_2)$, während $\overline{x_1 x_2}$ für $\neg(x_1 \wedge x_2)$ steht, was offensichtlich verschieden und auch nicht äquivalent ist.

$$\begin{array}{lcl}
\text{DNF} & : & \overline{x_1}\,\overline{x_2}\,\overline{x_3} \vee x_1\,\overline{x_2}\,\overline{x_3} \vee x_1\,\overline{x_2}\,x_3 \vee x_1\,x_2\,\overline{x_3} \vee x_1\,x_2\,x_3 \\
\text{RNF} & : & (\overline{x_1}\,\overline{x_2}\,\overline{x_3}) \oplus (x_1\,\overline{x_2}\,\overline{x_3}) \oplus (x_1\,\overline{x_2}\,x_3) \oplus (x_1\,x_2\,\overline{x_3}) \oplus (x_1\,x_2\,x_3) \\
\text{KNF} & : & (x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3})
\end{array}$$

Die konjunktive Normalform ist kürzer, weil die Beispielfunktion f_{bsp} öfter den Funktionswert 1 als den Funktionswert 0 annimmt. Es ist eine gute Übung nachzuweisen, dass tatsächlich sowohl f_{bsp} wie ursprünglich definiert als auch die DNF, RNF und KNF zur gleichen Funktion gehören. Das kann sowohl mit Hilfe einer Wertetabelle als auch unter Anwendung der Rechengesetze (Satz 3) geschehen. Es ist sicher eine gute Übung, beide Wege zu beschreiten.

4.2 Repräsentation boolescher Funktionen mit OBDDs

Allen bisher diskutierten Repräsentationen von booleschen Funktionen ist gemeinsam, dass sie zumindest manchmal sehr unhandlich sind, also sehr groß sind. Das muss auch so sein: Es gibt so viele verschiedene boolesche Funktionen, dass man nicht alle kompakt repräsentieren kann. Für den Umgang mit booleschen Funktionen im Computer hat sich allerdings eine andere Repräsentation etabliert, die sehr anschaulich ist und einige Vorzüge hat. Wir wollen sie darum kurz diskutieren.
Die Darstellung heißt *OBDD* (für Ordered Binary Decision Diagram) und kann am besten als graphische Darstellung beschrieben werden. Wir fangen damit an, dass wir eine Reihenfolge π (eine Ordnung oder auch Permutation) auf den Variablen festlegen, zum Beispiel $\pi = x_1, x_2, \ldots, x_n$. Ein OBDD zu dieser Ordnung π, das wir auch kurz πOBDD nennen, besteht aus Knoten, die entweder mit einer Variablen markiert sind oder mit 0 oder 1, und gerichteten Kanten (also Verbindungen von einem Knoten zu einem anderen Knoten), die entweder mit 0 oder mit 1 markiert sind. Dabei gilt es eine

Reihe von Spielregeln zu beachten: Jeder Knoten, der mit einer Variablen markiert ist, hat genau zwei Kanten, die ihn verlassen, von denen eine mit 0 und eine mit 1 markiert ist. Den Knoten, auf den die 0-Kante zeigt, nennen wir Null-Nachfolger; den Knoten, auf den die 1-Kante zeigt, nennen wir Eins-Nachfolger. Aus Knoten, die mit 0 oder 1 markiert sind, dürfen keine Kanten herausführen. Solche Knoten nennen wir Senken. Wir verlangen außerdem, dass es nur genau einen Knoten gibt, auf den keine Kante zeigt; diesen Knoten nennen wir Startknoten. Schließlich kommt noch die Variablenordnung ins Spiel: wenn die Nachfolgerknoten eines Knotens v mit Variablen beschriftet sind, so müssen diese Variablen in der Ordnung π hinter der Variablen des Knotens v stehen. Diese Regel garantiert unter anderem, dass es im OBDD keine Kreise geben kann und dass auf jedem Weg im OBDD jede Variable höchstens einmal auftauchen kann.
Wenn wir ein solches OBDD gegeben haben, so kann man zu jeder beliebigen Variablenbelegung einfach den Funktionswert berechnen. Man beginnt am Startknoten. Wenn man eine Senke erreicht hat, so ist der Funktionswert durch die Markierung der Senke gegeben. Andernfalls ist der Knoten mit einer Variable markiert und man betrachtet die Belegung dieser Variablen. Ist der Wert der Variablen 0, so folgt man der mit 0 markierten Kante zum Null-Nachfolger; andernfalls folgt man der mit 1 markierten Kante zum Eins-Nachfolger.

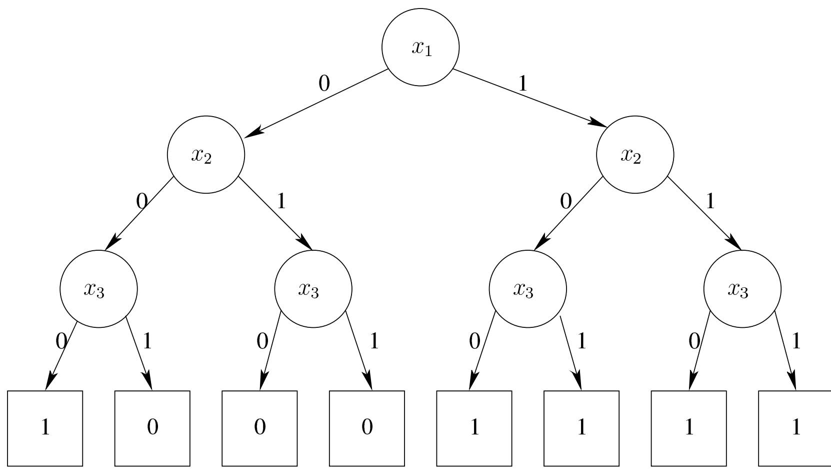


Abbildung 1: Ein πOBDD für f_{bsp} zu $\pi = x_1, x_2, x_3$

Man kann auch leicht angeben, welche Funktion ein OBDD darstellt. Tatsächlich stellt jeder Knoten im OBDD eine eigene Funktion dar: man kann sich ja jeden Knoten als Startknoten vorstellen und gedanklich alle Knoten „oberhalb“ dieses Knotens entfernen. Offensichtlich stellt eine 0-Senke

die Nullfunktion und eine 1-Senke die Einsfunktion dar (die Funktionen f_1 bzw. f_{16} aus Tabelle 4). Wenn wir einen Knoten v haben, der mit x_i markiert ist, so stellt er eine Funktion $f_v(x_1, \ldots, x_n)$ dar. Nehmen wir an, dass wir schon wissen, dass sein Null-Nachfolger die Funktion $f_0(x_1, \ldots, x_n)$ darstellt und sein Eins-Nachfolger die Funktion $f_1(x_1, \ldots, x_n)$. Dann stellt v die Funktion $\overline{x_i} f_0(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \vee x_i f_1(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$ dar. Tatsächlich kann man jede boolesche Funktion $g \colon B^n \to B$ für jedes $i \in \{1, 2, \ldots, n\}$ so zerlegen, es gilt stets

$$g(x_1, x_2, \ldots, x_n) = \overline{x_i} g(x_1, x_2, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \vee x_i g(x_1, x_2, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n).$$

Man nennt diese Art der Zerlegung einer booleschen Funktion *Shannon-Zerlegung*.

Wir wollen jetzt zur Veranschaulichung ein OBDD für unsere Beispielfunktion $f_{\text{bsp}} \colon B^3 \to B$ von oben erstellen. Dabei legen wir uns auf die Variablenordnung $\pi = x_1, x_2, x_3$ fest. Wir beginnen am Startknoten mit einem x_1-Knoten, geben ihm als Null- und Eins-Nachfolger je einen x_2-Knoten, diesen jeweils als Null- und Eins-Nachfolger je einen x_3 Knoten. Jetzt können wir wie oben beschrieben für jede Variablenbelegung den Kanten folgen und wissen dann, welche Senke wir jeweils an den Knoten anbringen müssen. Das Ergebnis sieht man in Abbildung 1.

Das ist ja nun auch ziemlich groß. Aber wir können offensichtlich noch Knoten sparen. Wir müssen ja nicht jedem Knoten seine eigenen zwei Senken spendieren; es genügt, jeweils eine 0-Senke und eine 1-Senke zu haben. Wir können also durch *Verschmelzung* von Knoten ein kleineres OBDD bekommen.

Sehen wir uns einen Moment das Ergebnis in Abbildung 2 an. Wir erkennen, dass wir noch mehr Knoten verschmelzen können: Die beiden rechten x_3-Knoten machen offensichtlich genau das gleiche: sie haben den gleichen Null- und den gleichen Eins-Nachfolger. Verschmelzen wir alle solche Knoten, so bekommen wir ein OBDD wie das in Abbildung 3.

In diesem OBDD gibt es aber immer noch offensichtlich überflüssige Knoten: Ein Knoten, bei dem Null- und Eins-Nachfolger gleich sind, ist zu nichts gut. Unabhängig vom Wert der Variablen kommen wir zum gleichen Nachfolger. Solche Knoten können wir einfach *eliminieren* und dadurch wiederum das OBDD verkleinern. Wir führen diese Elimination bei allen möglichen Stellen aus und erhalten das OBDD in Abbildung 4. Das ist wesentlich kleiner; und auch nicht mehr zu verkleinern. Wir halten die beiden Methoden zur Verkleinerung einmal ausdrücklich fest.

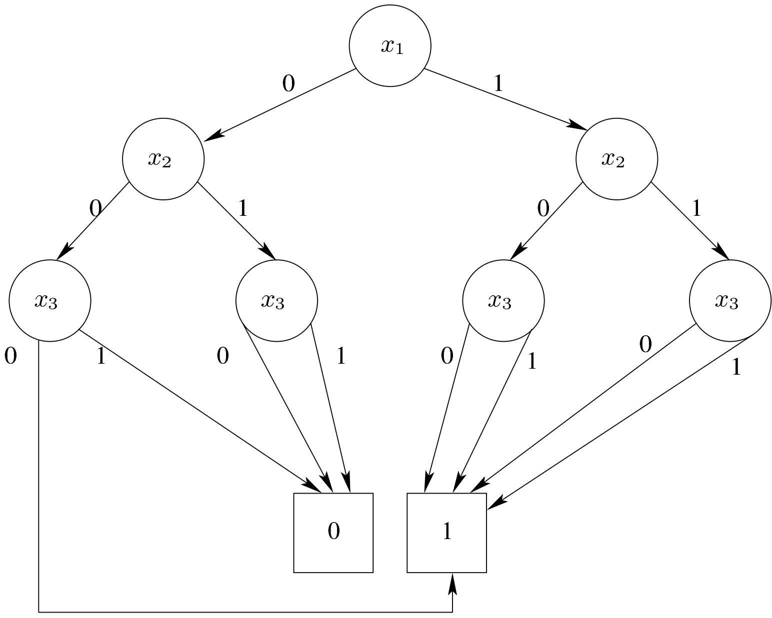


Abbildung 2: Ein πOBDD für f_{bsp} zu $\pi = x_1, x_2, x_3$

Verschmelzungsregel: Wenn in einem OBDD zwei Knoten gleich markiert sind und gleiche Null- und Eins-Nachfolger haben, so können die beiden Knoten verschmolzen werden.

Eliminationsregel: Wenn in einem OBDD ein Knoten gleichen Null- und Eins-Nachfolger hat, so kann der Knoten entfernt werden. Alle Kanten auf diesen Knoten werden auf den Nachfolgerknoten umgeleitet.

Natürlich ist man immer daran interessiert, möglichst kleine OBDDs zu haben. Darum sollte man die beiden Regeln immer anwenden, wenn das möglich ist. Interessant ist, dass man keine weiteren Regeln zu finden braucht. Wir notieren das Ergebnis als Satz, den wir natürlich auch beweisen wollen, dazu wird allerdings einiges an Vorbereitung notwendig sein.

Satz 9. *Wendet man in einem πOBDD die Verschmelzungsregel und die Eliminationsregel beliebig oft und in beliebiger Reihenfolge an, so ändert sich dadurch die durch das OBDD dargestellte Funktion nicht. Ist keine der beiden Regeln anwendbar, so ist das πOBDD ein kleinstes πOBDD für die dargestellte Funktion. Alle anderen kleinsten πOBDDs für die gleiche Funktion sind (bis auf Isomorphie) gleich.*

Wir können aus einer booleschen Funktion $f\colon B^n \to B$, die über den Variablen $x_1, x_2, \ldots, x_n$ definiert ist, eine Funktion $g\colon B^{n-1} \to B$ gewinnen, indem wir uns irgendeine Variable x_i aussuchen und sie mit einem festen Wert

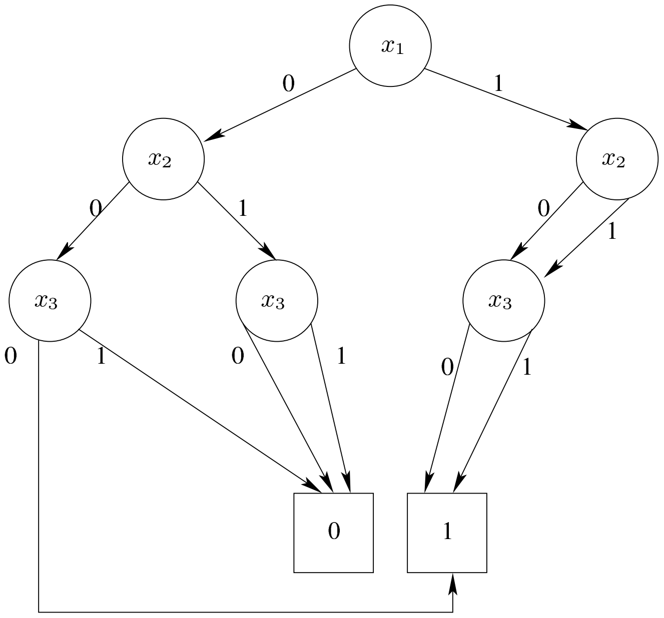


Abbildung 3: Ein πOBDD für f_{bsp} zu $\pi = x_1, x_2, x_3$

$c \in B$ belegen. Wir führen dazu die Schreibweise $f_{|x_i=c}$ ein, die bedeutet, dass x_i fest mit dem Wert c belegt wird. Wir nennen $f_{|x_i=c}$ eine Subfunktion von f. Natürlich können wir $f_{|x_i=c}$ auch als Funktion über allen n Variablen auffassen. Man macht sich leicht klar, dass für die Funktion $g := f_{|x_i=c}$ jedenfalls $g_{|x_i=0} = g_{|x_i=1}$ gilt. Wenn es für eine Funktion $f\colon B^n \to B$ eine Belegung $b \in B^n$ gibt, so dass $f_{|x_i=0}(b) \neq f_{|x_i=1}(b)$ gilt, so sagen wir, dass die Funktion f von der Variablen x_i essentiell abhängt. Es ist klar, dass man sich nicht darauf beschränken muss, eine Variable auf einen konstanten Wert festzulegen. Wir schreiben $f_{|x_{i_1}=c_1, x_{i_2}=c_2, \ldots, x_{i_m}=c_m}$ für die Subfunktion von f, die wir erhalten, wenn wir die Variable x_{i_j} fest mit dem Wert c_j belegen und das gleichzeitig für alle $j \in \{1, 2, \ldots, m\}$.

Betrachten wir nun ein OBDD für eine Funktion $f\colon B^n \to B$. Wir nehmen an, dass die Variablenordnung $x_1, x_2, \ldots, x_n$ ist; das ist keine wesentliche Einschränkung, wir können ja sonst die Variablen einfach umbenenennen. In diesem OBDD betrachten wir einen Knoten v, die Markierung von v sei x_i. Wir erreichen v vom Startknoten aus über einen Pfad, auf dem einige Variablen aus $\{x_1, x_2, \ldots, x_{i-1}\}$ als Knotenmarkierungen vorkommen. Bei jeder vorkommenden Variablen ist durch den Pfad festgelegt, ob sie fest mit 0 oder fest mit 1 belegt wird. Es wird an v also auf jeden Fall eine Subfunktion von f dargestellt. Es kann im OBDD verschiedene Pfade geben, die vom Startknoten zu v führen, so dass an v eventuell verschiedene Subfunktionen dargestellt

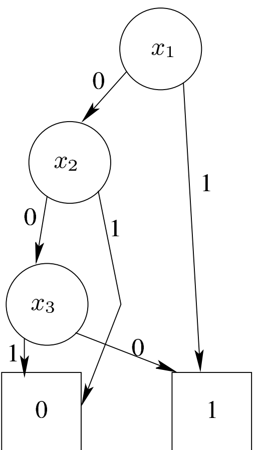


Abbildung 4: Das reduzierte πOBDD für f_{bsp} zu $\pi = x_1, x_2, x_3$

werden. Weil jeder Knoten aber nur eine Funktion darstellen kann, müssen diese verschiedenen Subfunktionen gleich sein. Wir formulieren das noch einmal ganz genau: Weil verschiedene Pfade vom Startknoten zum Knoten v verschiedenen Belegungen von Variablen entsprechen, erhalten wir in diesem Sinne verschiedene Subfunktionen. Weil ein Knoten v aber immer nur eine Funktion darstellen kann, müssen diese verschiedenen Subfunktionen für alle Eingaben jeweils gleiche Funktionswerte liefern, in diesem Sinn also gleich sein.

Wir werden jetzt in einem ersten Schritt ein Hilfsresultat beweisen. Wir zeigen, dass es jedenfalls ein kleinstes πOBDD gibt, sagen aus, welche Knoten es enthält und dass es bis auf Isomorphie eindeutig ist. Danach wird es uns leicht fallen zu beweisen, dass wir dieses kleinste πOBDD durch erschöpfende Anwendung der beiden Reduktionsregeln erhalten. Weil es zu jeder Funktion f und jeder Variablenordnung π ein eindeutiges kleinstes πOBDD gibt, das f darstellt, nennen wir dieses kleinste πOBDD auch das reduzierte πOBDD zu f. Es lohnt sich sicher darüber nachzudenken, ob es für eine Funktion f verschieden große reduzierte OBDDs geben kann, wenn man verschiedene Variablenordnungen betrachtet. Wir werden auf diese Frage noch im Abschnitt 4.3 zurückkommen.

Lemma 10. *Sei $f\colon B^n \to B$ eine boolesche Funktion über den Variablen $x_1, x_2, \dots, x_n$, π eine Variablenordnung. Für $i \in \{1, 2, \dots, n\}$ sei S_i die Menge der Subfunktionen von f, die man durch Konstantsetzen von $x_1, x_2, \dots, x_{i-1}$ erhält und die essentiell von x_i abhängen.*
Es gibt bis auf Isomorphie genau ein πOBDD minimaler Größe für f, es enthält für jedes $i \in \{1, 2, \dots, n\}$ genau $|S_i|$ mit x_i markierte Knoten.

Beweis. Wir dürfen annehmen, dass die Variablenordnung π gerade x_1, x_2, $\dots$, x_n ist, andernfalls können wir geeignet umbenennen. Wir zeigen nun zunächst konstruktiv, dass es tatsächlich ein solches πOBDD gibt, danach überlegen wir uns, warum es minimale Größe hat und eindeutig ist bis auf Isomorphie.

Wenn die Funktion f eine konstante Funktion ist, dann besteht ein minimales πOBDD offenbar nur aus der entsprechenden Senke und die Aussage ist richtig, weil alle S_i leer sind. Andernfalls legen wir zwei verschieden markierte Senken und für jede Funktion $g \in S_i$ einen mit x_i markierten Knoten an, der als Null-Nachfolger einen Knoten für die Funktion $g_{|x_i=0}$ und als Eins-Nachfolger einen Knoten für die Funktion $g_{|x_i=1}$ bekommt. Diese Knoten gibt es auch tatsächlich: Entweder ist eine solche Funktion $g_{|x_i=c}$ eine konstante Funktion, dann ist die entsprechende Senke der gewünschte Knoten. Andernfalls gibt es eine Variable x_j mit $j > i$, so dass $g_{|x_i=c}$ essentiell von x_j abhängt. Wir finden also einen solchen Knoten, der zur entsprechenden Funktion aus S_j korrespondiert.

Wir können leicht zeigen, dass wir auf diese Weise wirklich ein πOBDD erhalten, dass f berechnet. Wir zeigen dass induktiv „von unten nach oben“ im πOBDD für jeden Knoten. Der Induktionsanfang ist gesichert, da die beiden Senken die gewünschten Funktionen darstellen. Betrachten wir nun einen Knoten v für eine Funktion $g \in S_i$. Die Induktionsvoraussetzung sichert, dass die beiden Nachfolger die Funktionen $g_{|x_i=0}$ und $g_{|x_i=1}$ berechnen. Also wird an v die Funktion $\overline{x_i}\, g_{|x_i=0} \vee x_i\, g_{|x_i=1} = g$ berechnet wie behauptet.

Wir zeigen nun durch einen Widerspruchsbeweis, dass das so konstruierte πOBDD minimale Größe hat. Wir nehmen also an, dass es ein πOBDD mit weniger Knoten gibt, das die gleiche Funktion f berechnet. Dann gibt es aber ein $i \in \{1, 2, \dots, n\}$ und eine Funktion $g \in S_i$, so dass es keinen mit x_i markierten Knoten gibt, an dem die Funktion g repräsentiert wird. Wir betrachten den Pfad vom Startknoten, welcher der Konstantsetzung zu g entspricht. Der Pfad endet an einem Knoten v. Wenn dieser Knoten v nicht mit x_i markiert ist, kann an diesem Knoten nicht g repräsentiert werden, da g essentiell von x_i abhängt und im Teil-OBDD mit Startknoten v die Variable x_i gar nicht vorkommt. Wenn der Knoten v mit x_i markiert ist, dann wird aber eine von g verschiedene Subfunktion an diesem Knoten repräsentiert, es gibt also eine Belegung der Variablen, für die der falsche Funktionswert berechnet wird. Folglich kann dieses kleinere πOBDD nicht die Funktion f repräsentieren.

Wir haben jetzt also charakterisiert, wie ein minimales πOBDD für f aussieht: Es enthält für jede Subfunktion $g \in S_i$ einen mit x_i markierten Knoten, an dem g repräsentiert wird, der Null-Nachfolger ist ein Knoten für $g_{|x_i=0}$, der Eins-Nachfolger ein Knoten für $g_{|x_i=1}$. Ist ein πOBDD für f nicht isomorph zu diesem πOBDD, so muss es größer sein. □

Der Beweis von Lemma 10 war nicht ganz kurz und nicht ganz einfach. Die Mühe hat sich aber gelohnt: Wir können jetzt Satz 9 leicht beweisen.

Beweis von Satz 9. Wir überlegen uns zunächst, dass die Anwendung der Reduktionsregeln die dargestellte Funktion nicht verändert. Das ist für die Eliminationsregel klar, weil Knoten mit gleichem Null- und Eins-Nachfolger offenbar keinen Einfluss auf einen berechneten Funktionswert haben können. Bei der Verschmelzungsregel ist das ähnlich offensichtlich: Wenn in zwei mit x_i markierten Knoten gleiche Belegungen von x_i zum gleichen Nachfolger führen, so macht es keinen Unterschied, welchen der beiden Knoten man erreicht, man kann diese beiden Knoten also auch zu einem verschmelzen.
Wir müssen nun noch zeigen, dass ein πOBDD minimal ist, wenn keine Reduktionsregel mehr angewendet werden kann. Dafür beweisen wir umgekehrt, dass ein πOBDD, das nicht minimal ist, Anwendung einer Reduktionsregel erlaubt. Für konstante Funktionen ist das sehr einfach, weil ein πOBDD dafür nur eine Senke enthalten muss. Enthält ein πOBDD mehrere Senken mit gleicher Bezeichnung, so ist offenbar die Verschmelzungsregel anwendbar. Wir gehen also jetzt davon aus, dass wir ein nichtminimales πOBDD für eine nichtkonstante Funktion f haben, das genau eine mit 0 und genau eine mit 1 markierte Senke enthält. Wie gewohnt nehmen wir an, dass die Variablenordnung $x_1, x_2, \ldots, x_n$ ist.
Weil das πOBDD nicht minimal ist, gibt es ein $i \in \{1, 2, \ldots, n\}$, so dass es mehr als $|S_i|$ mit x_i markierte Knoten gibt. Wenn es mehrere solche Werte i gibt, betrachten wir den größten. Es gibt einen mit x_i markierten Knoten v, der entweder die gleiche essentiell von x_i abhängende Funktion repräsentiert wie ein anderer mit x_i markierter Knoten, oder der Knoten v repräsentiert eine Funktion, die nicht essentiell von x_i abhängt. Sei g die an diesem Knoten v repräsentierte Funktion. Weil wir ein maximales i betrachten, ist das πOBDD unterhalb von v isomorph zum reduzierten πOBDD. Wenn v eine nicht essentiell von x_i abhängende Funktion repräsentiert, gilt $g_{|x_i=0} = g_{|x_i=1}$. Weil das πOBDD unterhalb von v dem reduzierten πOBDD entspricht, sind Null- und Eins-Nachfolger von v gleich und die Eliminationsregel ist anwendbar. Wenn v eine essentiell von x_i abhängende Funktion repräsentiert, so gibt es einen ebenfalls mit x_i markierten Knoten v', der ebenfalls g repräsentiert. Weil wir unterhalb von v und v' Isomorphie zum reduzierten πOBDD haben,

müssen die Null- und Eins-Nachfolger von v und v' gleich sein, so dass die Verschmelzungsregel anwendbar ist. □

Beim praktischen Einsatz von OBDDs steht offensichtlich die Frage, wie man überhaupt zu einem πOBDD für eine boolesche Funktion $f\colon B^n \to B$ kommt, am Anfang. Das kann ganz verschieden sein und hängt natürlich davon ab, wie die Funktion f gegeben ist. Wir werden das im Folgenden besprechen und dabei davon ausgehen, dass die Variablenordnung π als $x_1, x_2, \ldots, x_n$ festgelegt ist. Obwohl das offenbar ein Spezialfall ist, gelten unsere Überlegungen doch für alle möglichen Variablenordungen: die Argumente für eine andere Variablenordnung ergeben sich einfach und direkt durch Umbenennung der Variablen. Dass unsere Ausführungen sich auf die spezielle Variablenordnung $x_1, x_2, \ldots, x_n$ beziehen, schränkt also ihre Allgemeingültigkeit nicht ein. Weil das häufig vorkommt, hat sich für diesen Sachverhalt („ohne Beschränkung der Allgemeinheit“) sogar eine Abkürzung eingebürgert. Wir schreiben kurz: O. B. d. A. sei $\pi = x_1, x_2, \ldots, x_n$.
Allen Darstellungsformen boolescher Funktionen, die wir besprochen haben, ist gemeinsam, dass für eine Belegung der Variablen $x_1, x_2, \ldots, x_n$ der Funktionswert $f(x_1, x_2, \ldots, x_n)$ leicht auszurechnen ist. Wir können darum ein πOBDD für f erhalten, indem wir einen vollständigen binären Baum über den Variablen aufbauen und unten an den Senken die entsprechenden Belegungen eintragen. Wir beginnen also mit dem Startknoten x_1, der bekommt zwei x_2-Knoten als Nachfolger, die bekommen jeweils zwei x_3-Knoten als Nachfolger, so dass wir vier x_3-Knoten haben, die dann jeweils zwei, zusammen folglich acht x_4-Knoten als Nachfolger bekommen. Das setzt sich fort, bis wir auf der vorletzten Ebene schließlich 2^{n-1} x_n-Knoten haben, die zusammen 2^n ausgehende Kanten zu Senken haben. Das so erstellte πOBDD hat insgesamt $\sum\limits_{i=0}^{n-1} 2^i = 2^n - 1$ mit Variablen beschriftete Knoten. Anschließend können wir das πOBDD durch Anwendung der beiden Reduktionsregeln so lange verkleinern, bis wir das reduzierte πOBDD erhalten. Wenn die Funktion f ursprünglich in einer Form gegeben war, die nicht viel kleiner ist (also zum Beispiel als Wertetabelle oder Wertevektor), so ist so ein Vorgehen sicher in Ordnung. Wenn die Funktion f ursprünglich viel kompakter repräsentiert war, das reduzierte πOBDD aber auch in etwa 2^n Knoten hat, so ist auch noch nichts auszusetzen. Wenn aber die Funktion f ursprünglich kompakt repräsentiert war (zum Beispiel durch eine kleine DNF oder in einem kleinen Schaltnetz, eine Repräsentation, die wir im Anschluss im Abschnitt 4.3 kennenlernen werden) und das reduzierte πOBDD ebenfalls klein ist, so ist ein Weg, der auf jeden Fall mit einem πOBDD mit mehr als 2^n Knoten startet, nicht hinnehmbar. Als Alternative diskutieren wir darum jetzt eine Metho-

de, bei der zumindest die Hoffnung besteht, auf dem Weg zum reduzierten πOBDD keine Schritte zu machen, die wesentlich größere „Hilfskonstrukte“ erzeugen.

Es ist klar, wie reduzierte πOBDDs für die beiden konstanten Funktionen (0 und 1) aussehen und wie das reduzierte πOBDD für x_i aussieht: es hat einen mit x_i beschrifteten Startknoten, der eine 0-Senke als Null-Nachfolger und eine 1-Senke als Eins-Nachfolger hat. Wenn wir das reduzierte πOBDD für eine boolesche Funktion g haben, so können wir leicht das reduzierte πOBDD für ihre Negation $\neg g$ angeben: wir müssen nur die Beschriftung der beiden Senken vertauschen. Jetzt werden wir beschreiben, wie wir aus einem πOBDD für g und einem πOBDD für h ein πOBDD für $g \otimes h$ berechnen, dabei ist $\otimes$ eine beliebige boolesche Funktion $\otimes \colon B^2 \to B$, also zum Beispiel $\wedge$, $\vee$ oder $\oplus$. Wir nennen diesen Schritt OBDD-Synthese.

Wir beschreiben die OBDD-Synthese als einen rekursiven Algorithmus, der als Eingabe zwei πOBDDs für g und h sowie die Verknüpfung $\otimes$ bekommt. Es ist wesentlich, dass beide OBDDs πOBDDs zur gleichen Variablenordnung π sind. Der Algorithmus liefert als Ergebnis einen Knoten (mit anhängenden Kanten, natürlich), der Startknoten des πOBDDs für $g \otimes h$ ist.

Vor der formalen Beschreibung hilft es, eine bildliche Vorstellung vom Ablauf zu haben. Wir können zu einer Belegung der Variablen $x_1, x_2, \dots, x_n$ den Funktionswert $(g \otimes h)(x_1, x_2, \dots, x_n)$ berechnen, indem wir uns die beiden πOBDDs für g und h nebeneinander vorstellen und beide parallel auswerten. Wir erreichen dann irgendwann in beiden πOBDDs Senken mit Werten $g(x_1, x_2, \dots, x_n)$ und $h(x_1, x_2, \dots, x_n)$ und können $(g \otimes h)(x_1, x_2, \dots, x_n) = g(x_1, x_2, \dots, x_n) \otimes h(x_1, x_2, \dots, x_n)$ leicht berechnen. Wir brauchen also ein πOBDD, das einen solchen Paralleldurchlauf durch die beiden πOBDDs für g und h simuliert. Weil in einem OBDD Variablen ausgelassen werden können, müssen wir gegebenenfalls in einem OBDD „warten“, damit die Auswertung parallel und synchron bleibt. Jetzt wird auch klar, warum wir πOBDDs zur gleichen Variablenordnung brauchen.

Wir rufen unseren Algorithmus OBDD-Synthese$(v, w, \otimes)$ auf, dabei ist v der Startknoten des πOBDD für g und w der Startknoten des πOBDD für h. Falls v keine Senke ist, bezeichne v_0 den Null-Nachfolger von v und v_1 den Eins-Nachfolger von v. Entsprechend bezeichnen w_0 und w_1 die Null- und Eins-Nachfolger von w, falls w keine Senke ist. Die Markierung von v sei m_v, die Markierung von w sei m_w.

Wir unterscheiden jetzt sechs Fälle nach den Markierungen m_v und m_w. Tatsächlich könnte man auch kompakter nur drei Fälle unterscheiden, wir opfern hier aber die Kompaktheit einer eindeutigen und ausführlichen Darstellung.

1. Fall: $m_v = m_w = x_i$ Wir erzeugen einen Knoten, der die Markierung x_i erhält. Null-Nachfolger wird der Knoten, der durch Aufruf von OBDD-Synthese(v_0, w_0, $\otimes$) erzeugt wird. Eins-Nachfolger wird der Knoten, der durch Aufruf von OBDD-Synthese(v_1, w_1, $\otimes$) erzeugt wird.

2. Fall: $m_v = x_i$, $m_w = x_j$, x_i liegt vor x_j gemäß der Variablenordnung π Dies ist der Fall, in dem wir in w warten müssen, weil x_i ausgelassen ist. Wir erzeugen einen Knoten, der die Markierung x_i erhält. Null-Nachfolger wird der Knoten, der durch Aufruf von OBDD-Synthese(v_0, w, $\otimes$) erzeugt wird. Eins-Nachfolger wird der Knoten, der durch Aufruf von OBDD-Synthese(v_1, w, $\otimes$) erzeugt wird.

3. Fall: $m_v = x_i$, $m_w = x_j$, x_j liegt vor x_i gemäß der Variablenordnung π Dieser Fall ist symmetrisch zum zweiten Fall. Wir erzeugen einen Knoten, der die Markierung x_j erhält. Null-Nachfolger wird der Knoten, der durch Aufruf von OBDD-Synthese(v, w_0, $\otimes$) erzeugt wird. Eins-Nachfolger wird der Knoten, der durch Aufruf von OBDD-Synthese(v, w_1, $\otimes$) erzeugt wird.

4. Fall: $m_v = x_i$, $m_w \in \{0, 1\}$ Dieser Fall entspricht dem zweiten Fall, in gewisser Weise liegt eine Konstante als Markierung „hinter“ jeder Variablen. Wir erzeugen einen Knoten, der die Markierung x_i erhält. Null-Nachfolger wird der Knoten, der durch Aufruf von OBDD-Synthese(v_0, w, $\otimes$) erzeugt wird. Eins-Nachfolger wird der Knoten, der durch Aufruf von OBDD-Synthese(v_1, w, $\otimes$) erzeugt wird.

5. Fall: $m_v \in \{0, 1\}$, $m_w = x_j$ Dieser Fall ist symmetrisch zum vierten Fall. Wir erzeugen einen Knoten, der die Markierung x_j erhält. Null-Nachfolger wird der Knoten, der durch Aufruf von OBDD-Synthese(v, w_0, $\otimes$) erzeugt wird. Eins-Nachfolger wird der Knoten, der durch Aufruf von OBDD-Synthese(v, w_1, $\otimes$) erzeugt wird.

5. Fall: $m_v \in \{0, 1\}$, $m_w \in \{0, 1\}$ Wir erzeugen einen Knoten, der eine Senke wird. Er erhält die Markierung $m_v \otimes m_w$.

Manchmal kennt man den Wert einer Verknüpfung $m_v \otimes m_w$ schon dann, wenn man aus $\{m_v, m_w\}$ nur einen Wert kennt. Das ist zum Beispiel der Fall, wenn $\otimes = \vee$ gilt und ein Wert 1 ist. Dann ist klar, dass die Verknüpfung 1 liefert unabhängig vom anderen Wert. Entsprechendes gilt für $\wedge$ und den Wert 0. In diesem Fall kann die Rekursion schon früher, also im vierten oder fünften Fall abgebrochen werden. Es wird dann direkt die entsprechende Senke erzeugt.
Man kann den Algorithmus OBDD-Synthese durch den geschickten Einsatz von Datenstrukturen so implementieren, dass direkt das reduzierte πOBDD

erzeugt wird, ohne dass der Zeitaufwand wesentlich größer wird. Weil Datenstrukturen und Programmierung aber nicht Gegenstand der Vorlesung „Rechnerstrukturen" sind, vertiefen wir das hier nicht.
Wir haben gesehen, dass die OBDD-Synthese nur für πOBDDs zur gleichen Variablenordnung π funktioniert. Das legt die Frage nahe, ob die Verwendung unterschiedlicher Variablenordnungen für verschiedene Funktionen überhaupt nötig ist. Vielleicht können wir uns einfach auf eine feste Variablenordnung einigen, die wir immer verwenden? Wir werden uns später an einem konkreten Beispiel davon überzeugen, dass das kein praktikables Vorgehen ist, weil die Wahl der Variablenordnung dramatische Auswirkungen auf die OBDD-Größe haben kann.
OBDDs sind für praktische Anwendungen so wichtig, weil man viele wichtige Operationen effizient mit ihnen durchführen kann. Wir wollen exemplarisch vier verschiedene Operationen ansprechen.
Manchmal möchte man zu einer booleschen Funktion $f\colon B^n \to B$ eine Subfunktion $f_{x_i=c}$ berechnen. Ist ein OBDD zu f gegeben, kann das effizient durchgeführt werden, indem man einmal alle Knoten des OBDDs durchläuft und dabei jede Kante, die auf einen x_i-Knoten zeigt, so ändert, dass sie beim c-Nachfolger des x_i-Knotens endet. Man wird mit Hilfe der Kenntnisse aus der Vorlesung „Datenstrukturen, Algorithmen und Programmierung 2" sehen, dass sich das in einer Zeit realisieren lässt, die proportional zur Größe des OBDD ist.
Möchte man zu einem gegebenen OBDD eine Eingabe bestimmen, die den Funktionswert 1 berechnet, so kann man sich zunächst im reduzierten OBDD davon überzeugen, dass es überhaupt eine 1-Senke gibt. Andernfalls stellt das OBDD die Nullfunktion dar und es gibt keine solche Eingabe. Gibt es eine 1-Senke, so folgt man den Kanten in umgekehrter Richtung, bis man die Wurzel erreicht und merkt sich jeweils, wie man den Kanten gefolgt ist: Geht man von einem Knoten über eine c-Kante zu einem x_i-Knoten, dann merkt man sich, dass $x_i = c$ zu setzen ist. Gibt es mehr als eine Kante, kann man sich eine beliebige Kante aussuchen. Jede Kante, der man auf diese Weise folgt, legt also die Belegung einer Variablen fest. Dass es in einem πOBDD eine feste Variablenfolge gibt, in der die Knoten besucht werden, sichert uns, dass wir keine Variable auf dem Weg doppelt (und damit ja potenziell widersprüchlich) belegen. Beim Erreichen der Wurzel haben wir also die Variablen, denen wir auf dem Pfad von der 1-Senke zur Wurzel begegnet sind, fest mit Werten belegt. Die noch unbelegten Variablen können beliebig belegt werden. Startet man statt in der 1-Senke in der 0-Senke, erhält man ganz analog eine Belegung, die den Funktionswert 0 erzeugt.
Man kann sogar relativ leicht zählen, wie viele Eingaben insgesamt den Funktionswert 1 erzeugen. Dazu durchlaufen wir das OBDD und bestimmen für

jeden Knoten, wie viele Belegungen dazu führen, dass dieser Knoten bei der Auswertung besucht wird. Wenn wir diese Anzahl für die 1-Senke bestimmt haben, wissen wir, wie viele Eingaben den Funktionswert 1 erzeugen. Für eine boolesche Funktion $f\colon B^n \to B$ gibt es insgesamt 2^n verschiedene Belegungen der n Variablen. Wir beginnen in einem OBDD für f am Startknoten, der für alle 2^n Belegungen erreicht wird, und durchlaufen das OBDD ebenenweise von oben nach unten, es müssen also beim Erreichen eines mit x_i markierten Knotens schon alle Knoten vorher besucht worden sein, die mit einer gemäß der Variablenordnung π vor x_i stehenden Variablen markiert sind. Initial setzen wir die Anzahl der Eingaben z, für die ein Knoten erreicht wird, auf $z = 0$, nur für den Startknoten ist dieser Wert wie erwähnt $z = 2^n$. An jedem Knoten außer den Senken teilen sich die Belegungen in zwei gleich große Mengen, ist der Knoten mit x_i markiert, in die Menge der Belegungen, in denen x_i mit 0 belegt ist und in die Menge der Belegungen, in denen x_i mit 1 belegt ist. Wenn wir einen Knoten das erste Mal betreten und er dabei Anzahl z hat, so addieren wir zu den Anzahlen seiner beiden Nachfolger jeweils $z/2$ hinzu. Am Ende können wir an der Einssenke die Anzahl z der Eingaben ablesen, die den Funktionswert 1 erzeugen.
Schließlich erwähnen wir noch, dass für zwei Funktionen $f, g\colon B^n \to B$, die jeweils durch πOBDDs gegeben sind, leicht festgestellt werden kann, ob beide Funktionen gleich sind. Dazu reduzieren wir im ersten Schritt die beiden πOBDDs. Wir wissen, dass das reduzierte πOBDD eindeutig ist, es genügt jetzt also wieder bei den beiden Startknoten beginnend parallel durch beide πOBDDs zu laufen und zu verifizieren, dass jeweils über die gleichen Kanten gleich markierte Knoten erreicht werden. Ist das an irgendeiner Stelle nicht der Fall, sind die beiden dargestellten Funktionen verschieden.

4.3 Schaltnetze

Bis jetzt haben wir uns zwar schon eine Menge an Grundlagen erarbeitet, sind aber noch nicht konkret in Richtung „Hardware“ gegangen. Das wollen wir nun ändern. Wir haben schon in der Einleitung besprochen, dass wir als „untere Grenze“ unserer Beschreibungen logische Bausteine verwenden wollen. Natürlich werden diese logischen Bausteine in der Praxis physikalisch realisiert, Abbildung 5 zeigt als Beispiel die Realisierung der booleschen Funktion NAND mit Hilfe von zwei Transistoren. Wir sehen daran exemplarisch, wie einfach und wenig aufwändig die Realisierung boolescher Funktionen in Hardware ist. Allerdings sind die physikalischen und elektrotechnischen Grundlagen dieser Realisierung nicht Gegenstand dieser Vorlesung. Wir abstrahieren in gewisser Weise also von der konkreten Realisierung, müssen aber einige praktische Aspekte im Auge behalten, damit unsere Überlegungen praktische Relevanz haben.

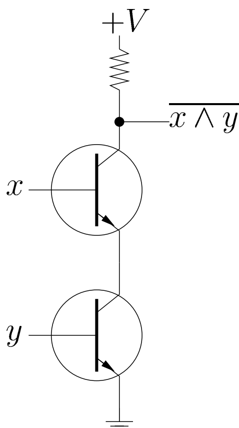


Abbildung 5: Realisierung eines NAND-Gatters

Einfache Grundbausteine, die Gatter genannt werden, haben eine gewisse Anzahl von Eingängen, eine gewisse Anzahl von Ausgängen und berechnen eine boolesche Funktion. Es gibt eine Reihe von gebräuchlichen Symbolen für diese Grundbausteine, die wir in Tabelle 5 zusammengestellt haben. Wir werden hier die Darstellung gemäß DIN EN 60617 (früher DIN 40900) (DIN steht für deutsche Industrienorm) bevorzugen, man sollte sich allerdings auch mit den beiden anderen Darstellungsformen, also der Darstellung nach der noch älteren DIN 40700 und dem von der IEEE (Institute of Electrical and Electronics Engineers) vorgeschlagenen Symbolen vertraut machen. Vor allem die letztgenannte Darstellung ist in englischsprachiger Literatur sehr häufig zu finden. Auch im Programm RaVi (http://ls12-www.cs.uni-dortmund.de/ravi) finden diese Symbole Verwendung.
Die Anzahl der Eingänge eines Gatters nennt man seinen Fan-In, die Anzahl der Ausgänge eines Gatters nennt man seinen Fan-Out. Man kann nun beliebige boolesche Funktionen realisieren, indem man einige solche Gatter geeignet miteinander kombiniert. Diese Kombination von Gattern nennt man ein Schaltnetz; dabei wollen wir aber nicht zulassen, dass ein Kreis entsteht. Die Wege im Schaltnetz sind gerichtet (allerdings verzichten wir im Gegensatz zur Darstellung von OBDDs darauf, den Kanten Pfeile zu geben): sie führen in Eingänge hinein und aus Ausgängen heraus. Man zeichnet Schaltnetze dabei stets so, dass die Wege entweder von links nach rechts oder von oben nach unten führen. Kreisfreiheit bedeutet demnach, dass in einem Schaltnetz für kein Gatter ein (richtig gerichteter) Weg von einem seiner Ausgänge zu einem seiner Eingänge führt.
Wir können nun praktisch beliebige Schaltnetze entwerfen. Für die Realisierung in der Praxis sind jedoch eine Reihe von Dingen zu bedenken. Jedes Gatter ist ein physikalisches Objekt, das bedeutet unter anderem, dass nur

Funktion	DIN 40700	DIN EN 60617	IEEE
$y = \overline{x_1}$	x_1 y	x_1 1 y	x_1 y
$y = x_1 \wedge x_2 \wedge x_3$	x_1 x_2 x_3 y	x_1 x_2 x_3 & y	x_1 x_2 x_3 y
$y = x_1 \vee x_2 \vee x_3$	x_1 x_2 x_3 y	x_1 x_2 x_3 ≥ 1 y	x_1 x_2 x_3 y
$y = \overline{x_1 \wedge x_2 \wedge x_3}$	x_1 x_2 x_3 y	x_1 x_2 x_3 & y	x_1 x_2 x_3 y
$y = \overline{x_1 \vee x_2 \vee x_3}$	x_1 x_2 x_3 y	x_1 x_2 x_3 ≥ 1 y	x_1 x_2 x_3 y
$y = x_1 \oplus x_2$	x_1 x_2 $\oplus$ y	x_1 x_2 $= 1$ y	x_1 x_2 y
$y = \overline{x_1}\,\overline{x_2} \vee x_1 x_2$	x_1 x_2 $\equiv$ y	x_1 x_2 $\equiv$ y	x_1 x_2 y

Tabelle 5: Symbole für Gatter (Grundbausteine)

eine begrenzte Anzahl von Ein- und Ausgängen ohne zusätzlichen Aufwand realisiert werden kann, dass die Gatter eine gewisse Größe haben, die Verwendung vieler Gatter also zu großen Schaltnetzen führt, die Gatter eine gewisse Schaltzeit haben, Schaltnetze mit langen Wegen also insgesamt lange Schaltzeiten haben (dieser Effekt wird noch durch die Signallaufzeiten auf den Leitungen verstärkt) und natürlich auch jedes Gatter mit Herstellungskosten verbunden ist, so dass Schaltnetze mit vielen Gattern teurer sind als Schaltnetze mit wenigen. Außerdem kann man davon ausgehen, dass mit zunehmender Anzahl von Gattern in der Realität die Fehleranfälligkeit wächst, so dass sich zusätzlich Wartungskosten erhöhen. Nicht zu vernachlässigen sind durchaus auch der Stromverbrauch und die Verlustleistung, vor allem das damit einhergehende Wärmeproblem.
Es gibt also eine Reihe von guten Gründen, warum man möglichst kleine und flache Schaltnetze entwerfen sollte. Wir werden uns diesem Problem etwas ausführlicher im Abschnitt 4.5 widmen. Hier wollen wir nur einige sehr grundsätzliche Überlegungen zum Schaltnetzentwurf machen und uns zwei Beispiele ansehen.
Wir haben uns überlegt, dass es eine große Anzahl von Kenngrößen beim Schaltnetzentwurf gibt, die alle berücksichtigt werden sollten. Oft beschränkt man sich auf zwei wichtige Werte, nämlich die Anzahl der Gatter im Schalt-

netz, die man als Größe des Schaltnetzes bezeichnet, und die maximale Anzahl von Gattern auf einem Weg von einem Eingang zu einem Ausgang im Schaltnetz; diese Anzahl bezeichnet man als Tiefe des Schaltnetzes. Es ist klar, dass die Schaltnetzgröße direkt mit den Herstellungskosten und die Schaltnetztiefe direkt mit der Schaltzeit korrespondiert. Wir wollen also möglichst kleine Schaltnetze mit möglichst geringer Tiefe realisieren.
Wir wissen schon, dass wir jede boolesche Funktion mit einem Schaltnetz der Tiefe 3 realisieren können. Dazu müssen wir nur auf die Darstellung der Funktion in einer Normalform, also in DNF, KNF oder RNF zurückgreifen. In der ersten Stufe des Schaltnetzes können von allen Variablen, die negiert in mindestens einem Minterm (bzw. Maxterm für die KNF) vorkommen, die Negation bereitgestellt werden. In der zweiten Stufe können dann durch geeignete Konjunktionen von Variablen und negierten Variablen alle Minterme bereitgestellt werden (bzw. durch Disjunktionen alle Maxterme), dann wird im letzten Schritt mit einer Verknüpfung all dieser Terme die Funktion realisiert. Im Allgemeinen führt das zu sehr großen und darum unbrauchbaren Schaltnetz-Entwürfen. Wenn jedoch nur sehr wenige Belegungen der Variablen den Funktionswert 0 oder den Funktionswert 1 berechnen, so kann dieses Vorgehen zu guten Realisierungen führen. Wir sehen uns ein einfaches Beispiel an und betrachten wieder die Funktion f_{bsp} (Abbildung 6).
Wir wollen jetzt noch eine weitere Funktion als Beispiel betrachten, wobei es sich diesmal nicht um eine willkürlich gewählte Funktion wie f_{bsp} handeln soll, sondern um eine in der Praxis wichtige Funktion. Die Funktion heißt MUX, sie realisiert einen *Multiplexer*, einen tatsächlich wichtigen Baustein, der es erlaubt, aus einer Anzahl von Datenbits mit Hilfe von Steuersignalen gezielt ein Datenbit auszuwählen. Die Funktion ist nicht auf einer festen Anzahl von Bits definiert, sie wird für eine Anzahl $d \in \mathbb{N}$ von Steuerleitungen und eine daraus resultierende Anzahl 2^d von Datenleitungen beschrieben. Die Datenleitungen heißen $x_0, x_1, \ldots, x_{2^d-1}$, die Steuerleitungen $y_1, \ldots, y_d$. Wir interpretieren die Belegung der d Steuerleitungen als Binärzahl $y = (y_1 y_2 \ldots y_d)_2$, der Funktionswert der Funktion MUX_d ist dann x_y. Aus diesem Grund nennen wir $y_1 y_2 \ldots y_d$ auch Adressbits. Für $d = 3$ geben wir in Tabelle 6 eine vereinfachte Funktionstabelle an, in der wir auf die explizite Wiedergabe der $2^{2^3} = 256$ verschiedenen Belegungen der Datenbits verzichten. Die vollständige Tabelle hat offenbar $2^{11} = 2048$ Zeilen.
Es ist nicht schwer, sich klar zu machen, dass die Funktion MUX_d für genau die Hälfte ihrer 2^{d+2^d} verschiedenen Eingaben den Funktionswert 1 liefert. Darum kann es offenbar schon für kleines d (selbst für $d = 2$ macht das schon keinen Spaß) keinen Sinn ergeben, ein Schaltnetz aus einer Normalform herzuleiten. Wir werden stattdessen für MUX_3 das reduzierte πOBDD zur Variablenordnung $\pi = y_1, y_2, y_3, x_0, x_1, \ldots, x_7$ angeben. Mit ein wenig

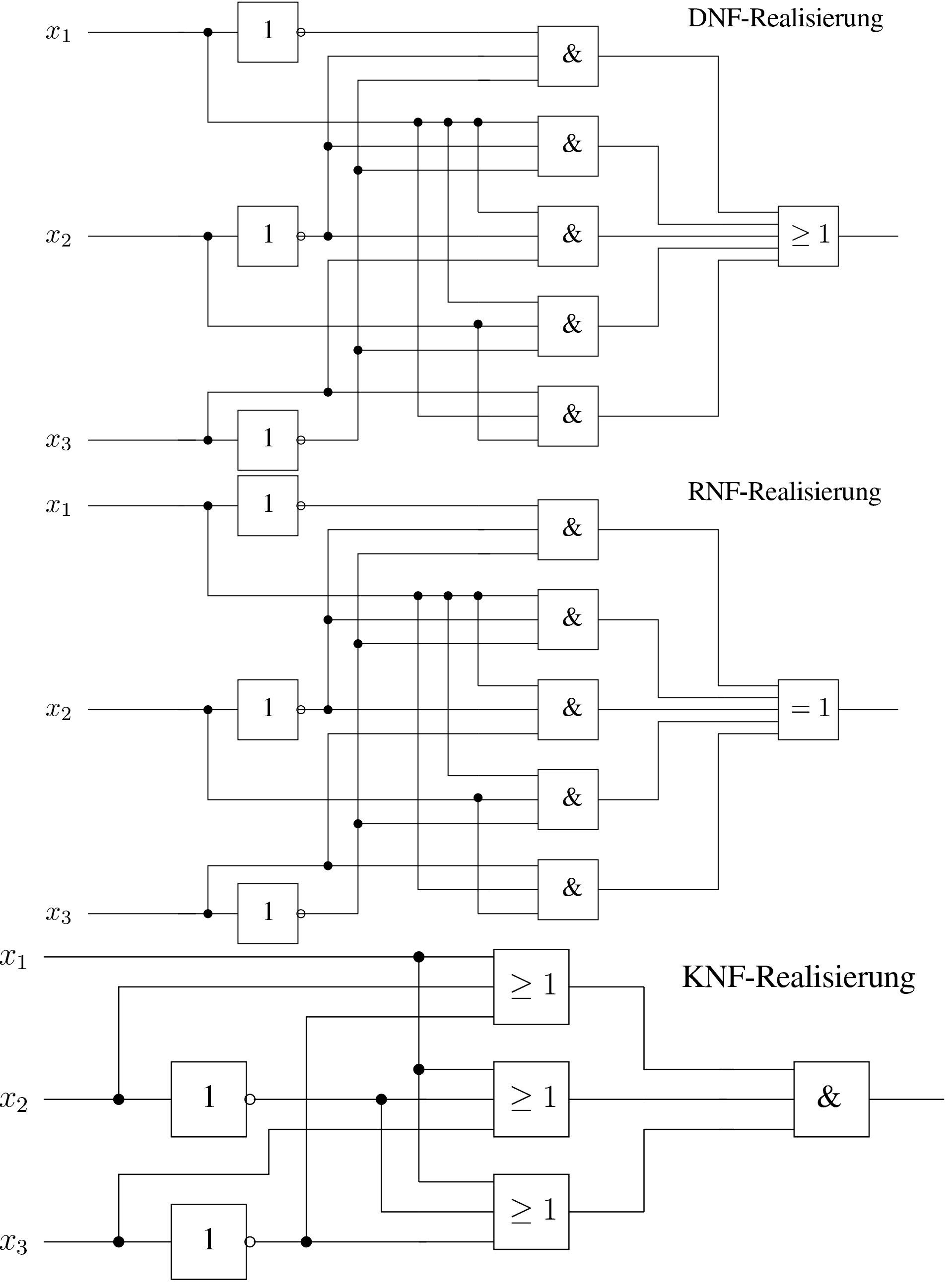


Abbildung 6: Schaltnetze für f_{bsp} basierend auf Normalformen

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

Tabelle 6: Vereinfachte Wertetabelle für MUX_3

Überlegung ist es hier nicht schwer, gleich ein reduziertes πOBDD zu bekommen, ohne den Weg über ein größtes πOBDD, das anschließend durch erschöpfende Anwendung der beiden Reduktions-Regeln reduziert wird, zu gehen.

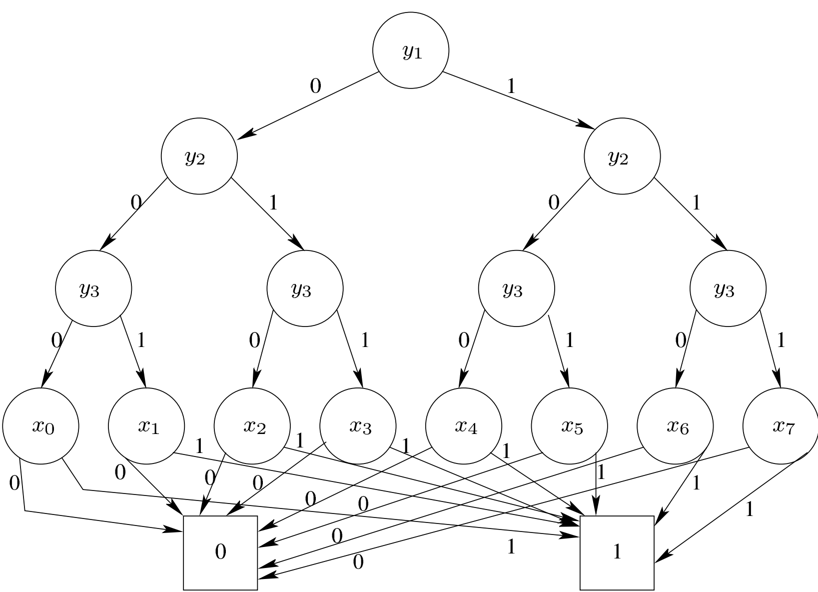


Abbildung 7: Reduziertes πOBDD für MUX_3

Man erkennt direkt, dass das πOBDD in Abbildung 7 tatsächlich reduziert ist: es ist keine der beiden Reduktionsregeln anwendbar. Ist dieses relativ kleine OBDD vielleicht auch für den Schaltnetz-Entwurf hilfreich? Die x_i-Knoten in Abbildung 7 realisieren offenbar jeweils genau die Funktion x_i. Jeder einzelne dieser Knoten wird genau für eine Belegung der y-Variablen erreicht. Das entspricht ja auch genau der Darstellung in der vereinfachten

Wertetabelle 6. Wir können jetzt also Entsprechendes zum Beispiel für den x_0-Knoten durch $x_0\,\overline{y_1}\,\overline{y_2}\,\overline{y_3}$ erreichen. Jetzt haben wir eine Idee für ein dreistufiges Schaltnetz, das mit drei Negationsgattern, acht Und-Gattern und einem Oder-Gatter auskommt, folglich nur Größe 12 und Tiefe 3 hat (Abbildung 8). Der Entwurf allerdings war „ad hoc“ und per Hand – im Wesentlichen also eher unstrukturiert. Außerdem ist der Fan-In des abschließenden Oder-Gatters sehr groß: er beträgt 2^d, wenn MUX_d realisiert werden soll. Wir werden uns im Abschnitt 4.5 noch einmal damit auseinander setzen.

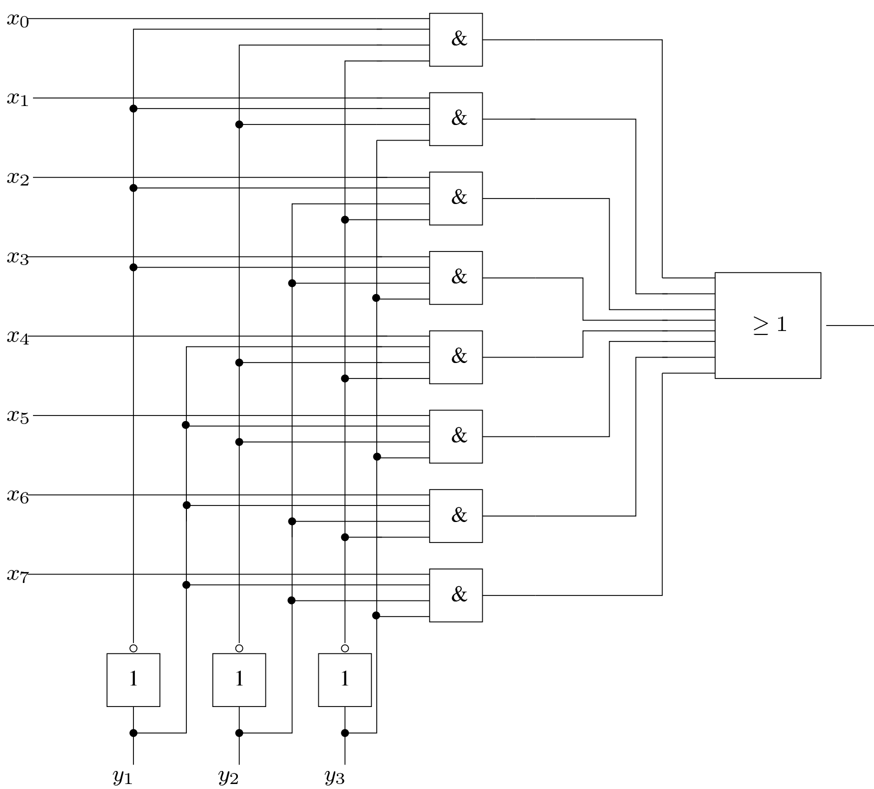


Abbildung 8: Schaltnetz für MUX_3

Wir hatten im Abschnitt 4.1 die Frage nach den Auswirkungen der Wahl der Variablenordnung auf die OBDD-Größe offen gelassen und wollen das jetzt am Beispiel von MUX_d nachholen. Wir haben schon gesehen, dass für die Variablenordnung $y_1, y_2, \ldots, y_d, x_0, x_1, \ldots, x_{2^d-1}$ das reduzierte πOBDD Größe

$$\underbrace{\left(\sum_{i=1}^{d} 2^{i-1}\right)}_{\text{Anzahl } y\text{-Knoten}} + \underbrace{2^d}_{\text{Anzahl } x\text{-Knoten}} + \underbrace{2}_{\text{Anzahl Senken}} = 2^d - 1 + 2^d + 2 = 2^{d+1} + 1$$

hat. Betrachten wir jetzt die Variablenordnung $\pi' = x_0, x_1, \ldots, x_{2^d-1}, y_1, y_2, \ldots, y_d$. Wie groß ist das reduzierte π'OBDD zu dieser Variablenordnung? Wir machen uns im Folgenden klar, dass es auf jeden Fall viel größer als $2^{d+1} + 1$ ist, ohne seine Größe genau zu bestimmen. Wir behaupten, dass für zwei beliebige verschiedene Belegungen der x-Variablen im π'OBDD für MUX_d auf jeden Fall zwei verschiedene Knoten erreicht werden müssen. Wir zeigen die Richtigkeit dieser Behauptung, indem wir das Gegenteil annehmen und dann folgern, dass das betrachtete π'OBDD nicht korrekt MUX_d berechnen kann. Wir führen also einen Beweis durch Widerspruch. Seien jetzt $x = (x_0, x_1, \ldots, x_{2^d-1})$ und $x' = (x'_0, x'_1, \ldots, x'_{2^d-1})$ zwei beliebige unterschiedliche Belegungen der x-Variablen. Weil sie unterschiedlich sind, gibt es einen Index i, so dass $x_i \neq x'_i$ gilt. Seien $y_1, y_2, \ldots, y_d$ nun so gewählt, dass $(y_1\, y_2 \cdots y_d)_2 = i$ gilt. Wenn in einem π'OBDD vom Startknoten ausgehend nach Lesen aller Datenvariablen gemäß den Belegungen x und x' der gleiche Knoten erreicht wird, so muss nach Lesen der Belegung von $y_1, y_2, \ldots, y_d$ die gleiche Senke erreicht werden. Die erreichte Senke kann ja nur noch von der Belegung der y-Variablen abhängen, die (in Abhängigkeit von x und x') fest gewählt ist. Es sind aber $\text{MUX}_d(x, y_1, y_2, \ldots, y_d) = x_i$ und $\text{MUX}_d(x', y_1, y_2, \ldots, y_d) = x'_i$ mit $x_i \neq x'_i$. Folglich berechnet das betrachtete π'OBDD nicht MUX_d.

Wir wissen jetzt, dass das reduzierte π'OBDD für MUX_d „unterhalb“ der Datenvariablen mindestens 2^{2^d} Knoten hat. Außerdem muss es für die Datenvariablen einen vollständigen binären Baum enthalten. Es besteht also mindestens aus $\left(\sum_{i=0}^{2^d-1} 2^i\right) + 2^{2^d} = 2^{2^d} - 1 + 2^{2^d} = 2^{1+2^d} - 1$ Knoten. Wir sehen, dass die Wahl einer guten Variablenordnung also ganz entscheidend für die OBDD-Größe sein kann. Schon für MUX_3 können wir ein OBDD der Größe 17 erhalten oder, bei ungünstiger Wahl der Variablenordnung, mindestens 511 Knoten benötigen. Für MUX_8 macht die Wahl der Variablenordnung den Unterschied zwischen gut handhabbar (513 Knoten) und keineswegs darstellbar ($> 2 \cdot 10^{77}$ Knoten) aus.

4.4 Rechner-Arithmetik

Computer sind vielfältig einsetzbar, heutzutage werden sie oft als „Multimedia-Geräte“ betrachtet. Ursprünglich sind Computer (wie ihr Name ja auch schon verrät) als Rechenmaschinen konzipiert und realisiert worden. Auch heute noch stellt das Rechnen einen sehr wichtigen Anwendungsbereich von Computern dar, selbst in Bereichen, in denen das für einen Anwender vielleicht nicht offensichtlich ist. Wir werden uns darum hier diesem Thema

stellen und dabei zunächst auf die in Kapitel 3 behandelten Zahlendarstellungen zurückkommen. Wie kann man in diesen Darstellungen rechnen? Dabei interessieren uns für die ganzen Zahlen hier nur Addition, Subtraktion und Multiplikation.
Man macht sich zunächst leicht klar, dass für das Rechnen in b-adischen Darstellungen im Wesentlichen die gleichen Rechenregeln gelten unabhängig von b. Wir betrachten als Beispiel die Addition von zwei Binärzahlen und gehen nach der uns allen bekannten Schulmethode für die schriftliche Addition vor.

	1	0	1	0	1	1	0	0
+			1	1	1	0	1	0
	1	1	1	0	0	1	1	0

An den hintersten drei Stellen geschieht nichts Aufregendes, da $0 + 0 = 0$ und $1 + 0 = 0 + 1 = 1$ gilt. An der vierten Stelle von hinten haben wir $1 + 1$, das Ergebnis ist 10, erzeugt also einen Übertrag. Dadurch haben wir an der folgenden Stelle $0 + 1 + 1$ und wiederum einen Übertrag. Danach ergibt sich $1 + 1 + 1$, was wiederum einen Übertrag ergibt, allerdings nur von 1, das Ergebnis ist ja 11. Wir sehen also, dass als Überträge nur 0 oder 1 auftreten können. Auch bei der Subtraktion können wir nach gleichem Schema vorgehen.
Die Multiplikation könnten wir prinzipiell durch fortgesetzte Addition ersetzen. Wir denken an den Entwurf von Multiplikationsschaltnetzen und können uns sicher nicht vorstellen, auf diese Art schnelle und gute Multiplizierer bauen zu können. Wir sehen uns darum an, was bei der Schulmethode passiert (Abbildung 9).

1	0	1	0	1	1	0	0	·	1	1	1	0	1	0
														0
						1	0	1	0	1	1	0	0	
												0		
				1	0	1	0	1	1	0	0			
			1	0	1	0	1	1	0	0				
		1	0	1	0	1	1	0	0					
	1	0	0	1	1	0	1	1	1	1	1	0	0	0

Abbildung 9: Beispielmultiplikation von Binärzahlen

Wir sehen, dass für die eigentlichen Multiplikationen jeweils entweder nur Zahlen zu kopieren sind (was in Schaltnetzen keine Bausteine erfordert) oder die 0 einzusetzen ist (was in Schaltnetzen ebenfalls keine Bausteine erfordert). Dann sind so viele Zahlen zu addieren, wie die kürzere der beiden Zahlen

lang ist. Wir wollen uns hier (noch) nicht überlegen, wie man das möglichst effizient realisiert und kommen darauf später zurück.

Schaltnetze für die Addition

Die Addition von zwei Binärzahlen fester Länge kann man sicher auch in einem Schaltnetz realisieren. Wir wollen uns überlegen, wie wir dabei vorgehen können. Zunächst brauchen wir eine Funktion, die zwei einzelne Bits als Eingabe bekommt und die Summe sowie einen eventuellen Übertrag berechnet. Wir suchen also eine Funktion $f_{\text{HA}}\colon B^2 \to B^2$, wobei $f_{\text{HA}}(x, y) = (c, s)$ sein soll, also s das Summenbit von $x + y$ und c den Übertrag (das Carrybit) von $x + y$ enthalten soll. Wir können auch leicht eine Funktionstabelle angeben (Tabelle 7).

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

Tabelle 7: Funktionstabelle f_{HA}

Wir lesen ohne Mühe direkt ab, dass $c = x\,y$ und $s = x \oplus y$ gilt. Wir können also auch leicht direkt ein Schaltnetz zur Realisierung angeben (Abbildung 10).

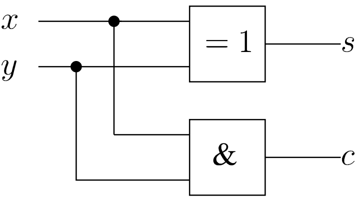


Abbildung 10: Schaltnetz eines Halbaddierers (HA)

Wir nennen einen solches Schaltnetz einen *Halbaddierer*. Er ist offensichtlich perfekt geeignet, die erste Stelle unserer Addition richtig durchzuführen. Aber schon an der zweiten Stelle hakt es: wir haben hier drei Bits zu addieren, die Bits der beiden Zahlen und das Übertragsbit der vorhergehenden Position. Wir brauchen jetzt also eine Funktion $f_{\text{VA}}\colon B^3 \to B^2$, die $f_{\text{VA}}(x, y, c') = (c, s)$ berechnet, also zu $x + y + c'$ das Summenbit s und das Übertragsbit c. Natürlich können wir wieder eine Funktionstabelle angeben (Tabelle 8).

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

Tabelle 8: Funktionstabelle f_{VA}

Das Summenbit nimmt offensichtlich genau dann den Wert 1 an, wenn eine ungerade Zahl von Eingabebits den Wert 1 haben. Wir können darum $s = x \oplus y \oplus c'$ schreiben. Das Carrybit ist etwas kniffliger. Es nimmt genau dann den Wert 1 an, wenn mindestens zwei Einsen in der Eingabe vorkommen, darum ist $c = xy \vee xc' \vee yc'$ sicher richtig. Wenn wir an eine Schaltnetz-Realisierung denken, ist es schöner, wenn man Gatter mehrfach verwenden kann. Zum Beispiel haben wir ein Gatter für $x \oplus y = x\,\overline{y} \vee \overline{x}\,y$, das genau dann eine 1 produziert, wenn genau eine der beiden Eingaben 1 ist. Wenn wir die Konjunktion mit c' bilden, bekommen wir eine 1, wenn genau zwei der Eingaben eine 1 haben und müssen nur noch $x\,y$ zusätzlich abdecken. Es gilt darum auch $c = xy \vee (c'\ (x \oplus y))$. Damit hat das in Abbildung 11 dargestellte Schaltnetz eines *Volladdierers* Größe 5 und Tiefe 3.

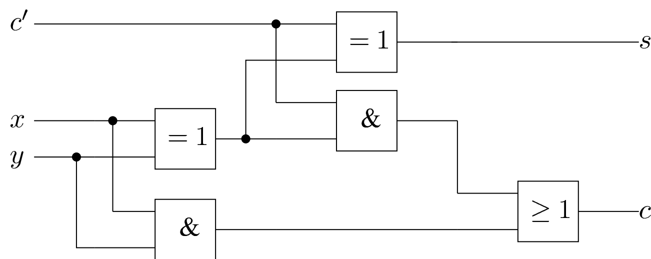


Abbildung 11: Schaltnetz eines Volladdierers (VA)

Wir können jetzt direkt die Schulmethode für die Addition zweier Binärzahlen in ein Schaltnetz übertragen. Wir könnten das für 4-Bit-Zahlen, 8-Bit-Zahlen, 16-Bit-Zahlen, ... machen. Es fällt uns aber genau so leicht, das Prinzip zu verallgemeinern und ein Prinzip-Schaltnetz zur Addition von zwei n-Bit Zahlen ($n \in \mathbb{N}$) angeben. Die Korrektheit des Schaltnetzes (Abbildung 12) ist dabei offensichtlich. Wir gehen davon aus, dass HA den Halbaddierer aus Abbildung 10 bezeichnet und VA_i jeweils den Volladdierer aus Abbildung 11.

Es ist natürlich entscheidend, dass die Summenbits jeweils an den oberen Ausgängen anliegen.

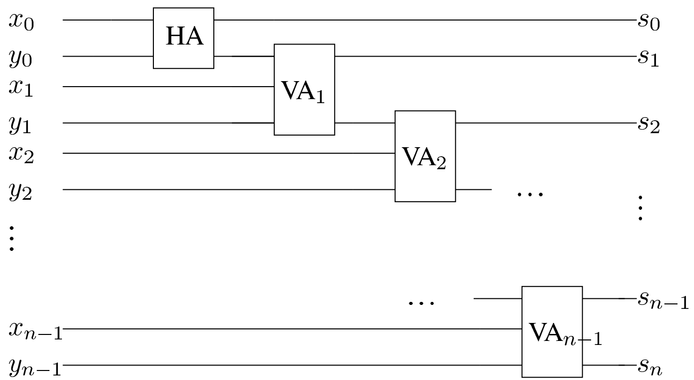


Abbildung 12: Addition von zwei n Bit Zahlen

Der Entwurf (Abbildung 12) sieht überzeugend aus: er ist klar strukturiert, einfach (und darum offensichtlich korrekt) und hat auch nur eine ziemlich kleine Größe: Er besteht aus $5(n-1)+2 = 5n-3$ Bausteinen. Die Tiefe beträgt allerdings $3(n-1)+1 = 3n-2$ Bausteine[4]. Das ist aber nun ganz und gar nicht überzeugend. In Schaltnetzen kann man die Gatter schichtenweise nach ihrer Tiefe anordnen. Alle Gatter einer Schicht können dann gleichzeitig ihre Werte berechnen. In diesem Schaltnetz für die Addition wird aber kaum etwas gleichzeitig berechnet: alle Addierbausteine können nur nacheinander rechnen. Das liegt daran, dass die Übertragsbits erst dann zur Verfügung stehen. Darum ist der Schaltkreis so langsam. Ob das nicht auch schneller geht?

Wir haben schon erkannt, dass das Problem in den Übertragsbits besteht. Wenn wir die Übertragsbits schon früher kannten, könnten die anderen Volladdierer schon früher mit der Berechnung anfangen und wir hätten das Gesamtergebnis schneller berechnet. Muss das nicht möglich sein? Schließlich stehen die Übertrage doch implizit mit der Eingabe schon fest.

Wenn wir zwei Bits x und y addieren, so gibt es drei wesentlich verschiedene Situationen. Die beiden Bits könnten 0 sein, dann kann ein von rechts kommender Übertrag keinen Übertrag erzeugen. An solchen Stellen wird also in gewisser Weise jeder Übertrag eliminiert. Wenn die beiden Bits 1 sind, entsteht ein Übertrag. Ein von rechts kommender Übertrag hat auf den Übertrag

[4] Tatsächlich könnten die beiden Bausteine, die x_i und y_i verknüpfen schon direkt berechnet werden, so dass nur noch eine Tiefe von 2 für jeden der Volladdierer übrig bliebe. Die Gesamttiefe dess Addierers betrüge mit dieser Optimierung also nur $2(n-1)+1 = 2n-1$, wäre damit aber immer noch linear von der Anzahl der zu addierenden Binärzahlen abhängig.

keinen Einfluss, er ändert nur das Summenbit. An solchen Stellen wird also ein Übertrag erzeugt. Schließlich können die beiden Bits unterschiedlichen Wert haben. Dann wird von dort aus kein Übertrag erzeugt, es entsteht aber ein Übertrag, wenn von rechts ein Übertrag kommt. An solchen Stellen wird also ein Übertrag unverändert weitergereicht. Wie können wir das ausnutzen? Wir könnten zunächst für alle Paare x_i, y_i mit einem Halbaddierer ein Summenbit v_i und ein Übertragsbit u_i berechnen. Wir verwenden jetzt andere Bezeichnungen, weil s_i und c_i weiterhin für die richtigen Summenbits und Übertragsbits der Gesamtsumme stehen sollen. Wir haben an der i-ten Stelle einen Übertrag, wenn an irgendeiner Stelle davor (also an einer Stelle j mit $j < i$) ein Übertrag erzeugt wird und an allen Stellen dazwischen (also an allen Stellen k mit $j < k < i$) der Übertrag weitergereicht wird. Es wird an Stelle j ein Übertrag erzeugt, wenn $u_j = 1$ gilt. Ein Übertrag wird an Stelle k weitergereicht, wenn $v_k = 1$ gilt. Wir haben also

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

und können alle Übertragsbits in Tiefe 3 berechnen. Aber das ist etwas unfair gezählt: Beim Schaltnetz aus Abbildung 12 haben alle Bausteine Fan-In 2. Hier haben wir Disjunktionen mit Fan-In n; und wir hatten schon diskutiert, dass das für große n nicht so einfach zu realisieren ist. Wir können aber einen Baustein mit Fan-In l durch ein Schaltnetz mit etwa l Bausteinen mit Fan-In 2 und Tiefe etwa $\log_2 l$ ersetzen, wie man beispielhaft für die Disjunktion in Abbildung 13 sehen kann. Der entstehende Addierer unterscheidet sich von der Schulmethode vor allem dadurch, dass Aufwand betrieben wird, um die Überträge schnell zu berechnen – im Vergleich zur Schulmethode die Überträge korrekt vorherzusagen. Man nennt diesen Addierer darum *Carry-Look-Ahead-Addierer*.

Wir können also zwei n Bit Zahlen in Tiefe etwa $2 \log n$ addieren, was ganz erheblich schneller ist: wir können so zum Beispiel zwei 128-Bit-Zahlen in Tiefe 20 addieren, während wir sonst Tiefe 255 haben. Je länger die Zahlen werden, desto beeindruckender wird der Unterschied. Aber ganz umsonst gibt es diese Beschleunigung leider nicht. Der Addierer nach Schulmethode kommt mit $5n - 3$ Gattern aus. Hier brauchen wir (bei der Verwendung von Gattern mit Fan-In 2) grob geschätzt etwa n^2 Gatter. Für die genannten 128-Bit-Zahlen ergibt sich also ein Mehrbedarf von 15 747 Gattern. Das ist natürlich sehr erheblich. Wir wollen darum jetzt noch einen dritten Versuch unternehmen, zu einem schnellen und kleinen Addierer zu kommen.

Unsere zentrale Struktureinsicht bei der Addition von zwei Binärzahlen ist, dass $(x_i, y_i) = (1, 1)$ einen Übertrag generiert, $(x_i, y_i) = (0, 0)$ einen Übertrag

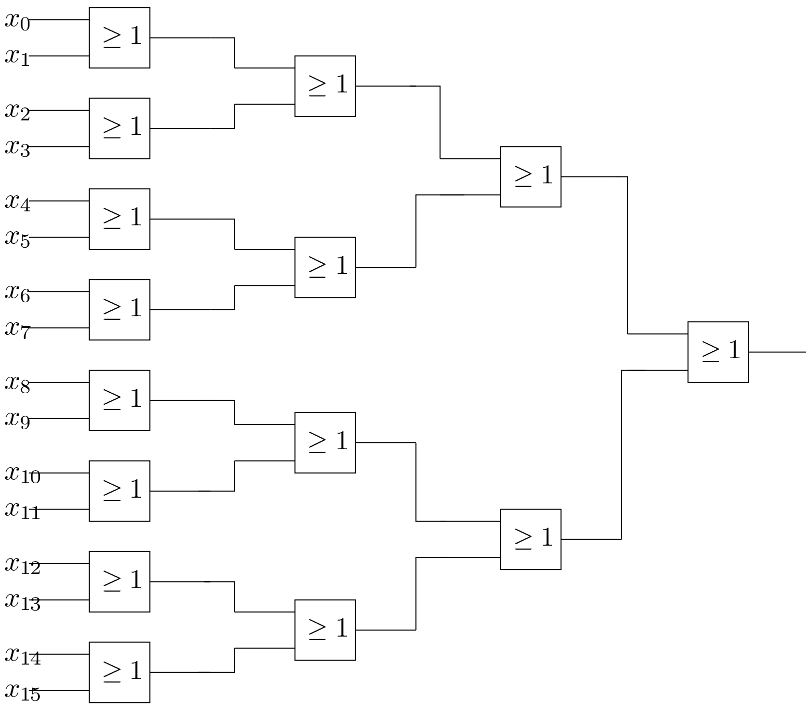


Abbildung 13: Baumartiges Schaltnetz ersetzt Gatter mit großem Fan-In

eliminiert und $(x_i, y_i) \in \{(0,1),(1,0)\}$ einen Übertrag weiter reicht. Wir kürzen diese Fälle jetzt mit G für das Generieren, E für das Eliminieren und W für das Weiterreichen eines Übertrags ab. Wir können uns vorstellen, dass wir für alle n Stellen parallel G, E und W korrekt berechnen. Das können wir mit n Halbaddierern erledigen, wie wir uns schon für den Carry-Look-Ahead-Addierer überlegt hatten. Wir wollen diese Informationen jetzt von rechts nach links durchreichen, so dass wir schließlich wissen, an welchen Stellen ein Übertrag zusätzlich zu den beiden Bits zu addieren ist. Wir können uns dazu eine Verknüpfung $\circ$ jeweils zwischen den Stellen vorstellen, wobei $G \circ G = G \circ E = G \circ W = G$ gilt, denn unabhängig davon, was rechts von einer Stelle passiert, gibt es im Falle „G“ einen Übertrag. Analog gilt $E \circ G = E \circ E = E \circ W = E$. Etwas anders sieht es für W aus. Es gilt offenbar $W \circ E = E$ und $W \circ G = G$, da ja W einen Übertrag weiterreicht, wenn er vorhanden ist. Darum gilt $W \circ W = W$, das Weiterreichen setzt sich ja über beliebig viele Stellen fort. Weil wir G, E und W mit Halbaddierern berechnen, codieren wir G durch $(1,0)$, E durch $(0,0)$ und W durch $(0,1)$. Jetzt brauchen wir einen „Baustein“ (also ein möglichst kleines und flaches Schaltnetz), der uns diese Verknüpfung $\circ$ berechnet. Der Baustein hat vier Eingänge i_1, i_2, i'_1, i'_2, wobei (i_1, i_2) und (i'_1, i'_2) jeweils G, E oder W codieren.

Entsprechend gibt es zwei Ausgänge o_1, o_2, die auch wieder G, E oder W codieren. Entsprechend der von uns gewählten Codierung wollen wir also Funktionen o_1 und o_2 gemäß der Wertetabelle 9 realisieren.

i_1	i_2	i'_1	i'_2	o_1	o_2
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	0	1	0	1
0	1	1	0	1	0
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0

Tabelle 9: Wertetabelle für die Berechnung von $\circ$

Wir können direkt ablesen, dass wir $o_1 = i_1 \vee i_2\, i'_1$ und $o_2 = i_2\, i'_2$ realisieren wollen und sehen, dass ein Schaltnetz der Größe 3 und Tiefe 2 ausreicht. Sei $v_i \in \{E, G, W\}$ das Symbol für die i-te Stelle. Sei $t_i := v_i \circ v_{i-1} \circ \cdots v_0$ das uns interessierende Resultat für die i-te Stelle. Wenn wir t_0, t_1, …, t_{n-1} berechnet haben, dann könnten wir direkt die Überträge ablesen: $t_i = G$ symbolisiert einen Übertrag an der i-ten Stelle, $t_i \in \{E, W\}$ symbolisiert keinen Übertrag, da $t_i = W$ nur gelten kann, wenn $v_0 = v_1 = \cdots = v_i = W$ gilt und von ganz rechts dann eben kein Übertrag weitergereicht wird. Weil wir G mit $(1, 0)$ codieren und E und W mit $(0, 0)$ bzw. $(0, 1)$, können wir an der vorderen Stelle den Übertrag direkt ablesen.
Wir haben jetzt einen dreistufigen Plan für die Addition. Auf der ersten Stufe berechnen wir mit n Halbaddierern parallel in Tiefe 1 und Größe $2n$ alle v_i. Auf der zweiten Stufe berechnen wir alle t_i. Wie wir das konkret machen und welchen Aufwand das erfordert, überlegen wir uns gleich. Auf der dritten Stufe berechnen wir die korrekte Summe, dazu genügen wieder Halbaddierer, in die wir die auf der ersten Stufe berechneten Summenbits und die aus t_i ablesbaren Überträge eingeben. Abgesehen von der zweiten Stufe haben wir also Tiefe 2 und Größe $4n$.
Wir werden uns jetzt drei verschiedene Algorithmen für die zweite Stufe, also für die Berechnung der t_i, überlegen. Eingabe ist dabei jeweils $v_0, v_1, \ldots, v_{n-1}$, Ausgabe jeweils $t_0, t_1, \ldots, t_{n-1}$. Die Algorithmen werden unterschiedliche Ansätze verfolgen und unterschiedliche Größe und Tiefe haben. Die dabei gewonnen Erkenntnisse werden uns schließlich zu einer sehr befriedigenden Lösung unseres Problem helfen.

Den ersten Algorithmus nennen wir Algorithmus A. Er lässt sich leicht in drei Schritten beschreiben. Wir gehen ab jetzt davon aus, dass n eine Zweierpotenz ist, also $n = 2^k$ für eine natürliche Zahl k gilt. Das hat den Vorteil, dass wir n immer halbieren können, bis wir schließlich bei Größe 1 ankommen. Sollte n ursprünglich keine Zweierpotenz sein, so können wir uns die Zahl mit führenden Nullen aufgefüllt vorstellen. Das kann ihre Länge nicht ganz verdoppeln, was nicht ganz so schlimm ist.

1. Berechne parallel die Paare $v_{n-1} \circ v_{n-2}$, $v_{n-3} \circ v_{n-4}$, $\ldots$, $v_1 \circ v_0$.
2. Wende Algorithmus A rekursiv für die Paare an.
3. Berechne parallel für alle geraden i $t_i = v_i \circ t_{i-1}$.

Algorithmus A ist vielleicht etwas überraschend. Die dritte Zeile setzt offenbar voraus, dass alle t_i für ungerade i nach Beendigung der Rekursion schon berechnet sind. Das ist aber auch tatsächlich der Fall: Für $n = 2$ funktioniert der Algorithmus offensichtlich korrekt, weil direkt am Anfang $t_1 = v_1 \circ v_0$ berechnet wird. Nehmen wir an, dass der Algorithmus für $n/2$ korrekt arbeitet. Dann liefert die Rekursion im zweiten Schritt als Ergebnis $(v_1 \circ v_0)$, $(v_3 \circ v_2) \circ (v_1 \circ v_0)$, $\ldots$, $(v_{n-1} \circ v_{n-2}) \circ (v_{n-3} \circ v_{n-4}) \circ \cdots \circ (v_1 \circ v_0)$, wobei die Klammerung den in der ersten Zeile erzeugten Paare entspricht. Die Korrektheit folgt also induktiv.

Kommen wir jetzt zur Analyse von Größe und Tiefe. Wir zählen dabei einen $\circ$-Baustein mit Größe 1 und Tiefe 1, für das Endergebnis müssen wir also die Größe mit 3 und die Tiefe mit 2 multiplizieren. Unsere Analyse verläuft Algorithmus A entsprechend rekursiv. Es bezeichne $S_A(n)$ die Größe bei Eingabegröße n, $D_A(n)$ entsprechend die Tiefe. Vereinbarungsgemäß gilt also $S_A(2) = D_A(2) = 1$. Wir können der Algorithmenbeschreibung direkt entnehmen, dass $S_A(n) = (n/2) + S_A(n/2) + (n/2) - 1 = S_A(n/2) + n - 1$ gilt. Ebenso können wir $D_A(n) = 1 + D_A(n/2) + 1 = D_A(n/2) + 2$ direkt ablesen. Wir behaupten, dass $S_A(n) = 2n - \log_2(n) - 2$ gilt und führen den Beweis per vollständiger Induktion. Wir haben $2 \cdot 2 - \log_2(2) - 2 = 4 - 1 - 2 = 1 = S_A(2)$ und der Induktionsanfang ist gesichert. Jetzt haben wir $S_A(n) = S_A(n/2) + n - 1 = 2(n/2) - \log_2(n/2) - 2 + n - 1 = n - \log_2(n) + 1 - 2 + n - 1 = 2n - \log_2(n) - 2$ wie behauptet.

Für die Tiefe behaupten wir, dass $D_A(n) = 2\log_2(n) - 1$ gilt und führen den Nachweis wiederum mittels vollständiger Induktion. Wir haben mit $2\log_2(2) - 1 = 2 - 1 = 1 = D_A(2)$ den Induktionsanfang. Außerdem gilt $D_A(n) = D_A(n/2) + 2 = 2\log_2(n/2) - 1 + 2 = 2\log_2(n) - 2 - 1 + 2 = 2\log_2(n) - 1$ wie behauptet.

Die Größe $2n - \log_2(n) - 2$ (also eigentlich $6n - 3\log_2(n) - 6$) ist schon sehr zufriedenstellend, für die Tiefe wäre es schön, den Faktor 2 vor $\log_2(n)$ loswerden zu können. Das wird uns mit dem nächsten Algorithmus gelingen, den wir Algorithmus B nennen.

1. Wende Algorithmus B parallel auf $(v_{n-1}, v_{n-2}, \dots v_{n/2})$ und $(v_{(n/2)-1}, v_{(n/2)-2}, \dots, v_0)$ an.

2. Berechne parallel alle t_i mit $i \geq n/2$ aus $t_{(n/2)-1}$ und dem passenden Stück aus dem Ergebnis des Aufrufs für $(v_{n-1}, v_{n-2}, \dots, v_{n/2})$.

Die Korrektheit von Algorithmus B ist offensichtlich, wir wollen auch hier Größe und Tiefe abschätzen und dabei zunächst einen $\circ$-Baustein mit Größe und Tiefe 1 zählen. Wir bezeichnen die Größe diesmal mit $S_B(n)$, die Tiefe mit $D_B(n)$ und haben natürlich $S_B(1) = D_B(1) = 0$, da für $n = 1$ ja nichts ausgerechnet werden muss. Wir lesen direkt $S_B(n) = 2S_B(n/2) + n/2$ und $D_B(n) = D_B(n/2) + 1$ ab.
Für die Tiefe behaupten wir, dass $D_B(n) = \log_2(n)$ gilt, wir also wie gewünscht den Faktor 2 eingespart haben. Das lässt sich auch leicht nachrechnen: Es ist $\log_2(1) = 0 = D_B(1)$ und $D_B(n) = D_B(n/2) + 1 = \log_2(n/2) + 1 = \log_2(n) - 1 + 1 = \log_2(n)$ wie behauptet.
Leider ist die Größe nicht so gut wie für Algorithmus A. Wir behaupten $S_B(n) = (1/2)n\log_2(n)$ und rechnen $(1/2) \cdot 1 \cdot \log_2(n) = 0 = S_B(1)$ sowie $S_B(n) = 2S_B(n/2) + n/2 = 2((1/2)(n/2)\log_2(n/2)) + n/2 = (n/2) \cdot (1 + \log_2(n) - 1) = (1/2)n\log_2(n)$ nach.
Was wir uns eigentlich wünschen ist ein Algorithmus, der die Tiefe von Algorithmus B erreicht (also Tiefe $\log_2(n)$ bzw. für uns schließlich Tiefe $2\log_2(n)$), dabei aber lineare Größe behält, also Größe $\sim n$. Es ist eine erstaunliche Tatsache, dass das gelingt, wenn man die beiden betrachteten Algorithmen geschickt miteinander kombiniert. Wir beschreiben jetzt zwei Algorithmen A_0 und A_1, deren Verwandtschaft mit unseren Algorithmen A und B ganz offensichtlich sein wird.
Algorithmus A_0:

1. Wende parallel Algorithmus A_0 auf $(v_{n-1}, v_{n-2}, \dots, v_{n/2})$ und Algorithmus A_1 auf $(v_{(n/2)-1}, v_{(n/2)-2}, \dots, v_0)$ an.

2. Berechne so früh wie möglich die fehlenden t_i für $i \geq n/2$.

Algorithmus A_1:

1. Berechne parallel die Paare $v_{n-1} \circ v_{n-2}$, $v_{n-3} \circ v_{n-4}$, $\dots$, $v_1 \circ v_0$.

2. Wende Algorithmus A_0 auf diese Paare an.

3. Berechne parallel alle t_i für gerade i.

Die Korrektheit der beiden Algorithmus folgt direkt aus der Korrektheit von Algorithmus A und Algorithmus B. Wir können also sofort zur Analyse kommen. Wir schauen uns zunächst die Tiefe an und bezeichnen mit $D_0(n)$ die Tiefe von A_0, mit $D_1(n)$ die Tiefe von A_1 und schließlich mit $D_1^*(n)$ die Tiefe innerhalb von A_1, bei der t_{n-1} fertig berechnet ist. Diese Tiefe $D_1^*(n)$ ist entscheidend für die Tiefe $D_0(n)$ von A_0.
Natürlich gilt $D_0(1) = D_1(1) = D_1^*(1) = 0$. Außerdem können wir aus Algorithmus A_1 direkt $D_1(n) = D_0(n/2) + 2$ und $D_1^*(n) = D_0(n/2) + 1$ ablesen. Für Algorithmus A_0 ist die Lage etwas verwickelter. Wir führen je einen rekursiven Aufruf von A_0 und A_1 für halb so viel Argumente aus. Der rekursive Aufruf von A_0 liefert $D_0(n) \geq D_0(n/2)$. Es gilt sogar $D_0(n) \geq D_0(n/2) + 1$, weil wir ja nach diesem rekursiven Aufruf noch die fehlenden t_i berechnen müssen, was zusätzlich (unserer Zählweise nach) Tiefe 1 einbringt. Wegen des rekursiven Aufrufs von Algorithmus A_1 gilt auch sicher $D_0(n) \geq D_1(n/2)$. Wir können hier aber nicht noch 1 hinzuzählen: die Berechnung der fehlenden t_i kann schon nach Tiefe $D_1^*(n/2)$ einsetzen. Wir haben also insgesamt $D_0(n) = \max\{D_0(n/2) + 1, D_1(n/2), D_1^*(n/2) + 1)\}$. Wir behaupten, das $D_0(n) = \log_2(n)$, $D_1(n) = \log_2(n) + 1$ und $D_1^*(n) = \log_2(n)$ gilt. Die Richtigkeit der Behauptung lässt sich wieder leicht nachrechnen.
Für die Größe haben wir zunächst offensichtlich $S_0(1) = S_1(1) = 0$ und $S_0(2) = S_1(2) = 1$, außerdem können wir $S_0(n) = S_0(n/2) + S_1(n/2) + n/2$ und $S_1(n) = S_0(n/2) + n - 1$ aus den Algorithmen direkt ablesen. Wir sparen uns hier die etwas verwickelte Analyse und geben uns mit der Einsicht zufrieden, dass $S_0(n) \leq 4n$ gilt, was sich wiederum leicht durch Induktion nachweisen lässt.
Wir haben jetzt also insgesamt für die zweite Stufe unseres Addierers Tiefe $2\log_2(n)$ bei Größe $\leq 12n$, so dass wir nun insgesamt ein Schaltnetz für die Addition zweier n-Bit-Zahlen angeben können, das Tiefe $2\log_2(n) + 2$ bei Größe $\leq 16n$ hat. Wir sind damit nur etwas mehr als drei Mal so groß wie der (quälend langsam rechnende) Schuladdierer. Für die oben betrachtete Addition von 128-Bit-Zahlen hätten wir die Tiefe von 255 auf 16 gesenkt, dabei die Größe nur recht moderat von 637 auf 2 048 erhöht.
Dieser Addierer ist als *Ladner/Fischer-Addierer* bekannt, er beruht wesentlich auf der effizienten Berechnung der Verknüpfung $\circ$. Diese Berechnung, die keine besonderen Eigenschaften von $\circ$ ausnutzt und darum für jede beliebige assoziative Verknüpfung $\otimes$ funktioniert, ist 1980 von Richard E. Ladner und Michael J. Fischer vorgestellt worden (Ladner, Fischer: Parallel Prefix Computation, Journal of the ACM 27(4):831–838, 1980).

Schaltnetze für die Multiplikation

Kommen wir jetzt noch einmal auf die Multiplikation zweier Betragszahlen zurück. Wir hatten uns überlegt, dass bei Vorgehen nach der Schulmethode die eigentlichen Multiplikationen ganz einfach zu realisieren sind und als wesentliches Problem die Addition von vielen Binärzahlen übrig bleibt. Wenn die beiden Faktoren jeweils die Länge n haben, so müssen wir n Zahlen $m_1, m_2, \ldots, m_n$ der Länge $2n$ addieren. Wir könnten das mit $n-1$ der eben diskutierten Addierer erledigen, indem wir jeweils nacheinander $m_1 + m_2$, dann $\underbrace{(m_1+m_2)}_{\text{schon berechnet}} + m_3$ usw. berechnen. Damit kommen wir auf eine Gesamttiefe von etwa $2(n-1)\log n$, was ganz klar viel zu tief ist.
Dass so ein naives Vorgehen nicht gut sein kann, war uns natürlich schon vorher klar: Es ist in Schaltnetzen niemals eine gute Idee, alles hintereinander machen zu wollen. Und es liegt auch nahe, wie man die Addition der Zahlen $m_1, \ldots, m_n$ wesentlich beschleunigen kann: Wir können die Zahlen paarweise gleichzeitig addieren, also $m_1 + m_2$, $m_3 + m_4$ usw. gleichzeitig berechnen. Danach können wir die Summen $\underbrace{(m_1+m_2)}_{\text{schon berechnet}} + \underbrace{(m_3+m_4)}_{\text{schon berechnet}}$, $\underbrace{(m_5+m_6)}_{\text{schon berechnet}} + \underbrace{(m_7+m_8)}_{\text{schon berechnet}}$ usw. berechnen, danach noch weiter zu

$$\underbrace{(m_1+m_2+m_3+m_4)}_{\text{schon berechnet}} + \underbrace{(m_5+m_6+m_7+m_8)}_{\text{schon berechnet}}$$

zusammenfassen, bis wir schließlich die Gesamtsumme $m_1 + \cdots + m_n$ berechnet haben. Wie viele von unseren Addierern brauchen wir dazu? Auf der ersten Ebene führen wir $n/2$ Additionen durch und erhalten dann natürlich $n/2$ Summen. Damit haben wir auf der zweiten Ebene diese als $n/2$ Summanden, die wir in $n/4$ Additionen zu $n/4$ Summanden der dritten Ebene führen. Das setzt sich fort, bis wir auf der untersten Ebene nur noch zwei Summanden haben. Wenn n eine Zweierpotenz ist, funktioniert das tatsächlich genau so, ansonsten fehlen einem einige Additionen auf manchen Ebenen. So genau wollen wir hier jetzt aber nicht zählen. Wie viele Ebenen sind es denn jetzt? Wir haben auf der i-ten Ebene $n/2^i$ Additionen und suchen die Ebene i, auf der es nur noch eine Addition ist. Wir brauchen also nur $n/2^i = 1$ nach i auflösen und sehen, dass wir $\log_2 n$ Ebenen haben. Jetzt können wir auch einfach die Anzahl der Addierer ausrechnen, die wir dabei benutzen. Auf der ersten Ebene starten wir mit $n/2$ Addierern, dann halbiert sich diese Anzahl von Ebene zu Ebene. Wir kommen also mit

$$\sum_{i=1}^{\log_2 n} \frac{n}{2^i} = \frac{n}{2^{\log_2 n}} \cdot \sum_{i=1}^{\log_2 n} 2^{(\log_2 n)-i} = \sum_{i=0}^{(\log_2 n)-1} 2^i = 2^{\log_2 n} - 1 = n-1$$

Addierern aus, brauchen aber dabei nur Gesamttiefe etwa $2(\log_2 n)^2$. Wir können also ganz erheblich an Tiefe sparen (und somit schneller werden) und trotzdem in der Größe nicht wesentlich zunehmen.
Es wird vermutlich niemanden überraschen, dass es noch einmal erheblich besser geht. Überraschend ist aber vielleicht, dass das mit Mitteln geht, die wir hier schon entwickelt haben – man braucht nur einen kleinen, cleveren Einfall dazu. Wir haben ja gerade jeweils aus zwei Zahlen eine Zahl gemacht (durch Addition) und so die Anzahl der Zahlen um 1 reduziert. Für diese Addition braucht man natürlich einen Addierer, der ja selbst schon relativ komplex ist und Tiefe etwa $2 \log_2 n$ hat. Wir machen noch ganz ausdrücklich eine trivial klingende Beobachtung: Die neue Zahl, die wir errechnen, hat natürlich den gleichen Betrag wie die beiden ersetzen Zahlen, so dass sich auf jeder Ebene die Summe aller Zahlen nicht ändert. Diese Eigenschaft garantiert ja gerade die Korrektheit unseres Entwurfs. Wir können jetzt diese beiden Ideen (also erstens die Anzahl der Zahlen schrittweise zu reduzieren und zweitens die Summe der beteiligten Zahlen dabei nicht zu verändern) aufgreifen und innovativ weiterentwickeln. Wir könnten ja auch jeweils drei Zahlen nehmen und daraus zwei neue Zahlen berechnen, die dabei die gleiche Summe haben. Wir wollen zunächst einmal annehmen, dass wir einen Baustein haben, der eine solche Funktion realisiert: Er bekommt als Eingabe drei Zahlen der Länge $2n$ und produziert als Ausgabe zwei Zahlen der Länge $2n$ mit gleicher Summe[5]. Man nennt einen solchen Baustein CSA (kurz für Carry Save Adder). Wir fangen auf der ersten Ebene mit n Zahlen an, benutzen dann $\lfloor n/3 \rfloor$ Carry Save Adder und erhalten damit auf der Ebene darunter höchstens $2 \lfloor n/3 \rfloor + 2$ Zahlen; wenn n nicht durch drei teilbar ist, können bis zu zwei Zahlen übrig bleiben. Wir überschätzen die Anzahl von CSA-Bausteinen und Zahlen auf der nächsten Ebene, wenn wir stattdessen mit $\lceil n/3 \rceil$ CSA-Bausteinen und mit $2 \lceil n/3 \rceil$ Zahlen rechnen. Auf der Ebene darunter können wir dann wieder etwa ein Drittel der Zahlen zusammenfassen, es bleiben bis zu zwei Zahlen übrig. Die Aufteilung der Zahlen auf diese Weise kann man gut grafisch darstellen (siehe Abbildung 14, dabei symbolisiert jetzt jede Kante eine *Zahl*). Man nennt die entstehende Anordnung (manchmal) Addierbaum oder Wallace-Tree (vorgeschlagen von C. S. Wallace 1960). Wir können also die Anzahl der CSA-Bausteine auf der i-ten Ebene durch

$$\left(\frac{2}{3}\right)^i \cdot n + \sum_{j=0}^{i-1} \frac{2^{j+1}}{i} < \left(\frac{2}{3}\right)^i \cdot n + 6$$

[5]Das soll nur funktionieren, wenn es keinen Übertrag gibt, der über $2n$ Stellen hinausgeht. Mit Bereichsüberschreitungen wollen wir uns hier nicht befassen. Man sieht leicht ein, dass sie wegen der Beschaffenheit der Zahlen (Woher kommen sie gleich nochmal?) auch nicht oft auftreten können.

nach oben abschätzen. Wir sehen, dass diesmal mit höchstens $(\log_{3/2} n) + 12$ Ebenen auskommen. Bis jetzt scheinen wir noch nichts gewonnen zu haben. Aber bis jetzt haben wir ja auch noch nicht in die CSA-Bausteine hinein gesehen. Unsere Hoffnung ist, dass man einen CSA-Baustein wesentlich besser realisieren kann als einen Addierer.

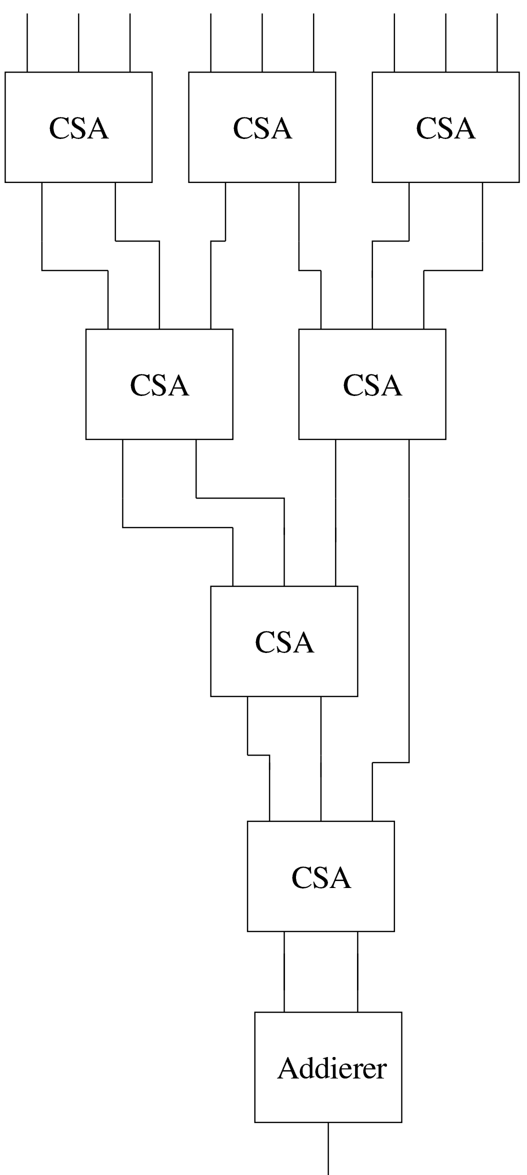


Abbildung 14: Wallace-Tree (Addierbaum)

Denken wir zunächst über Zahlen der Länge 1 nach, also über einzelne Bits. Wir haben drei Bits x, y, z und wollen diese so addieren, dass wir zwei neue

Zahlen bekommen, die in der Summe $x+y+z$ ergeben. Um ein Problem mit dem Übertrag zu vermeiden, sollten wir aus unseren Bits gedanklich 2-Bit-Zahlen machen, die jeweils eine führende Null haben. Damit erlauben wir uns dann auch zwei 2-Bit-Zahlen als Ergebnis. Über die Addition von drei Bits haben wir schon intensiv nachgedacht – dabei sind wir zum Volladdierer gekommen, der das Ergebnis als Summen- und Übertragsbit bereitstellt. Wenn wir uns vor das Summenbit eine Null und hinter das Carrybit eine Null denken, dann haben wir unsere zwei Zahlen gefunden. Wir können also einen CSA-Baustein alleine durch Volladdierer realisieren, wenn wir darauf achten, die Bits an die passenden Stellen zu schreiben und gegebenenfalls durch Nullen zu ergänzen. Wir können also einen CSA-Baustein mit $3l$ Eingängen und $2l$ Ausgängen mit l Volladdierern realisieren, die alle parallel arbeiten können. Wir kommen also mit Tiefe 3 und Größe $15l$ aus. Damit ergibt sich jetzt eine Gesamttiefe von etwa $3(\log_{3/2} n) + 39 + 2\log_2 n$.

Wir haben ziemlich grob gerechnet. Trotzdem ist deutlich geworden, dass wir durch Anwendung von CSA-Bausteinen und deren Anordnung in einem Wallace-Tree in der Tiefe wesentlich gewonnen haben, ohne in der Größe zu verlieren. Den für große n wesentlichen Term konnten wir von $(\log n)^2$ auf $\log n$ reduzieren.

Subtraktion ganzer Zahlen

Für die Subtraktion „$a - b$" genügt es, von b zu $-b$ überzugehen und dann zu addieren. Für drei von unseren Darstellungen ist das ganz einfach: bei der Vorzeichenbetragsmethode genügt es, das Vorzeichenbit zu invertieren. Bei der Einerkomplementdarstellung sind alle Bits zu invertieren, bei der Zweierkomplementdarstellung ist nach der Invertierung noch 1 zu addieren. All das ist kein großes Problem. Schwieriger wird es mit der Exzessdarstellung; bei dieser Darstellung funktioniert nicht einmal bei positiven Zahlen die Addition nach dem oben vorgestellten Muster. Nehmen wir zwei positive Zahlen x und y, die ja als $x+b$ und $y+b$ dargestellt werden. Addiert man einfach, so erhält man $x+y+2b$, was $x+y+b$ darstellt und nicht wie gewünscht $x+y$. Man muss also noch die Verschiebung b abziehen, was je nach Wert von b einfach oder schwierig sein kann. Wir kommen gleich wieder darauf zurück, wenn wir uns bei Gleitkommazahlen mit einem konkreten Bias b beschäftigen können.

Worüber wir noch gar nicht nachgedacht haben, sind negative Zahlen. Die oben diskutierten Addierer sind ausschließlich für nicht-negative Zahlen gedacht. Natürlich können wir analog auch Subtrahierer entwerfen. Aber ist das eigentlich nötig? Bei der Vorzeichenbetragsmethode ist das wohl so. Aber bei

der Einer- und Zweierkomplementdarstellung lohnt es sich, einmal genauer hinzusehen.

Wir fangen mit der Einerkomplementdarstellung an. Es sei a eine Zahl, ihr bitweises Komplement sei $\overline{a}$. In der Einerkomplementdarstellung stellt $\overline{a}$ die Zahl $-a$ dar. Unsere Addierer interpretieren die Zahlen als nicht-negative Zahlen. Wenn man das tut, dann gilt $a + \overline{a} = 2^l - 1$. Wenn wir also $b - a$ berechnen wollen, so geben wir $b + \overline{a}$ in den Addierer, was dieser als $b + 2^l - 1 - a$ interpretiert. Gut ist, dass 2^l in unseren l Bits einfach als Übertrag verschwindet, so dass wir tatsächlich $b-a-1$ erhalten. Wenn wir anschließend noch 1 addieren, hat der Addierer das korrekte Ergebnis der Subtraktion berechnet. Wir können also in der Einerkomplementdarstellung mit Hilfe eines Addierers subtrahieren, wir müssen nur zur Korrektur dem Ergebnis 1 hinzuzählen – also eine zweite Addition durchführen.

Jetzt sehen wir uns noch die Zweierkomplementdarstellung an. Es gilt natürlich weiterhin $a + \overline{a} = 2^l - 1$, wenn man nur nicht-negative Zahlen aus den Repräsentationen heraus liest. In der Zweierkomplementdarstellung stellen wir $-a$ als $\overline{a} + 1$ dar. Wenn wir jetzt $b - a$ berechnen wollen, geben wir also $b+\overline{a}+1$ in den Addierer, was dieser als $b+2^l-1-a+1 = 2^l+(b-a)$ interpretiert. Der in l Bits nicht darstellbare Überlauf 2^l verschwindet, so dass direkt korrekt $b - a$ berechnet wird. Wir brauchen also in der Zweierkomplementdarstellung gar nichts weiter unternehmen, um zu subtrahieren. Das macht der Addierer einfach mit. Insofern ist die Zweierkomplementdarstellung noch praktischer als die Einerkomplementdarstellung. Das dürfte der Grund dafür sein, dass in der Praxis die Zweierkomplementdarstellung die am häufigsten benutzte ist. Dafür ist ein Vorzeichenwechsel etwas komplizierter als in der Einerkomplementdarstellung.

Machen wir uns noch einmal klar, woran man bei der Addition zweier Zahlen, die in Zweierkomplementdarstellung gegeben sind, erkennen kann, ob das Ergebnis in der gegebenen Anzahl von Bits dargestellt werden kann. Es ist zweckmäßig, dabei drei Fälle zu unterscheiden.

1. Wenn zwei positive Zahlen addiert werden, ist das Ergebnis sicher positiv. Wenn die Darstellung in der gegebenen Länge nicht möglich ist, entsteht ein Übertragsbit, das an der ganz links gelegenen Stelle (dem MSB) entsteht. Das Ergebnis der Addition von zwei positiven Zahlen ist also immer dann gültig, wenn das MSB den Wert 0 hat.

2. Wenn eine positive und eine negative Zahl addiert werden, ist das Ergebnis immer darstellbar, da es größer als die negative und kleiner als die positive Zahl ist, die ja beide darstellbar sind.

3. Wenn zwei negative Zahlen addiert werden, entsteht in jedem Fall ein Überlauf, der aber ignoriert werden kann. Das hatten wir uns oben überlegt. Woran erkennen wir dann, ob das Ergebnis korrekt darstellbar ist bei der gegebenen Anzahl von Stellen? Wir erinnern uns daran, dass $-a$ als $\overline{a} + 1$ dargestellt wird. Interpretiert man die Darstellung als einfache Binärdarstellung, so wird $-a$ als $2^l - 1 - a + 1 = 2^l - a$ dargestellt. Entsprechend wird $-b$ als $2^l - b$ dargestellt. Wir wissen, dass in Zweierkomplementdarstellung mit Länge l die kleinste darstellbare Zahl die -2^{l-1} ist. Wenn $a + b > 2^{l-1}$ gilt, dann folgt daraus, dass $2^l - (a + b) < 2^{l-1}$ gilt. Na und? Nun, wir können die Berechnung von $(-a) + (-b)$ interpretieren als $2^l - a + 2^l - b = 2^l + 2^l - (a + b)$. In l Bits erscheint 2^l gerade nicht, wir können also den ersten Summanden ignorieren – es handelt sich um den schon erwähnten Überlauf. Für den Rest haben wir, dass das Ergebnis genau dann nicht in l Bits dargestellt werden kann, wenn $2^l - (a + b) < 2^{l-1}$ ist. In diesem Fall hat das MSB den Wert 0. Das Ergebnis der Addition von zwei negativen Zahlen ist also immer dann gültig, wenn das MSB den Wert 1 hat.

Gleitkomma-Arithmetik

Schon etwas anders funktioniert das Rechnen mit Gleitkommazahlen. Wir haben zwei Zahlen x und y, die als $(-1)^{s_x} \cdot m_x \cdot 2^{e_x}$ und $(-1)^{s_y} \cdot m_y \cdot 2^{e_y}$ gegeben sind. Dabei sind s_x, s_y die Vorzeichenbits, m_x, m_y sind die Mantissen (zu denen wir uns die implizite 1 schon hinzu addiert denken) und schließlich e_x, e_y die Exponenten, die in Exzessdarstellung mit Bias $b = 2^{l-1} - 1$ vorliegen. In gleicher Darstellung soll am Ende das Ergebnis $z = (-1)^s \cdot m \cdot 2^e$ vorliegen. Bei Rechnungen mit Gleitkommazahlen ist der trickreiche (und vielleicht dadurch ja gerade interessante) Teil das Runden. Wir wollen hier aber nicht so tief in die Materie einsteigen und uns damit zufrieden geben, die Prinzipien zu verstehen.

Multiplikation von Gleitkommazahlen: Wir beginnen mit der Multiplikation, die strukturell am einfachsten ist – vorausgesetzt, man hat sich klar gemacht, wie man ganze Zahlen multipliziert und addiert. Das neue Vorzeichen ergibt sich einfach als $s_x \oplus s_y$, die beiden Mantissen müssen multipliziert werden, die Exponenten addiert. Wir haben also $x \cdot y = z$ mit $s = s_x \oplus s_y$, $m = m_x \cdot m_y$ und $e = e_x + e_y$. Bei allen Rechnungen mit den Mantissen muss natürlich immer beachtet werden, dass die Eins vor dem Komma zwar nicht mit abgespeichert ist, sie aber implizit vorhanden ist und darum bei den Rechnungen berücksichtigt werden muss. Wir berechnen also eigentlich $(1 + m_x) \cdot (1 + m_y)$ und müssen zur Abspeicherung der Ergebnismantisse

wieder normalisieren und dabei die führende Eins fallenlassen. Für den Exponenten haben wir uns oben schon überlegt, dass wir dazu mit „normalen" Rechenschaltkreisen $e_x + e_y - b$ berechnen müssen, dabei ist b der feste Bias.

Addition von Gleitkommazahlen: Die Addition zweier Gleitkommazahlen ist schon wesentlich komplizierter. Die Feinheiten entstehen auch hier wieder bei der Rundung; wir gehen aus Zeitgründen wiederum nicht darauf ein. Nehmen wir also an, dass wir x und y addieren wollen. Wir nehmen an, dass der Exponent von x größer ist, andernfalls vertauschen wir die Rollen von x und y. Hier sehen wir einen Grund dafür, für die Exponenten die Exzessdarstellung zu verwenden: Bei der Exzessdarstellung bleibt die Reihenfolge der Zahlen erhalten, der Größenvergleich ist also einfacher als in den anderen Darstellungen.
Im zweiten Schritt prüfen wir, ob die Vorzeichen von x und y gleich sind. Wenn das nicht der Fall ist, muss ja eine Subtraktion durchgeführt werden. Dazu gehen wir von m_y zum Zweierkomplement von m_y über, also zu $\overline{m_y}+1$. Das erlaubt es, später für die Addition der Mantissen auf jeden Fall einen normalen Addierer zu benutzen. Jetzt werden wir m_y um $e_x - e_y$ nach rechts verschieben – man spricht auch von *Exponentenanpassung*. Dadurch haben wir jetzt (gedanklich) die beiden Exponenten gleich gemacht. Also können wir im nächsten Schritt die beiden Mantissen addieren. Es ist klar, dass dabei nicht auf alle Stellen von m_y (die ja vermutlich zum Teil nach rechts heraus geschoben worden sind) Rücksicht genommen werden kann – ein Problem, auf das wir weiter unten noch etwas näher eingehen werden. Falls $e_x = e_y$ war (also x und y in der gleichen Größenordnung liegen), kann bei der Addition eine negative Zahl herauskommen – natürlich in Zweierkomplementdarstellung. Falls das so ist, müssen wir das Vorzeichen ändern und wieder zurück in die Betragszahldarstellung konvertieren. Jetzt müssen wir das Ergebnis wieder normalisieren, also die vorläufige Mantisse verschieben und den Exponenten des Ergebnisses (den wir ja vorläufig als e_x angenommen hatten) entsprechend anpassen. Danach werden die erforderlichen Rundungen der Mantisse durchgeführt. Schließlich muss man noch das Vorzeichen des Ergebnisses richtig setzen. Wenn x und y gleiches Vorzeichen hatten, so ist das natürlich auch das Vorzeichen des Ergebnisses. Andernfalls kommt es darauf an, welche der beiden Zahlen das positive Vorzeichen hatte und ob wir nach der Addition der Mantisse einen Übergang vom Zweierkomplement hatten.

Probleme bei der Addition: Ein Szenario Wie schon weiter oben angedeutet, ist die Gleitkommaaddition die „problematischere" der beiden hier betrachteten Gleitkommagrundrechenarten, da bei der Exponentenan-

passung unter Umständen signifikante Bits der Mantisse der kleineren Zahl verloren gehen. Auch wenn die Auswirkungen dieses Effekts durch Rundung und die Verwendung sogenannter Hilfsbits etwas abgemildert werden können, so birgt die Addition von Gleitkommazahlen doch generell das Risiko, dass ein verfälschtes Ergebnis entsteht[6]. Der Effekt kann dann besonders stark ausfallen, wenn nicht nur ein Paar sondern eine größere Anzahl von Zahlen addiert (bzw. subtrahiert) wird.
Betrachten wir also eine Folge von n Gleitkommazahlen $[x_i]$ mit $0 \leq i \leq n$. Wir stellen uns nun die Aufgabe, die Summe S aller x_i mit den Möglichkeiten bzw. unter den Einschränkungen der Gleitkommaarithmetik möglichst exakt zu berechnen. Die naive und intuitiv auch unmittelbar naheliegende Lösung besteht darin, die Summation direkt in ein entsprechendes Programm umzusetzen, z.B. indem die Elemente der Folge $[x_i]$ in aufsteigender

$$S\uparrow = \sum_{i=0}^{n-1} x_i$$

oder absteigender Reihenfolge

$$S\downarrow = \sum_{i=n-1}^{0} x_i$$

aufsummiert werden (bzw. gemäß jeder anderen Sortierung der Elemente x_i). Aus theoretischer Sicht (Kommutativität und Assoziativität der Addition) – und auch intuitiv betrachtet – sollten beide Methoden dasselbe Ergebnis liefern. In der Praxis – d.h. bei der Verwendung von Gleitkommaarithmetik – werden jedoch $S\uparrow$ und $S\downarrow$ i.a. *nicht gleich* sein. Dieser Effekt läßt sich relativ leicht auf einem beliebigen Rechner mit Hilfe eines einfachen Programms überprüfen, das eine längere Folge von Zufallszahlen nach beiden Varianten addiert und die Ergebnisse vergleicht.
Um diesem unangenehmen Problem zu begegen, bieten sich prinzipiell zwei mögliche Verfahrensklassen an. Zum einen könnte man die Genauigkeit der Gleitkommazahlenrepräsentation durch die Verwendung einer größeren Anzahl von Bits zur Darstellung der Mantisse erhöhen. Zwar wird man dann in bestimmten konkreten Berechnungsszenarien geringere Fehler beobachten, das prinzipielle Problem besteht allerdings weiterhin, da es in der Natur der Gleitkommazahlenrepräsentation begründet liegt. Daher ist der einzig aussichtsreiche Weg zur Verbesserung der Rechengenauigkeit die Modifikation

[6] Im Extremfall hat die Addition einer kleineren zu einer größeren Gleitkommazahl sogar gar keine Auswirkung auf das Ergebnis, d.h. mit $x \gg y$ und $|y| > 0$ beobachtet man $x + y = x$. Dieser spezielle Effekt wird auch als *Absorption* bezeichnet.

```
S = 0;                  /* Summe */
E = 0;                  /* geschaetzter Fehler */
for i = 0 to n-1 {
        Y = x[i] - E;   /* bish. Fehler beruecksichtigen */
        Z = S + Y;      /* neues Summationsergebnis */
        E = (Z - S) - Y; /* neue Fehlerschaetzung */
        S = Z;
        }
```

Abbildung 15: Algorithmus (Kahan-Summation) zur numerisch stabilen Berechnung einer Summe von Gleitkommazahlen $S = \sum_{i=0}^{n-1} x_i$

des Berechnungsschemas selbst, d.h. die Anwendung einer „schlaueren” Methode zur Addition einer Zahlenfolge.

Die Kahan-Summation: Um numerisch genauere Ergebnisse für die Addition einer größeren Anzahl von Gleitkommazahlen zu erzeugen, existieren verschiedene Möglichkeiten, die vielfach voraussetzen, dass die Anzahl der zu addierenden Zahlen vor Berechnungsbeginn bekannt sein muss oder die Zahlen selbst in sortierter Reihenfolge vorliegen müssen. Eine besonders einfach umzusetzende Methode stellt die sogenannte *Kahan-Summation* dar. Die Grundidee liegt dabei darin, neben dem Ergebnis der Addition auch eine Schätzung des entstandenen Fehlers zu berechnen und diesen bei weiteren Additionen zu berücksichtigen. Vereinfacht ausgedrückt „merkt” man sich den Fehler unabhängig von der Bildung der Summe, bis er numerisch „wichtig genug” ist, um in das Ergebnis der Summation einzugehen. Ein entsprechender Algorithmus ist in Abbildung 15 gezeigt.

Ausblick: Wir wollen unseren Ausflug in die Rechnerarithmetik hier beenden. Wir haben eine ganze Reihe von Themen noch gar nicht angesprochen: bei ganzen Zahlen den Entwurf wirklich effizienter Addierer und Multiplizierer, bei Gleitkommazahlen das Runden und den Entwurf effizienter Addierer und Multiplizierer. Außerdem haben wir die Division überhaupt nicht thematisiert.

4.5 Optimierung von Schaltnetzen

Optimierung ist ein immer beliebtes Schlagwort: nicht nur verbessern, sondern gleich eine best mögliche Lösung finden. Wir bedienen uns auch dieser Überschrift und werden tatsächlich auch ein wenig optimieren im engeren

Sinn. Es geht uns aber vor allem schlicht darum, zu einem besseren Entwurf von Schaltnetzen zu kommen, ohne dass wir damit gleich den Anspruch verbinden, „optimale“ Schaltnetze entworfen zu haben. Wir hatten uns ja schon eingangs überlegt, dass es da viele verschiedene und zum Teil widersprechende Optimierungsziele gibt.
Wir wollen damit anfangen, dass wir uns überlegen, wie wir vom Ad-hoc-Entwurf von Schaltnetzen, wie wir ihn in Abschnitt 4.3 praktiziert haben, wegkommen und zu einem systematischeren Entwurf kommen. Die Grundidee ist, einfachere Schaltnetze, die wir entworfen haben, als hilfreiche Bausteine zu benutzen, ganz so, wie wir auf unterster Ebene die einfachen Gatter verwenden.
Als ein Beispiel für einen solchen Entwurf kann man den Addierer nach Schulmethode (Abbildung 12, Seite 53) nennen, bei dem wir auf Halbaddierern und Volladdierern aufgesetzt haben. Aber das geht auch schon eine Stufe früher. Wir haben den Halbaddierer elementar aus nur zwei Gattern mit Tiefe 1 gebaut (Abbildung 10, Seite 51). Für den Volladdierer sind wir wieder ganz von vorne angefangen. Hätte man nicht den Halbaddierer als Baustein verwenden können? Wir sehen uns dazu eine geeignete Wertetabelle an (Tabelle 10).

			„$x+y$“		„A_s+c'“		
c'	x	y	A_c	A_s	B_c	B_s	$A_c \vee B_c$
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	0	1	0	1	0
0	1	1	1	0	0	0	1
1	0	0	0	0	0	1	0
1	0	1	0	1	1	0	1
1	1	0	0	1	1	0	1
1	1	1	1	0	0	1	1

Tabelle 10: Wertetabelle zum strukturierten VA-Entwurf

In der Spalte unter „$x+y$“ sehen wir uns das Ergebnis der Addition von x und y, das Summen- und Übertragsbit kann natürlich von einem Halbaddierer bereitgestellt werden. Das Summenbit und das bisher nicht berücksichtigte „alte“ Übertragsbit c' benutzen wir als Eingabe für einen zweiten Halbaddierer, dessen Funktionswerte wir uns in der Spalte unter „$A_s + c'$“ ansehen. Wir erkennen direkt, dass das Summenbit dieses Halbaddierers dem Summenbit der Gesamtsumme entspricht. Jetzt fehlt nur noch das Übertragsbit der Gesamtsumme. Es ist unmittelbar klar, dass es eines gibt, wenn mindestens einer der beiden Halbaddierer eines berechnet hat. Folglich erhalten

wir das Gesamtübertragsbit als Disjunktion von A_c und B_c, dargestellt in der letzten Spalte. Damit erhalten wir insgesamt folgendes strukturiert entworfenes Schaltnetz für einen Volladdierer (Abbildung 16), das zwei Halbaddierer und ein Oder-Gatter verwendet, insgesamt also mit Größe 5 und Tiefe 3 auskommt. Wir unterscheiden uns damit nicht von unserem ersten Entwurf eines Halbaddierers; das Schaltnetz ist aber besser strukturiert und in seiner Funktionsweise leichter nachzuvollziehen.

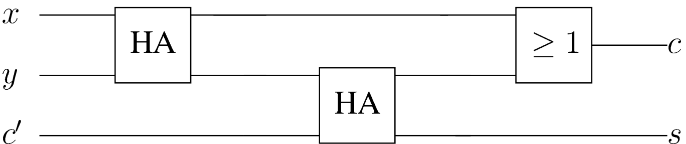
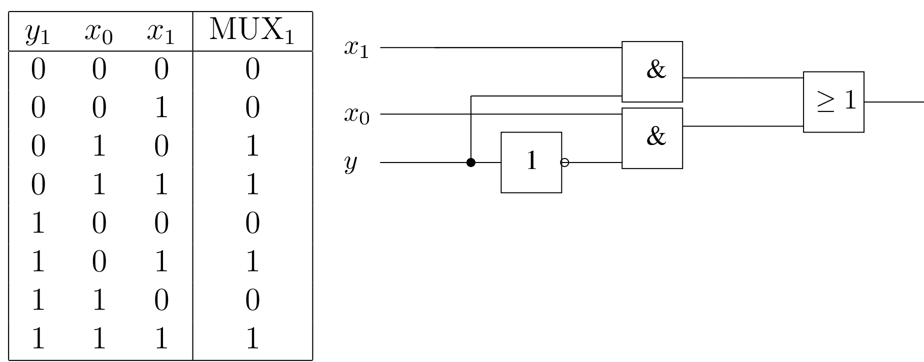


Abbildung 16: Volladdierer aus Halbaddierern

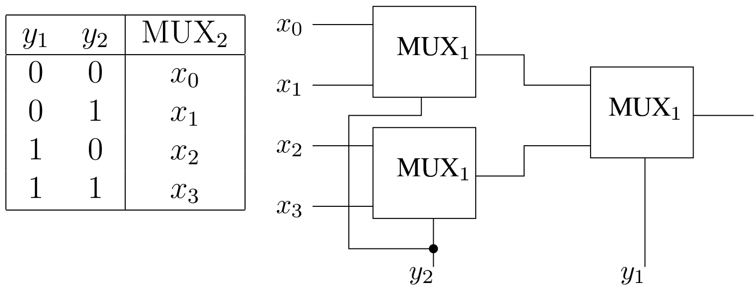
Wir wiederholen dieses Vorgehen noch einmal für den Multiplexer. Wir haben uns bisher ad hoc einen Multiplexer für $d = 3$ entworfen. Diesmal wollen wir systematischer vorgehen und beginnen erst einmal mit einem 1-Multiplexer. Wir können leicht die vollständige Wertetabelle und einen dazu passendes Schaltnetz angeben (Abbildung 17).



y_1	x_0	x_1	MUX_1
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Abbildung 17: Wertetabelle und Schaltnetz für MUX_1

Jetzt kommen wir zum 2-Multiplexer. Ein Blick in die vereinfachte Wertetabelle (Abbildung 18) verrät uns, dass wir zunächst der Steuerleitung y_2 mit Hilfe eines 1-Multiplexers hilfreiche Informationen entnehmen können. Es kann so sowohl entschieden werden, welche Datenleitung von x_0 und x_1 als auch welche Datenleitung von x_2 und x_3 in Frage kommen. Wir treffen diese Entscheidung mit Hilfe von zwei 1-Multiplexern. Dann haben wir noch zwei Datenleitungen; welche davon die auszuwählende ist, entscheidet die andere Steuerleitung y_1. Wir können also insgesamt mit drei 1-Multiplexern einen 2-Multiplexer realisieren. Wir stellen das Resultat in Abbildung 18 dar.



y_1	y_2	MUX_2
0	0	x_0
0	1	x_1
1	0	x_2
1	1	x_3

Abbildung 18: Vereinfachte Funktionstabelle und Schaltnetz für MUX_2

Unsere Überlegungen zum 2-Multiplexer lassen sich leicht verallgemeinern. Nehmen wir an, dass wir einen d-Multiplexer haben und daraus einen $2d$-Multiplexer konstruieren wollen. Wir können die „hinteren“ d Adressbits (also $y_{d+1}, \ldots, y_{2d}$) benutzen, um aus den ersten 2^d Datenbits eines zu selektieren; dazu verwenden wir natürlich einen d-Multiplexer. Ohne Kenntnis der anderen Adressbits wissen wir, dass dieses Datenbit das selektierte Datenbit sein könnte und dass die anderen $2^d - 1$ Datenbits mit Sicherheit nicht selektiert sind. Unsere Kenntnis bezüglich des selektieren Bits gründet sich darauf, dass ein gesetztes Bit in den ersten d Adressbits bedeutet, dass gar kein Bit aus den ersten 2^d Datenbits selektiert ist. Ist zum Beispiel von diesen ersten d Adressbits nur genau das letzte gesetzt, so wird ein Bit aus den zweiten 2^d Datenbits selektiert. Welche das ist, ist in den hinteren d Adressbits codiert. Die $2^{2d} = 2^d \cdot 2^d$ Datenbits zerfallen also auf natürliche Weise in 2^d Blöcke von je 2^d Bits. Die hinteren d Adressbits reichen aus, um aus jedem dieser Blöcke ein Datenbit als potenziell selektiert auszuwählen. Das können wir mit 2^d d-Multiplexern realisieren. Aus den 2^d Ergebnissen dieser d-Multiplexer können wir dann mit einem weiteren d-Multiplexer und den ersten d Adressbits das passende Bit selektieren. Das Ergebnis stellen wir in Abbildung 19 dar.

Wir können also aus $2^d + 1$ d-Multiplexern einen $2d$-Multiplexer bauen. Der Entwurf hat den Vorteil, gut strukturiert und darum offensichtlich korrekt zu sein. Natürlich entstehen so sehr große Multiplexer. Nennen wir mal $S(d)$ die Größe eines d-Multiplexers (der allerdings $2^d + d$ Eingänge hat). Wir können die Negationsgatter ignorieren, da wir die nicht auf jeder Ebene neu zu bilden brauchen. Es genügt, im gesamten Schaltnetz jedes Steuerbit einmal negiert zur Verfügung zu stellen. Also gilt $S(1) = 3$. Wenn wir die Negationsgatter dann korrekt mitzählen wollen, brauchen wir nur d zu addieren. Aufgrund der Konstruktion haben wir direkt $S(2d) = (2^d + 1) \cdot S(d)$. Man überzeugt sich leicht mittels vollständiger Induktion, dass $S(d) = 3 \cdot (2^d - 1)$ gilt. Wir sehen, dass wir leider schon für kleine d sehr große Schaltnetze erhalten.

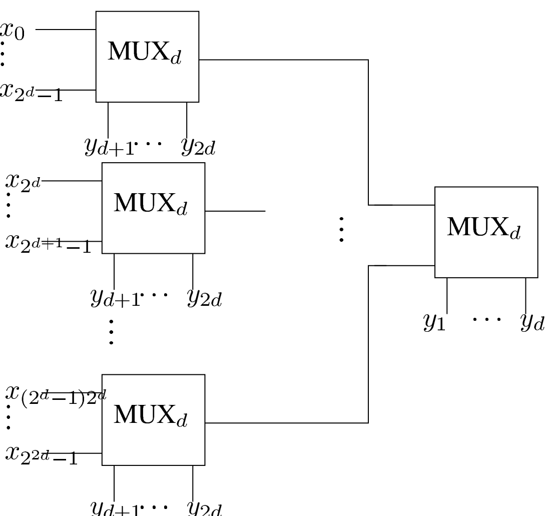


Abbildung 19: Strukturiert entworfener $2d$-Multiplexer

Etwas näher am Begriff der Optimierung sind Überlegungen zum Entwurf von kleineren Schaltnetzen. Wir wollen dazu noch einmal auf die Darstellung von booleschen Funktionen durch ihre Normalform zurückkommen. Ausgehend von einer solchen Normalform können wir direkt ein Schaltnetz für die Funktion erhalten, indem wir für jede Verknüpfung einen entsprechendes Gatter benutzen. Wir erhalten dann ein Schaltnetz mit so vielen Gattern, wie der Ausdruck Verknüpfungen hatte. Dabei können wir die Negationen außer acht lassen, da es ja wie schon beim Multiplexer erläutert ausreicht, jede Variable höchstens einmal zu negieren. Wenn man es nun schafft durch Anwendung der Rechenregeln aus Satz 3 einen äquivalenten aber kürzeren Ausdruck für die Funktion zu finden, führt das direkt zu einem kleineren Schaltnetz. Weil dieses Vorgehen direkt auf Anwendung der Rechenregeln der booleschen Algebra beruht, kann man von algebraischer Vereinfachung sprechen.

Wir wollen uns das an einem Beispiel ansehen und betrachten die Funktion $f\colon B^4 \to B$, die durch den Wertevektor $(1,0,1,0,1,0,1,0,1,1,1,1,0,0,0,0)$ gegeben ist. Wir starten mit ihrer disjunktiven Normalform, die genau acht Minterme enthält. Also ist

$$\begin{aligned} f(x_1,x_2,x_3,x_4) = {} & \overline{x_1}\,\overline{x_2}\,\overline{x_3}\,\overline{x_4} \vee \overline{x_1}\,\overline{x_2}\,x_3\,\overline{x_4} \vee \overline{x_1}\,x_2\,\overline{x_3}\,\overline{x_4} \vee \overline{x_1}\,x_2\,x_3\,\overline{x_4} \\ & \vee x_1\,\overline{x_2}\,\overline{x_3}\,\overline{x_4} \vee x_1\,\overline{x_2}\,\overline{x_3}\,x_4 \vee x_1\,\overline{x_2}\,x_3\,\overline{x_4} \vee x_1\,\overline{x_2}\,x_3\,x_4 \end{aligned}$$

und wir können mit dem Rechnen loslegen. Man erkennt durch scharfes Hin-

sehen, dass man gleich vier Mal die Resolutionsregel anwenden kann und erhält dann

$$f(x_1, x_2, x_3, x_4) = \overline{x_1}\,\overline{x_2}\,\overline{x_4} \lor \overline{x_1}\,x_2\,\overline{x_4} \lor x_1\,\overline{x_2}\,\overline{x_3} \lor x_1\,\overline{x_2}\,x_3,$$

worauf man wieder gleich zwei Mal die Resolutionsregel anwenden kann. Wir haben also nun

$$f(x_1, x_2, x_3, x_4) = \overline{x_1}\,\overline{x_4} \lor x_1\,\overline{x_2}$$

nachgewiesen, was offenbar erheblich kleiner ist. Man sollte sich zur Veranschaulichung die beiden zugehörigen Schaltnetze aufzeichnen.
Die erzielten Vereinfachungen sind beim betrachteten Beispiel sicher beeindruckend. Aber es wäre schön, einen etwas systematischeren Weg zu haben, solche Vereinfachungen zu finden. Für boolesche Funktionen mit drei oder vier Variablen gibt es ein anschauliches grafisches Verfahren, das wir jetzt kennen lernen wollen. Die Verallgemeinerung auf mehr als vier Variablen ist prinzipiell möglich, dann wird allerdings die Darstellung sehr unübersichtlich, so dass das Verfahren nicht mehr wirklich praktikabel ist.
Zu einer booleschen Funktion $f\colon B^4 \to B$ definieren wir ein so genanntes KV-Diagramm (Maurice Karnaugh und Edward W. Veitch haben die Methode unabhängig voneinander entwickelt und in getrennten Artikeln 1952 bzw. 1953 veröffentlicht). Es handelt sich um eine 4×4 Matrix, die also 16 Einträge hat, für jede Variablenbelegung einen. Die Spalten beschriften wir mit Belegungen von $x_1\,x_2$, die Zeilen mit Belegungen von $x_3\,x_4$, dabei unterscheiden sich zwei benachbarte Belegungen immer in genau einem Bit. Die Nachbarschaft ist dabei zyklisch, neben den üblichen Nachbarschaften sind auch die erste und vierte Spalte sowie die erste und vierte Zeile benachbart. Eine schematische Darstellung findet man in Abbildung 20.

$x_3\,x_4$ \ $x_1\,x_2$	00	01	11	10
00				
01				
11				
10				

Abbildung 20: KV-Diagramm für $f\colon B^4 \to B$

Es ist jetzt also jeder der 16 Matrixeinträge genau einer Belegung der Variablen zugeordnet; wir schreiben in die einzelnen Zellen nun die zugehörigen Funktionswerte; zur Verbesserung der Übersicht bietet es sich an, nur Einseinträge tatsächlich durchzuführen und die mit 0 belegten Zellen einfach leer

zu lassen. In dieser Matrix suchen wir jetzt Rechtecke maximaler Größe, dabei beachten wir die zyklische Nachbarschaft und betrachten nur Rechtecke, deren Kantenlängen Zweierpotenzen sind, also Länge 1, 2 oder 4 haben. Einem solchen Rechteck ordnen wir dann eine Konjunktion der Literale (also der Variablen bzw. negierten Variablen) zu, die in diesem Rechteck nur in einer Belegung vorkommen. Ein Beispiel findet man in Abbildung 21. Wir sehen, dass wir zwei 2×2 Rechtecke identifizieren können, die alle Einseinträge abdecken. Dass es nur zwei Rechtecke sind, dass sie gleiche Größe haben und dass sie überschneidungsfrei sind, liegt natürlich an der speziellen Funktion. Bei anderen Funktionen kann das jeweils anders sein. Wir haben also die beiden Konjunktion $x_2\, x_4$ und $\overline{x_2}\, \overline{x_4}$. Die dargestellt Funktion ergibt sich als Disjunktion all dieser Konjunktionen. Wir erkennen im Beispiel folglich, dass die Funktion $f(x_1, x_2, x_3, x_4) = x_2\, x_4 \vee \overline{x_2}\, \overline{x_4}$ dargestellt wird.

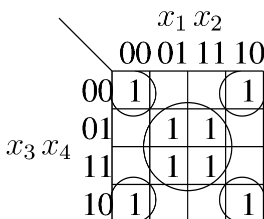


Abbildung 21: KV-Diagramm für $f\colon B^4 \to B$

Hat man eine Funktion $f\colon B^3 \to B$, so lässt man die unteren zwei Zeilen der Matrix weg und erhält eine 2×4 Matrix, die Zeilen sind dabei mit $x_3 = 0$ und $x_3 = 1$ beschriftet. Ansonsten wird das Verfahren ganz unverändert angewendet. Natürlich kann man für eine Funktion $f\colon B^2 \to B$ auch die hinteren zwei Spalten weglassen und zu einer 2×2 Matrix kommen, die ebenso behandelt werden kann. Das lohnt sich aber kaum, da bei nur zwei Variablen auch die oben diskutierte algebraische Vereinfachung nicht schwieriger ist.

Funktionen mit nur vier Variablen sind natürlich winzig. Darum brauchen wir dringend Verfahren, die auch für Funktionen mit mehr Variablen funktionieren. Bevor wir konkret dazu kommen, werden wir noch einige Begriffe, mit denen wir schon eine Weile implizit umgehen, explizit definieren. Das erleichtert es uns, über die Dinge zu sprechen und unsere Vorstellungen klarer zu artikulieren.

Eine Variable oder eine negierte Variable nennen wir ein *Literal*. Eine Konjunktion von Literalen nennen wir ein *Monom*. Wir interessieren uns hier für die Darstellung boolescher Funktionen als Disjunktion von Monomen. Wir nennen diese Darstellung, also eine Disjunktion einiger Monome, ein *Polynom*. In einer solchen Darstellung ist offensichtlich der Funktionswert genau

dann 1, wenn mindestens eines der Monome den Wert 1 annimmt. Ein Monom m heißt *Implikant* einer Funktion f, wenn es diese Eigenschaft hat, wenn also

$$m(x_1, x_2, \ldots, x_n) = 1 \Rightarrow f(x_1, x_2, \ldots, x_n) = 1$$

gilt. Wenn wir eine disjunktive Darstellung von f haben, so können wir die Größe eines Schaltnetzes für f schätzen, indem wir in jedem Monom die Literale zählen und diese Anzahlen addieren. Wir bezeichnen diese Summe als Kosten des Polynoms. Diese Kosten entsprechen exakt der Anzahl der Eingänge von Bausteinen für die Konjunktion in einer Schaltnetzrealisierung. Sie sind also ein ungefähres Maß für die tatsächlichen Kosten einer solchen Schaltnetzrealisierung, geben diese Kosten aber nicht exakt wieder. Trotzdem wollen wir die Kosten des Polynoms untersuchen und als realistische Schätzung für die Größe eines Schaltnetzes verwenden. Es geht uns jetzt also um die Minimierung von Polynomen; ein Polynom mit minimalen Kosten nennen wir *Minimalpolynom*.

Wir nennen ein Monom m_v eine *Verkürzung eines Monoms m*, wenn m_v nur Literale enthält, die auch in m vorkommen. Wir nennen ein Monom m_v eine *echte Verkürzung eines Monoms m*, wenn m_v eine Verkürzung von m ist und weniger Literale als m enthält. Weil Implikanten auch Monome sind, können wir diese Begriffe direkt auf Implikanten übertragen. Es ist klar, dass ein Minimalpolynom niemals einen Implikanten und eine seiner Verkürzungen enthält, da die Verkürzung ausreicht. Das motiviert die Einführung des Begriffs des Primimplikanten: ein *Primimplikant* (PI) einer Funktion f ist ein Implikant m zu f, der keine echte Verkürzung hat, die auch Implikant zu f ist. Wir sehen, dass ein Minimalpolynom nur Primimplikanten enthält.

Ein Primimplikant deckt möglichst viele Einseinträge der Wertetabelle von f ab. Wir brauchen für ein Minimalpolynom möglichst wenige Primimplikanten, die alle Einseinträge überdecken. Es bietet sich jetzt also an, die Berechnung eines Minimalpolynoms zweischrittig durchzuführen: im ersten Schritt werden alle Primimplikanten berechnet, im zweiten dann eine kostenminimale Auswahl durchgeführt. Das haben wir schon gemacht, als wir uns mit KV-Diagrammen beschäftigt haben. Jedes maximale Rechteck mit Zweierpotenzseitenlängen entspricht genau einem Primimplikanten. Die minimale Überdeckung, die wir im Anschluss gesucht haben, entspricht gerade der kostenminimalen Auswahl. Leider funktionieren KV-Diagramme wirklich gut nur für Funktionen $f \colon B^n \to B$ mit $n \leq 4$. Wir sind aber oft an Funktionen mit viel mehr Variablen interessiert. Dazu braucht man ein allgemeineres Verfahren, das wir nun kennenlernen wollen. Wir betrachten den Algorithmus von Quine und McCluskey, den wir in zwei Abschnitten vorstellen wollen. Wir beginnen mit der Berechnung aller Primimplikanten.

Der Algorithmus startet mit einer Liste aller Implikanten zu f, also mit einer Liste alle Minterme zu einschlägigen Indizes. Es spielt keine große Rolle, ob wir eine solche Liste als Eingabe für den Algorithmus verlangen oder von einer Wertetabelle (oder einem Wertevektor) ausgehen und diese Liste in einem Vorbereitungsschritt berechnen lassen.

Algorithmus 11 (Quine/McCluskey; Berechnung aller PI).

1. L_0 `ist Liste aller Minterme;` i`:=0`
2. PI `:=` $\emptyset$
3. `So lange` $L_i \neq \emptyset$
4. $\quad L_{i+1}$ `:=` $\{m \mid \exists x_j \colon \{m\,x_j, m\,\overline{x_j}\} \subseteq L_i\}$
5. $\quad PI$ `:=` $PI \cup \{m \in L_i \mid m$ `hat keine echte Verkürzung in` $L_{i+1}\}$
6. $\quad i$ `:=` $i+1$

Wenn man alle Primimplikanten zu einer Funktion f hat, braucht man zur Bestimmung eines Minimalpolynoms „nur noch“ eine möglichst kleine Teilmenge davon auswählen, die alle Einseingaben überdeckt.
Wir beschränken uns jetzt hier auf die Auswahl einer möglichst kleinen Menge von Primimplikanten und berücksichtigen die Anzahl der Literale in den Primimplikanten nicht mehr. Dieses vereinfachte Kostenmaß wird durch die in Abschnitt 4.7 eingeführten PLAs motiviert, bei denen es uns auf die Anzahl der verwendeten Primimplikanten ankommen wird, es aber unwichtig sein wird, wie groß die Anzahl der Literale in jedem Primimplikanten ist.
Für eine möglichst günstige Auswahl der Primimplikanten führen wir zunächst den Begriff der *Primimplikantentafel* (PI-Tafel) ein. Für jeden Primimplikanten hat diese Tabelle eine Zeile, für jeden Einseingabe eine Spalte. Wir tragen in der i-ten Zeile in der j-ten Spalte (also an Position (i, j)) eine 1 ein, wenn der i-te Primimplikant die j-te Eingabe überdeckt, wenn also der Implikant für die durch die Spalte definierte Belegung der Variablen 1 berechnet. Alle anderen Positionen werden mit 0 beschriftet.
Wir werden jetzt zunächst versuchen, diese Tabelle zu verkleinern. Dazu beschreiben wir drei Regeln, die es jeweils erlauben, eine Zeile oder eine Spalte zu streichen.

Streichung überdeckender Spalten: *Wenn eine Spalte s an jeder Stelle eine 1 hat, an der eine andere Spalte s' eine 1 hat, so kann die Spalte s gestrichen werden.* Alle Primimplikanten, die s' überdecken, überdecken auch s. Da s' auf jeden Fall überdeckt werden muss, wird s auf jeden Fall überdeckt und muss nicht explizit berücksichtigt werden.

Streichung von Kernimplikanten-Zeilen: *Wenn in einer Spalte nur genau eine 1 steht, heißt der zugehörige Implikant* Kernimplikant. *Er*

wird zur Überdeckung benutzt. Die Zeile und jede Spalte, die in dieser Zeile eine 1 hat, kann gestrichen werden. Die betroffene Spalte kann nur durch diesen Kernimplikanten überdeckt werden. Er muss also zu jeder Primimplikanten-Darstellung von f gehören. Natürlich muss kein Einseintrag öfter als einmal überdeckt werden, so dass alle überdeckten Spalten gestrichen werden können.

Streichung überdeckter Zeilen: *Wenn eine Zeile z an jeder Stelle eine 1 hat, an der eine andere Zeile z' eine 1 hat, so kann die Zeile z' gestrichen werden.* Der Primimplikant in der Zeile z' überdeckt nur eine (echte) Teilmenge der Einseinträge der von z überdeckten Einseinträge. Darum ist es auf keinen Fall ungünstiger, z zur Überdeckung zu benutzen.

Die Streichung überdeckter Zeilen wird aufmerksame Leserinnen und Leser vielleicht überraschen. Wenn sie bei einer gerade aufgestellten Primimplikantentafel anwendbar wäre, so hätten wir offenbar einen Fehler gemacht. Der zu streichende Implikant müsste dann eine echte Verkürzung sein und könnten ganz offenbar kein Primimplikant sein. Diese Situation kann also anfangs nicht gegeben sein. Nachdem allerdings die anderen Regeln angewendet worden sind, kann auch die dritte Regel anwendbar werden.
Nach Anwendung aller Regeln muss gewählt werden. Man glaubt, dass das Problem eine kostenoptimale Auswahl zu finden, sehr schwierig zu lösen ist. Eine genaue Begründung für diese Vermutung muss auf ein späteres Semester verschoben werden auf einen Zeitpunkt, zu dem die NP-Vollständigkeitstheorie bekannt ist. Wir begnügen uns darum mit einer heuristischen Auswahl: Man wählt den Primimplikanten einer Zeile mit möglichst vielen Einseinträgen. Danach können wieder die drei Streichungsregeln erschöpfend angewendet werden. So verfährt man, bis die Tabelle keine Zeilen und Spalten mehr enthält. Wenn die Berechnung einer optimalen Auswahl erforderlich ist, ist ein Vergleich aller nach erschöpfender Anwendung der drei Streichungsregeln möglichen Auswahlen eine Option.
Ein wichtiger Sonderfall, den wir noch kurz erwähnen wollen, sind *unvollständig definierte boolesche Funktionen*, die man auch partiell definierte boolesche Funktionen nennt. Formal können wir solche Funktionen als Funktion $f\colon B^n \to \{0,1,*\}$ definieren, wobei wir „$*$“ als eine Art „don’t care“-Symbol verstehen: es ist gleichgültig, welchen Funktionswert die Funktion dort annimmt. Eine unvollständig definierte boolesche Funktion $f\colon B^n \to \{0,1,*\}$ wird also durch jede boolesche Funktion $f'\colon B^n \to B$ realisiert, für die

$$\forall x \in \{0,1\}^n\colon (f(x) = *) \lor (f(x) = f'(x))$$

gilt. Weil es uns zur Realisierung von f genügt, eine sie realisierende boolesche Funktion zu realisieren, haben wir bei der Wahl der konkreten booleschen Funktion große Freiheit. Diese können wir zum Beispiel nutzen, um in einem KV-Diagramm die Einträge so mit Nullen und Einsen zu ergänzen, dass wir möglichst wenige Primimplikanten bekommen. In gewisser Weise wird das Problem der Realisierung von f also dadurch einfacher. Andererseits kann so viel Wahlfreiheit auch eine Bürde sein: Wie wählen wir die Belegungen möglichst geschickt? In den meisten Fällen ist es auch tatsächlich keine wesentliche Vereinfachung: Wir betrachten zwei spezielle Erweiterungen einer unvollständig definierten booleschen Funktion zu vollständig definierten booleschen Funktionen: die Funktion f_1 entsteht, indem man alle frei wählbaren Werte mit 1 besetzt. Analog definieren wir f_0; hier besetzt mal alle frei wählbaren Werte mit 0. Zentral ist die folgende Beobachtung.

Satz 12. *Sei $f\colon B^n \to \{0, 1, *\}$ eine partiell definierte boolesche Funktion. Sei $f_1\colon B^n \to B$ die Ergänzung von f, die an allen freien Stellen den Wert 1 annimmt. Minimalpolynome von f enthalten nur Primimplikanten von f_1.*

Beweis. Sei $p = m_1 \vee m_2 \vee \cdots \vee m_k$ ein Minimalpolynom für f. Offenbar stellt auch p eine Ergänzung f_e von f dar und ist deren Minimalpolynom. Folglich enthält p nur Primimplikanten von f_e. Jeder Primimplikant von dieser Ergänzung f_e ist ein Implikant von f_1, da f_1 ja sicher dann den Funktionswert 1 hat, wenn f_e Funktionswert 1 hat. Sei m ein solcher Primimplikant zu f_e, dann gibt es also einen Primimplikanten m' von f_1, der Verkürzung von m ist (natürlich nicht notwendig echte Verkürzung). Ersetzen wir m durch m', so bleibt p ein Polynom für f. Weil p dadurch nicht kürzer werden kann – sonst wäre es kein Minimalpolynom – muss für jedes Monom in p gelten, das es Primimplikant von f_1 ist. □

Wir können jetzt also systematisch vorgehen: Zuerst berechnen wir alle Primimplikanten von f_1, danach suchen wir eine möglichst günstige Überdeckung der Einsstellen von f_0. Neue Algorithmen brauchen wir also für diesen Fall nicht zu suchen.

Bevor wir diesen Abschnitt verlassen, wollen wir über den verfolgten Ansatz noch einmal kritisch nachdenken. Wir berechnen zu einer Funktion ein Minimalpolynom, indem wir in einem ersten Schritt die Menge aller Primimplikanten berechnen und dann im zweiten Schritt eine möglichst günstige Auswahl treffen. Dieser zweischrittige Ansatz ist natürlich dann besonders ungünstig, wenn eine Funktion ein kleines Minimalpolynom aber sehr viele Primimplikanten hat. Aber gibt es solche Funktionen überhaupt? Leider können wir uns davon überzeugen, dass das in der Tat der Fall ist.

Satz 13. *Für jedes $k \in \mathbb{N}$ gibt es Funktionen $f\colon B^n \to B$ mit $n = 2k-1$, die durch k Monome dargestellt werden können und $2^k - 1$ Primimplikanten haben.*

Beweis. Wir definieren die Funktionen induktiv und beginnen mit der Funktion $f_1\colon B^1 \to B$, die wir durch $f_1(x_1) := x_1$ definieren. Offenbar hat f genau den einen Primimplikanten x_1, der f_1 darstellt. Es gilt hier $k = 1 = 2^k - 1$. Wir gehen nun induktiv davon aus, dass die Funktion $f_k\colon B^{2k-1} \to B$ schon definiert ist und definieren die Funktion $f_{k+1} \to B^{2(k+1)-1} \to B$ durch

$$f_{k+1}(x_1, x_2, \ldots, x_{2k+1}) := x_{2k} f_k(x_1, x_2, \ldots, x_{2k-1}) \vee \overline{x_{2k}}\, x_{2k+1}.$$

Gemäß Induktionsvoraussetzung ist f_k durch ein Polynom mit k Monomen dargestellt, wir erhalten also für f_{k+1} ein Polynom mit $k+1$ Monomen. Uns fehlt noch der Nachweis über die Anzahl der Primimplikanten von f_{k+1}. Dabei dürfen wir wieder gemäß Induktionsvoraussetzung davon ausgehen, dass f_k genau $2^k - 1$ Primimplikanten hat.
Offensichtlich ist $\overline{x_{2k}}\, x_{2k+1}$ Primimplikant von f_{k+1}, weil die beiden Variablen neu sind und nicht beide in den anderen Monomen vorkommen. Wir werden jetzt zeigen, dass für jeden Primimplikanten $m \in \mathrm{PI}(f_k)$ stets $x_{2k}m \in \mathrm{PI}(f_{k+1})$ und $x_{2k+1}m \in \mathrm{PI}(f_{k+1})$ gilt. Damit haben wir als Anzahl der Primimplikanten von f_{k+1} gerade $2 \cdot \left(2^k - 1\right) + 1 = 2^{k+1} - 1$ wie behauptet.
Wir müssen also nur noch zeigen, dass $x_{2k}m$ und $x_{2k+1}m$ beides Primimplikanten von f_{k+1} sind, wenn m Primimplikant von f_k ist. Natürlich ist $x_{2k}m$ jedenfalls Implikant von f_{k+1}, so ist f_{k+1} ja gerade definiert. Es ist aber auch $x_{2k+1}m$ Implikant von f_{k+1}: Wir betrachten dazu eine Belegung $a \in B^{2k+1}$ und nehmen an, dass $x_{2k+1}m(a) = 1$ gilt. Falls $a_{2k} = 0$ gilt, so ist $\overline{x_{2k}}x_{2k+1}(a) = 1$ und es gilt $f_{k+1}(a) = 1$. Ist andererseits $a_{2k} = 1$, dann ist $x_{2k}m(a) = 1$ und es gilt wiederum $f_{k+1}(a) = 1$. Also ist wie behauptet auch $x_{2k+1}m$ jedenfalls Implikant von f_{k+1}. Um nachzuweisen, dass es sich auch in beiden Fällen um Primimplikanten handelt, betrachten wir nun ihre echten Verkürzungen. Man sieht leicht ein, dass m kein Implikant von f_{k+1} ist. Die Belegung von $x_{2k} = x_{2k+1} = 0$ lässt f_{k+1} den Funktionswert 0 annehmen, unabhängig von m. Betrachten wir nun eine echte Verkürzung m' von m. Offenbar kann $x_{2k}m'$ kein Implikant von f_{k+1} sein, sonst wäre ja m' ein Implikant von f_k. Das steht aber im Widerspruch zur Voraussetzung, dass m Primimplikant von f_k ist. Es bleibt noch der Fall, $x_{2k+1}m'$ Implikant von f_{k+1} ist. Dann betrachten wir eine Belegung $b \in B^{2k-1}$ mit $m'(b) = 1$. Wir definieren die Belegung $a \in B^{2k+1}$ als Verlängerung von b um $a_{2k} = a_{2k+1} = 1$. Offenbar ist dann auch $x_{2k+1}m'(a) = 1$. Wir nehmen an, dass $x_{2k+1}m'$ Implikant von f_{k+1} ist, dann muss auch $f_{k+1}(a) = 1$ gelten. Für das „neue“ Monom gilt $\overline{x_{2k}}x_{2k+1}(a) = 0$, also muss auch $f_k(a') = 1$ gelten, damit $f_{k+1}(a) = 1$

gelten kann. Folglich ist m' ein Implikant von f_k, das steht aber im Widerspruch zur Voraussetzung, dass m Primimplikant von f_k ist. Also haben wir jetzt insgesamt, dass sowohl $x_{2k}m$ als auch $x_{2k+1}m$ Primimplikanten von f_{k+1} sind und der Beweis ist vollständig. □

Diese Funktionen stellen natürlich die Sinnhaftigkeit unseres Ansatzes, immer erst die Menge aller Primimplikanten zu berechnen, in Frage. Es sind aber leider keine wesentlichen besseren Algorithmen bekannt. Eine Diskussion, warum es sehr überraschend wäre, effiziente Algorithmen zu finden, müssen wir auf die Vorlesung „Grundbegriffe der theoretischen Informatik“ bzw. „Theoretische Informatik für Studierende der Angewandten Informatik“ verschieben.

4.6 Hazards

Wenn wir uns die technische Realisierung eines Schaltnetzes ansehen, so gibt es in der Realität natürlich Abweichungen vom idealisierten Modell. Ein ganz wichtiger Aspekt in dieser Beziehung sind Signallaufzeiten auf Leitungen und Schaltzeiten der einzelnen Bausteine. Von Bedeutung ist das, wenn es dazu kommt, dass bei einem Wechsel der Eingabe der am Ausgang erzeugte Wert nicht immer dem korrekten Funktionswert entspricht oder unerwartet wechselt. Man spricht in diesem Fall von einem *Hazard*.
Wir unterscheiden zwei verschiedene Arten von Hazards, *Funktionshazards* und *Schaltungshazards*. Ein Funktionshazard hat nichts mit der Schaltung zu tun und tritt schon an der Funktion auf. Wir definieren gleich exakt, was wir damit meinen. Falls eine Funktion keinen Funktionshazard aufweist, können an der Schaltung trotzdem Hazards auftreten; diese Hazards nennen wir dann Schaltungshazard.
Bei beiden Arten von Hazards unterscheiden wir noch zwei Fälle. Wir sprechen von einem *statischen Hazard*, wenn der Wechsel der Eingabe nicht zu einer Änderung des Ausgangswertes führt, ein solcher Ausgabenwechsel aber zwischenzeitlich auftritt (auftreten kann). Ein so genannter *dynamischer Hazard* liegt vor, wenn beim Wechsel der Eingabe sich der Wert am Ausgang ändern soll, tatsächlich aber mehr als ein Wechsel auftritt. Wir beschränken uns hier im Wesentlichen auf statische Hazards; dynamische Hazards sind nicht wesentlich verschieden.
Wir wollen noch formal exakt fassen, was wir unter einem statischen Funktionshazard verstehen. Es sei $f\colon B^n \to B$ eine boolesche Funktion. Es seien $a, b \in B^n$ mit $a \neq b$ und $f(a) = f(b)$ gegeben. Für $x \in B^n$ und $i \in \{1, \ldots, n\}$ bezeichne x_i das i-te Bit in x. Wir sagen, f hat einen statischen Funkti-

onshazard für den Wechsel von a nach b, wenn es ein $c \in B^n$ gibt, für das $c_i \in \{a_i, b_i\}$ für alle $i \in \{1, \dots, n\}$ und $f(c) \neq f(a)$ gilt.
Die Funktion $f(x_1, x_2, x_3) = \overline{x_1}\,\overline{x_2}\,x_3$ zum Beispiel hat einen statischen Funktionshazard für den Übergang von 000 nach 011. Wechselt nämlich die Eingabe in der Reihenfolge 000, 001, 011, so liegt für 001 der Funktionswert 1 an, obwohl $f(000) = f(011) = 0$ gilt.
Ob eine Funktion einen Funktionshazard hat, kann man anschaulich im KV-Diagramm erkennen: Um zu überprüfen, ob für den Wechsel von a nach b ein Funktionshazard vorliegt, kann man sich alle kürzesten Wege von a nach b im KV-Diagramm der Funktion ansehen und auf Funktionswertwechsel achten. Wenn es mehr davon gibt als minimal nötig auf mindestens einem dieser kürzesten Wege, gibt es einen Funktionshazard. Offensichtlich ist das Auftreten von Hazards in Schaltungen, die Funktionen mit Funktionshazard realisieren, nicht ohne besondere Maßnahmen vermeidbar. Aber was ist mit Schaltungshazards?
Betrachten wir jetzt noch die Funktion $f(x_1, x_2, x_3) = x_1\,x_2 \vee \overline{x_2}\,x_3$, für die wir ein Schaltnetz in Abbildung 22 sehen.

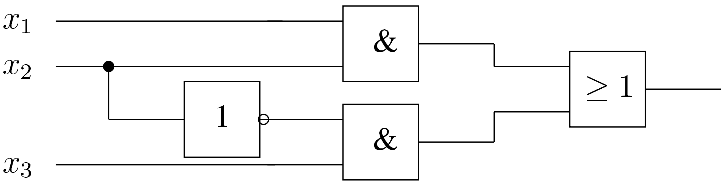


Abbildung 22: Schaltungshazard beim Übergang von 111 nach 101

Für den Übergang von 111 nach 101 kann kein Funktionshazard vorliegen, da ja gar kein anderer Eingabewert „dazwischen liegt“. Es kann aber einen Schaltungshazard geben: Anfangs ist der Ausgang auf 1, da das obere Und-Gatter 1 berechnet. Wechselt x_2 von 1 nach 0 und schalten das obere Gatter zusammen mit dem Disjunktions-Gatter schneller als die beiden unteren Gatter, so liegt für eine kurze Zeit am Ausgang eine 0 an, obwohl der Funktionswert natürlich 1 ist.
Wenn man bereit ist, zusätzlichen Hardwareaufwand zu betreiben, lassen sich solche Schaltungshazards vermeiden. Wir betrachten zweistufige (dreistufig, wenn man die Negationen mitzählt) Schaltnetze in disjunktiver Form. Das Schaltnetz in Abbildung 22 ist ein solches Schaltnetz. Wenn ein solches Schaltnetz ein Und-Gatter für *jeden* Primimplikanten von f enthält und jedes Und-Gatter einem Primimplikanten entspricht, so hat das Schaltnetz keine statischen Schaltungshazards. Beim Beispiel oben ist der Primimplikant $x_1\,x_3$ nicht realisiert, so dass ein Schaltungshazard beobachtbar ist.

Der Beweis dieser Aussage ist relativ leicht. Wir nehmen an, dass für den Übergang von a nach b kein Funktionshazard vorliegt und $f(a) = f(b)$ gilt. Wir nehmen an, dass sich a und b in nur einem Bit unterscheiden, andernfalls argumentiert man analog für alle Zwischenschritte c. Gilt $f(a) = f(b) = 0$, so ist kein Primimplikant von f 1 für a oder b, so dass bei allen Und-Gattern nur 0 anliegen kann. Interessanter ist der Fall $f(a) = f(b) = 1$. Wir betrachten die beiden Minterme zu a und b. Weil a und b sich in nur einem Bit unterscheiden, können wir die Resolutionsregel anwenden. Das Ergebnis ist eine Verkürzung der Minterme, ein Implikant der entweder selbst Primimplikant ist oder eine Verkürzung hat, die Primimplikant ist. In jedem Fall hängt dieser Primimplikant nicht von dem sich ändernden Bit ab und darum liegt an dem Und-Gatter dieses Primimplikanten die ganze Zeit 1 an, so dass sich der Ausgabewert nicht ändert.

4.7 Programmierbare Bausteine

Wenn man in der Praxis eine Schaltung realisieren will, ist vor allem der Aspekt der Wirtschaftlichkeit zu berücksichtigen. Wir wollen kleine, schnelle und kostengünstige Schaltungen haben. Wenn sehr viele Exemplare einer Schaltung benötigt werden, kann natürlich ein spezieller Entwurf durchgeführt und in einer hochintegrierten Schaltung eigens realisiert werden. Wenn aber nur verhältnismäßig kleine Stückzahlen gebraucht werden, ist die Produktion eines solchen speziellen Chips im Allgemeinen zu kostspielig. Einen Ausweg aus dieser Situation bieten programmierbare Bausteine, die eine wohldefinierte Grundfunktionalität zur Verfügung stellen, in ihrem Verhalten beeinflusst – also programmiert – werden können und so prinzipiell jede boolesche Funktion realisieren können. Dabei kann diese Programmierung entweder permanent geschehen (durch einen physikalischen Eingriff, zum Beispiel ätzen), oder auch reversibel sein.
Wir konkretisieren diese Ausführungen am so genannten *programmierbaren logischen Feld* (programmable logic array, kurz PLA). Wir stellen uns zunächst einen einfachen Grundbaustein mit zwei Eingängen und zwei Ausgängen vor, der in vier verschiedenen „Typen“ existiert, wobei jeder „Typ“ eine andere boolesche Funktion realisiert. Wir wählen eine etwas andere Darstellung als bisher und stellen uns diesen Grundbaustein wie in Abbildung 23 vor. Wir bezeichnen die unterschiedlichen Typen einfach mit Betragszahlen, es ist also $t \in \{0, 1, 2, 3\}$. Je nach Typ werden unterschiedliche Funktionen $f_1 \colon B^2 \to B$ und $f_2 \colon B^2 \to B$ realisiert, die wir jeweils benennen. In Tabelle 11 findet man eine Übersicht. Es ist offensichtlich, dass die zur Verfügung stehenden Funktionen als Menge funktional vollständig sind: Der Multiplizierer realisiert die Konjunktion, der Negat-Multiplizierer die Negation, wenn

wir nur den Eingang x konstant mit 1 belegen. Außerdem ist klar, dass alle vier Baustein-Typen leicht zu realisieren sind: Für den Identer wird gar kein Gatter gebraucht, für den Multiplizierer und den Addierer reicht jeweils ein Gatter, für den Negat-Multiplizierer reichen zwei Gatter, wobei eines ein besonders leicht zu realisierendes Negationsgatter ist.

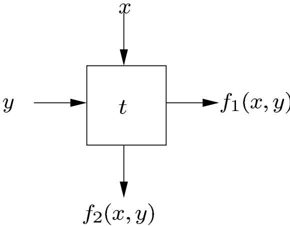


Abbildung 23: Grundbaustein eines PLA

Typ	Name	$f_1(x, y)$	$f_2(x, y)$
0	Identer	y	x
1	Addierer	$x \vee y$	x
2	Multiplizierer	y	$x \wedge y$
3	Negat-Multiplizierer	y	$x \wedge \overline{y}$

Tabelle 11: In PLA-Grundbausteinen realisierte Funktionen

Ein PLA ist eine rechteckige Anordnung solcher Bausteine mit l Eingängen und Ausgängen sowie k Spalten, so wie sie in Abbildung 24 zu sehen ist (für $l = 5$ und $k = 4$).

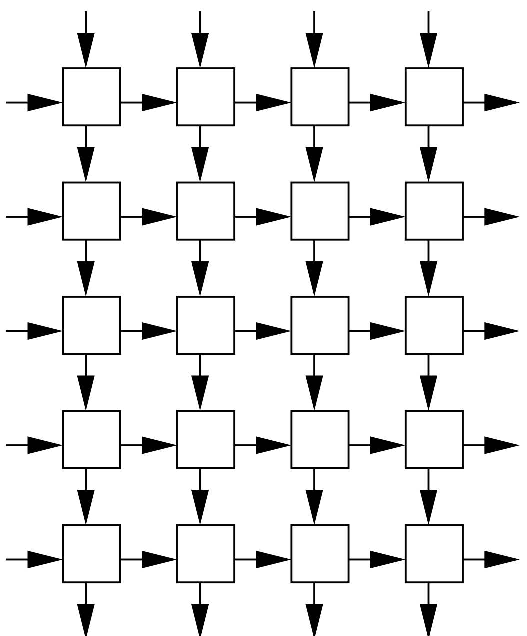


Abbildung 24: PLA mit fünf Eingängen und vier Spalten

Der Typ jeder Zelle ist nicht bei der Produktion festgelegt; das kann später beim Einsatz bestimmt werden. PLAs sind also dank Massenproduktion preiswert zu produzieren und universell einzusetzen. Bei der Verwendung hält man sich an die folgenden Konventionen: Zur Realisierung einer Funktion $f\colon B^n \to B^m$ wählt man ein PLA mit $l = n + m$ Zeilen und einer ausreichend großen Anzahl von Spalten. Was „ausreichend groß“ genau bedeutet überlegen wir uns etwas später. Man unterscheidet einen oberen und einen unteren Teil des PLA. Der obere Teil besteht aus den ersten n Zeilen, der untere aus den unteren m Zeilen. Die n Variablen der booleschen Funktion liegen links im oberen Teil als Eingänge an, im unteren Teil liegt in den Zeilen die Konstante 0 an. In den Spalten liegt oben überall die Konstante 1 an. Im oberen Teil verwendet man als Typen nur Identer, Multiplizierer und Negat-Multiplizierer (0, 2 und 3). Man überlegt sich, dass man dadurch beliebige Konjunktionen von Variablen und negierten Variablen in den oberen n Zeilen realisieren kann. Wir können also zum Beispiel alle Minterme einer Funktion so realisieren, jedenfalls wenn die Anzahl der Spalten im PLA groß genug ist. Im unteren Teil (m Zeilen) verwendet man nur noch Identer und Addierer (0 und 1), so dass man dort dann von den von oben kommenden Konjunktionen beliebige Disjunktionen bilden kann. Dabei kann jede Disjunktion natürlich nur höchstens so viele Elemente haben, wie Spalten vorhanden sind. Das motiviert jetzt in gewisser Weise die Verwendung von Minimalpolynomen noch

einmal: Wenn wir eine Darstellung mit wenig Implikanten finden, können wir ein kleines PLA wählen. Allerdings ist die Situation etwas anders als bei Minimalpolynomen: In einem PLA kann in jeder Spalte ein Implikant realisiert werden. Weil in jeder Ausgabezeile aber beliebig aus den Spalten gewählt werden kann, kann man solche Implikanten bei Funktionen mit mehr als einem Ausgabebit mehrfach verwenden. Es ist dann nicht mehr unmittelbar klar, dass es günstig ist, für jede Funktion ein Minimalpolynom zu berechnen. Tatsächlich kann man sich leicht an einem Beispiel klar machen, dass man durch Mehrfachverwendung von Implikanten günstigere Realisierungen erhalten kann als durch ausschließliche Verwendung von Minimalpolynomen.
Wir betrachten die Funktionen

$$\begin{aligned} p_1(x_1, x_2, x_3) &= x_1\, x_3 \vee \overline{x_2}\, x_3 \\ p_2(x_1, x_2, x_3) &= \overline{x_1}\, x_2 \vee x_2\, x_3 \end{aligned}$$

und sehen direkt, dass es sich jeweils um ein Minimalpolynom handelt. Wenn wir an Realisierung in PLAs denken, können wir die Kosten ermitteln, indem wir zunächst wieder für jedes Monom die Anzahl der Literale zählen, diesmal aber mehrfach vorkommende Monome nicht mehrfach zählen. Das ergibt für p_1 Kosten $2 + 2 = 4$ und für p_2 Kosten $2 + 2 = 4$ getrennt betrachtet, wir realisieren also beide Funktionen gemeinsam in einem PLA mit Kosten $4 + 4 = 8$, weil kein Monom mehrfach vorkommt.
Wir betrachten nun

$$\begin{aligned} p'_1(x_1, x_2, x_3) &= x_1\, x_2\, x_3 \vee \overline{x_2}\, x_3 \\ p'_2(x_1, x_2, x_3) &= \overline{x_1}\, x_2 \vee x_1\, x_2\, x_3 \end{aligned}$$

und stellen fest, dass sowohl $p'_1(x_1, x_2, x_3) = p_1(x_1, x_2, x_3)$ als auch $p'_2(x_1, x_2, x_3) = p_2(x_1, x_2, x_3)$ für alle Belegungen $(x_1, x_2, x_3) \in B^3$ gilt, es werden also die gleichen Funktionen dargestellt. Wir haben für p'_1 Kosten $3 + 2 = 5$ und für p'_2 Kosten $2 + 3 = 5$. Weil aber $x_1\, x_2\, x_3$ in einem PLA nur einmal realisiert werden muss, haben wir insgesamt Kosten $3 + 2 + 2 = 7$, also tatsächlich weniger als bei der Lösung mit Minimalpolynomen.
Wir wollen uns überlegen, wie wir eine kostenminimale Lösung berechnen können. Dazu halten wir zunächst noch einmal formal fest, was wir eigentlich genau wollen.

Definition 14. *Ein* Minimalpolynom *für* $f\colon B^n \to B^m$ *ist eine Folge von* m *Polynomen* $(p_1, p_2, \ldots, p_m)$ *mit minimalen Kosten, dabei ist* p_i *ein Polynom für* f_i*. Bei den Kosten zählen mehrfach vorkommende Monome nur einmal.*

Bei der Minimalpolynomberechnung spielten Primimplikanten eine zentrale Rolle: Monome, die Implikanten sind und die keine echte Verkürzung haben, die ebenfalls Implikant ist. Wir haben bewiesen, dass Minimalpolynome für boolesche Funktionen $f\colon B^n \to B$ nur aus Primimplikanten bestehen. Wir gehen hier analog vor, allerdings wird der Begriff des Primimplikanten bei Funktionen $f\colon B^n \to B$ durch den Begriff des multiplen Primimplikanten ersetzt.

Definition 15. *Ein Monom m ist ein* multipler Primimplikant *von $f = (f_1, f_2, \dots, f_k)$ mit $f_i\colon B^n \to B$, wenn m Primimplikant von $\bigwedge\limits_{i \in I} f_i$ ist für eine nicht-leere Menge $I \subseteq \{1, 2, \dots, k\}$.*

Satz 16. *Minimalpolynome für boolesche Funktionen $f\colon B^n \to B^k$ enthalten nur multiple Primimplikanten von f.*

Beweis. Wir betrachten ein Minimalpolynom $(p_1, p_2, \dots, p_k)$ für f und ein Monom m daraus. Es muss eine nicht-leere Indexmenge $I \subseteq \{1, 2, \dots, k\}$ geben, so dass m ein Implikant von f_i ist für alle $i \in M$. Andernfalls stellt das Polynom, in dem m vorkommt, nicht die passende Funktion dar. Folglich ist m zumindest Implikant von $\bigwedge\limits_{i \in I} f_i$. Wir zeigen durch einen Widerspruchsbeweis, dass m auch Primimplikant von $\bigwedge\limits_{i \in I} f_i$ ist. Dazu nehmen wir an, dass m nicht Primimplikant von $\bigwedge\limits_{i \in I} f_i$ ist, es gibt also eine echte Verkürzung m' von m, die ebenfalls Implikant von $\bigwedge\limits_{i \in I} f_i$ ist. Wenn wir in $(p_1, p_2, \dots, p_k)$ nun m durch m' ersetzen, sinken dadurch die Kosten. Das Ergebnis dieser Ersetzung nennen wir $(p'_1, p'_2, \dots, p'_k)$. Es bleibt zu überprüfen, ob weiterhin die gleiche Funktion f dargestellt wird.

Wir betrachten dazu zu einem beliebigen $i \in I$ das Polynom p'_i für f_i und seinen Funktionswert $p'_i(x)$ für ein beliebiges $x \in B^n$. Wir unterscheiden zwei Fälle: Ist $p'_i(x) = 0$, so ist $m'(x) = 0$ und dann natürlich auch $m(x) = 0$, weil ja m' eine echte Verkürzung von m ist. In diesem Fall ist also $p_i(x) = f_i(x) = 0$ und die Funktion wird weiterhin korrekt dargestellt. Im anderen Fall ist $p'_i(x) = 1$. Falls p'_i von p_i verschieden ist, liegt das daran, dass m' ein Implikant von p'_i ist. Weil gemäß Voraussetzung m' Implikant von $\bigwedge\limits_{i \in I} f_i$ ist, muss es auch Implikant von f_i sein. Also enthält p'_i nur Implikanten von f_i und wir haben $p'_i(x) = 1 \Rightarrow f_i(x) = 1$ und auch in diesem Fall wird die Funktion korrekt dargestellt.

Insgesamt haben wir, dass die Funktion trotz Durchführung der Verkürzung weiter korrekt dargestellt wird, so dass im Gegensatz zur Voraussetzung,

$(p_1, p_2, \dots, p_k)$ kein Minimalpolynom für f gewesen sein kann. Folglich muss die Annahme, dass m nicht Primimplikant von $\bigwedge\limits_{i \in I} f_i$ ist, falsch sein. □

Nachdem wir wissen, dass wir Minimalpolynome aus multiplen Primimplikanten zusammensetzen können, möchten wir natürlich alle multiplen Primimplikanten berechnen können – analog zur Berechnung alle Primimplikanten für alle Funktionen $f\colon B^n \to B$. Dabei wird uns das folgende Theorem 17 wertvolle Hilfe sein.

Satz 17. *Sei für nicht-leere Mengen* $I \subseteq \{1, 2, \dots, k\}$ *die Menge* $PI\left(\bigwedge\limits_{i \in I} f_i\right)$ *die Menge aller Primimplikanten von* $\bigwedge\limits_{i \in I} f_i$ *für eine boolesche Funktion* $f = (f_1, f_2, \dots, f_k)$ *mit* $f_i\colon B^i \to B$ *für alle* $i \in \{1, 2, \dots, k\}$. *Dann gilt:*

$$m \in PI\left(\bigwedge_{i \in I} f_i\right) \Rightarrow \forall i \in I\colon \exists m_i \in PI(f_i)\colon m = \bigwedge_{i \in I} m_i$$

Beweis. Wir betrachten ein $m \in \mathrm{PI}\left(\bigwedge\limits_{i \in I} f_i\right)$, das natürlich für alle $i \in I$ Implikant von f_i ist. Wir betrachten für jedes $i \in I$ jeweils ein Monom $m_i \in \mathrm{PI}(f_i)$, dass Verkürzung von m ist (nicht notwendig eine echte Verkürzung). Dann ist natürlich $\bigwedge_{i \in I} m_i$ ebenfalls eine Verkürzung von m. Weil m ein Primimplikant von $\bigwedge\limits_{i \in I} f_i$ ist, kann offenbar $\bigwedge_{i \in I} m_i$ keine echte Verkürzung von m sein. Also ist $\bigwedge_{i \in I} m_i = m$. □

Theorem 17 sagt uns, dass wir multiple Primimplikanten aus Primimplikanten für die einzelnen booleschen Funktionen f_i zusammensetzen können. Wie wir Primimplikanten berechnen, hatten wir uns ja ausführlich überlegt. Wir können dieses Basiswissen zusammen mit dieser Erkenntnis zu einem Algorithmus zur Berechnung der multiplen Primimplikanten ausbauen.

Algorithmus 18 (Berechnung multipler Primimplikanten)**.**

1. `Für alle` $i \in \{1, 2, \dots, k\}$ `berechne` $M_{\{i\}} := \mathtt{PI}(f_i)$.
2. $M := \bigcup\limits_{i=1}^{k} \{\{i\}\}$; $M' := \bigcup\limits_{i=1}^{k} M_{\{i\}}$.
3. `Wiederhole`
4.   `Für` $I, J \in M$ `mit` $I \cap J = \emptyset$
5. $M_{I \cup J} := \{m \mid m = m'm''$ `mit` $m' \in M_I$, $m'' \in M_J$,
        `keine echte Verkürzung von` m `in` $M_{I \cup J}\}$.
6. $M := M \cup \{I \cup J\}$; $M' := M' \cup M_{I \cup J}$.
7. `So lange, bis` M' `eine Iteration unverändert bleibt.`

Der Algorithmus terminiert, weil die Menge der multiplen Primimplikanten endlich ist, folglich kann M nicht unbeschränkt wachsen. Seine Korrektheit ergibt sich unmittelbar aus Theorem 17. Analog zur Situation bei booleschen Funktionen $f\colon B^n \to B$ müssen wir hier nach der Berechnung aller multiplen Primimplikanten noch eine möglichst günstige Überdeckung aller Einsen suchen. Wir wollen das hier aber nicht mehr vertiefen und lieber zu PLAs, dem Thema dieses Abschnitts zurückkehren.

Eine erwähnenswerte Anwendung von PLAs ist die Realisierung eines ROM-Bausteines. Nehmen wir an, dass wir 2^n Worte, die aus jeweils m Bits bestehen, fest speichern wollen. Wir verwenden dazu ein PLA mit $n + m$ Zeilen und 2^n Spalten. Die Realisierung ist sehr einfach und schematisch, steht allerdings im Vergleich zur gerade beschriebenen Standardform in gewisser Weise auf dem Kopf: Wir führen die n Adressvariablen, welche ja zur Adressierung von 2^n Worten genau ausreichen, in den oberen n Zeilen als Eingangsvariablen ein. Die unteren m Zeilen erhalten die Konstante 0 als Eingang. In allen Spalten verwenden wir oben die Konstante 1 als Eingang. Das gespeicherte Wort soll in den unteren m Zeilen als Ausgabe anliegen. In den oberen n Zeilen verwenden wir ausschließlich Multiplizierer und Negat-Multiplizierer und das auf sehr regelmäßige Art und Weise. Wir denken uns die Spalten von links nach rechts beginnend mit 0 durchnummeriert, wir schreiben dann für ein Bit b der Binärdarstellung dieser Spaltennummer $3 - b$ als Bausteintypen in die Spalten. Zur Klärung hilft sicher das in Tabelle 12 dargestellte Beispiel. In den unteren m Zeilen schreiben wir in die Spalten einfach die Bits des zu speichernden Wortes als Bausteintypen.

Warum ist diese Realisierung korrekt? Wir beginnen mit den oberen n Zeilen. Weil nur die Typen 2 und 3 verwendet werden, werden die Adressvariablen jeweils in den Zeilen unverändert weitergereicht, so dass in jeder Spalte die unveränderten Adressvariablen zur Verfügung stehen. Liegt nun eine Adresse an, so stimmt sie entweder in allen Bausteinen einer Spalte überein oder es gibt mindestens einen abweichenden Baustein. Stimmt ein Baustein überein (also gibt es Eingabe 0 bei Bausteintyp 3 bzw. Eingabe 1 bei Bausteintyp 2), so sieht man direkt, dass eine 1 nach unten weitergegeben wird, falls von oben eine 1 eingegeben wird. Stimmen hingegen Baustein und Eingabe nicht überein, so wird auf jeden Fall in der Spalte eine 0 weitergegeben, unabhängig vom Eingabewert in der Zeile. Folglich wird in genau einer Spalte eine 1 berechnet und zwar genau in der durch die Adressvariablen adressierten Spalte. In den unteren m Zeilen reichen die Bausteine vom Typ 0 einfach die in den Zeilen anliegenden Nullbits weiter. Am Ausgang kann nur dann eine 1 anliegen, wenn in dieser Zeile wenigstens in einer Spalte ein Baustein vom Typ 1 vorhanden ist. Eine solche 1 wird aber nur dann nach rechts durchgegeben, wenn in der entsprechenden Zeile von oben eine

1 kommt, also die entsprechende Spalte adressiert ist. – Die Situation mag etwas unübersichtlich erscheinen, ist im Grunde aber klar strukturiert und nicht schwierig zu überschauen. Es lohnt sich auf jeden Fall, sich eine konkrete Adresse zu wählen und das Beispiel aus Tabelle 12 für diese Eingabe konkret durchzugehen.

x_2	3	3	3	3	2	2	2	2	
x_1	3	3	2	2	3	3	2	2	
x_0	3	2	3	2	3	2	3	2	
	$w_{0,0}$	$w_{1,0}$	$w_{2,0}$	$w_{3,0}$	$w_{4,0}$	$w_{5,0}$	$w_{6,0}$	$w_{7,0}$	$w_{(x)_2,0}$
	$w_{0,1}$	$w_{1,1}$	$w_{2,1}$	$w_{3,1}$	$w_{4,1}$	$w_{5,1}$	$w_{6,1}$	$w_{7,1}$	$w_{(x)_2,1}$
	$w_{0,2}$	$w_{1,2}$	$w_{2,2}$	$w_{3,2}$	$w_{4,2}$	$w_{5,2}$	$w_{6,2}$	$w_{7,2}$	$w_{(x)_2,2}$
	$w_{0,3}$	$w_{1,3}$	$w_{2,3}$	$w_{3,3}$	$w_{4,3}$	$w_{5,3}$	$w_{6,3}$	$w_{7,3}$	$w_{(x)_2,3}$

Tabelle 12: ROM-Realisierung ($8 = 2^3$ Worte der Länge 4) durch ein PLA

Man kann PLAs einmalig durch Veränderung der Hardware programmieren; ätzen ist zum Beispiel eine Möglichkeit, das zu realisieren. In gewisser Weise noch interessanter ist aber eine softwaremäßige Realisierung, die beliebig wieder verändert werden kann. Weil wir genau $4 = 2^2$ verschiedene Bausteintypen in einem PLA haben, könnte man sich einen Grundbaustein vorstellen, der zwei zusätzliche Steuerleitungen als Eingänge hat, deren Belegung dann seinen Typ festlegt. Gesucht ist also im Grunde eine boolesche Funktion $f\colon B^4 \to B^2$, wie sie in Tabelle 13 dargestellt ist.

Typ	s	t	f_1	f_2
0	0	0	y	x
1	0	1	$x \vee y$	x
2	1	0	y	$x \wedge y$
3	1	1	y	$x \wedge \overline{y}$

Tabelle 13: Funktionale Beschreibung PLA-Steuerung

Es ist nicht schwer zu verifizieren, dass wir mit $f_1(x, y, s, t) = y \vee \overline{s}\, t\, x$ und $f_2(x, y, s, t) = \overline{s}\, x \vee s\, x(y \oplus t)$ eine geeignete Realisierung gefunden haben, für die wir leicht ein entsprechendes Schaltnetz angeben könnten. Wir brauchen jetzt also zur Programmierung eines solchen PLA nur je Baustein zwei Bits zu speichern; folglich genügt uns ein ROM-Baustein (ein fester Speicher), der $2(n + m)k$ Bits aufnehmen kann, um eine Funktion $f\colon B^n \to B^m$ mit k Monomen in einer Polynomdarstellung zu realisieren. In Zeiten fallender ROM-Preise ist das ganz sicher kein Problem.

5 Sequenzielle Schaltungen

Bei der Konstruktion von Schaltnetzen ist eine unumstößliche Regel, dass die „Verdrahtung“ des Schaltnetzes keinen Kreis bilden darf. Die Einführung dieser Regel haben wir nie diskutiert, aber sie ist sicher leicht zu begründen, wie ein Blick auf Abbildung 25 schnell deutlich macht.

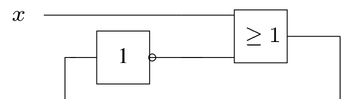


Abbildung 25: Flimmerschaltung

Abbildung 25 stellt offenbar kein Schaltnetz dar: es ist ja ein Kreis enthalten: der Ausgang des Oder-Gatters wird zurückgeführt und als Eingang des Negations-Gatters verwendet. Trotzdem ist klar, dass diese Schaltung technisch realisiert werden kann. Wir können sowohl Negations-Gatter als auch Oder-Gatter realisieren, die gezeigte Verdrahtung ist sicher nicht schwer nachzubauen. Was wird denn in der gezeigten Schaltung eigentlich realisiert? Nehmen wir an, dass $x = 1$ gilt. Dann wird am Ausgang des Oder-Gatters unabhängig vom anderen Eingang eine 1 berechnet. Es gibt in dem Fall also kein Problem. Wenn nun $x = 0$ ist, so kommt es auf den zweiten Eingang an: Wenn am Negations-Gatter eine 0 anliegt, so erhält das Oder-Gatter als Eingabe eine 1 und berechnet eine 1. Wenn am Negations-Gatter eine 1 anliegt, so erhält das Oder-Gatter als Eingabe eine 0 und berechnet eine 0. Das Fatale ist, dass die Eingabe des Negations-Gatters gerade die vom Oder-Gatter berechnete Ausgabe ist und hier ein direkter Widerspruch besteht. Das ist sicher nicht schön. Was passiert denn jetzt in der Realität? Begeisterte Science-Fiction-Leser vermuten vielleicht, dass sich die Schaltung „in einem rosa Logik-Wölkchen“ auflöst. Die Realität ist weniger spektakulär: Die Gatter haben ja alle gewisse Schaltzeiten, die Signale auf den Leitungen gewisse Laufzeiten. Darum liegt in sehr schnell wechselnder Folge am Ausgang des Oder-Bausteins abwechselnd eine 0 und eine 1 an. Aus diesem Grund haben wir die Schaltung in der Bildunterschrift auch als Flimmerschaltung bezeichnet. Für die Praxis ist sie so natürlich erst einmal nicht brauchbar. Die Idee, einen Ausgang wieder als Eingang zu benutzen, ist aber so dumm eigentlich nicht: wenn man kontrollieren könnte, wann und wie Ausgangswerte wieder als Eingabe verwendet werden, wären sicher interessante Dinge realisierbar: man könnte sich vorstellen, etwa nicht-feste Speicher realisieren zu können – eine zentrale Funktion von Computern, um die wir uns bisher noch nicht gekümmert haben, weil wir uns bisher noch nicht darum kümmern konnten.

Wir wollen uns noch ein zweites Beispiel für eine solche Schaltung „mit Kreis“ (wir nennen solche Schaltungen übrigens *Schaltwerk*) ansehen und betrachten Abbildung 26.

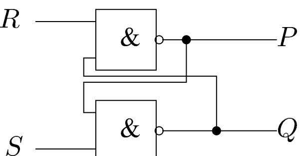


Abbildung 26: NAND-Kippstufe

Wir möchten gerne das Verhalten dieser Schaltung beschreiben; als ein geeignetes Instrument haben wir schon Wertetabellen kennen gelernt. Allerdings hängen die Werte jetzt nicht nur von den Eingaben sondern zusätzlich von der Zeit ab. Wir wollen das berücksichtigen, indem wir jeden Wert mit einem Zeitindex t versehen und einmal grob vereinfachend annehmen, dass die Schaltzeiten adäquat mit Δ beschrieben werden können, die Angabe $t+\Delta$ also ein weiterer sinnvoller Zeitindex ist. Wir sehen uns also alle vier möglichen Belegungen für R_t und S_t an und halten fest, was denn für $P_{t+\Delta}$ und $Q_{t+\Delta}$ sowie $P_{t+2\Delta}$ und $Q_{t+2\Delta}$ gilt. Dabei nehmen wir an (was in der Realität natürlich auch zutrifft), dass zu diesem Zeitpunkt auf den anderen beiden Eingängen der NAND-Gatter schon P_t und Q_t anliegen. Welche Werte konkret anliegen, hängt davon ab, welches der beiden NAND-Gatter schneller schaltet. Wenn wir annehmen, dass das obere Gatter schneller schaltet, so liegt immer zuerst der neue Wert für P an, was dann zu Tabelle 14 führt.

R_t	S_t	$P_{t+\Delta}$	$Q_{t+\Delta}$	$P_{t+2\Delta}$	$Q_{t+2\Delta}$
0	0	1	1	1	1
0	1	1	0	1	0
1	0	$\overline{Q_t}$	1	0	1
1	1	$\overline{Q_t}$	Q_t	$\overline{Q_t}$	Q_t

Tabelle 14: Wertetabelle zur NAND-Kippstufe (oben schneller)

Weil die Situation in der Tat etwas verwickelt erscheinen kann, besprechen wir Tabelle 14 zeilenweise. Wir erinnern uns zunächst daran, dass $\text{NAND}(x, y) = 0$ genau dann gilt, wenn $x = y = 1$ gilt, andernfalls wird 1 berechnet. Liegt also bei R und S 0 an, so wird unabhängig von der anderen Eingabe auf jeden Fall 1 berechnet, das ändert sich auch nicht. Für $(R, S) = (0, 1)$ erinnern wir uns zunächst daran, dass das obere Gatter schneller schaltet. Wegen $R_t = 0$ berechnet es eine 1, dann berechnet das langsamer schaltende zweite Gatter

eine 0; auch das ändert sich nicht mehr. Ist $(R, S) = (1, 0)$, so berechnet das obere (schnellere) Gatter einen Wert, der genau das Komplement der anderen Eingabe ist, also $\overline{Q_t}$. Das untere Gatter berechnet aber auf jeden Fall eine 1, weil als Eingabe mindestens eine 0 anliegt. Folglich berechnet das obere Gatter im zweiten Schritt auf jeden Fall eine 0. Bleibt noch die letzte Zeile, $(R, S) = (1, 1)$. Beim oberen Gatter wird wie schon im letzten Fall $\overline{Q_t}$ berechnet. Da am unteren Gatter bei S_t auch eine 1 anliegt, wird nun das Komplement der Ausgabe des schnelleren Gatters berechnet, also $\overline{\overline{Q_t}} = Q_t$; spätere Änderung gibt es nicht.
Nehmen wir jetzt noch an, dass das untere Gatter schneller als das obere schaltet. Die Argumentation für die $(0, 0)$-Zeile ändert sich offenbar nicht. Die Argumentation für die $(0, 1)$-Zeile trifft nun gerade auf die $(1, 0)$-Zeile zu und umgekehrt. Bei der vierten Zeile ändert sich wieder nichts. Das führt uns also zu Tabelle 15.

R_t	S_t	$P_{t+\Delta}$	$Q_{t+\Delta}$	$P_{t+2\Delta}$	$Q_{t+2\Delta}$
0	0	1	1	1	1
0	1	1	$\overline{P_t}$	1	0
1	0	0	1	0	1
1	1	P_t	$\overline{P_t}$	P_t	$\overline{P_t}$

Tabelle 15: Wertetabelle zur NAND-Kippstufe (oben langsamer)

Wir schauen uns die Schaltung aus Abbildung 26 noch einmal an und beschäftigen uns mit den Tabellen 14 und 15 aus etwas mehr „algebraischer" Sicht. Es ist nicht schwer, einen funktionalen Zusammenhang zwischen den jeweiligen Werten von P und Q in Abhängigkeit von R, S, P und Q anzugeben, wobei der genau funktionale Zusammenhang natürlich davon abhängt, welches Gatter schneller schaltet. Nehmen wir zunächst an, dass das obere Gatter schneller schaltet, also Tabelle 14 gilt. Dann erhalten wir folgende Rechnung. Weil wir annehmen, dass die Eingaben R und S für längere Zeit unverändert anliegen, können wir hier auf den Zeitindex verzichten.

$$\begin{aligned} P_{t+2\Delta} = \overline{R\,Q_{t+\Delta}} = \overline{R} \vee \overline{Q_{t+\Delta}} = \overline{R} \vee \overline{\overline{S\,P_{t+\Delta}}} = \overline{R} \vee S\,P_{t+\Delta} = \overline{R} \vee S\left(\overline{R\,Q_t}\right) \\ = \overline{R} \vee S\left(\overline{R} \vee \overline{Q_t}\right) = \overline{R} \vee S\,\overline{Q_t} \end{aligned}$$

Führen wir die gleiche Rechnung für den anderen Fall (also gemäß Tabelle 15) durch, so erhalten wir folgende Gleichungskette.

$$\begin{aligned} P_{t+2\Delta} = \overline{R\,Q_{t+2\Delta}} = \overline{R} \vee \overline{Q_{t+2\Delta}} = \overline{R} \vee \overline{\overline{S\,P_{t+\Delta}}} = \overline{R} \vee S\,P_{t+\Delta} = \overline{R} \vee S\left(\overline{R\,Q_{t+\Delta}}\right) \\ = \overline{R} \vee S\left(\overline{R} \vee \overline{Q_{t+\Delta}}\right) = \overline{R} \vee S\,\overline{Q_{t+\Delta}} = \overline{R} \vee S\left(\overline{\overline{S\,P_t}}\right) = \overline{R} \vee S\left(S\,P_t\right) = \overline{R} \vee S\,P_t \end{aligned}$$

Wir sehen direkt, dass im Fall $P_t = Q_t$ das Ergebnis von den Schaltzeiten abhängt. Ist aber $P_t \neq Q_t$ (also $P_t = \overline{Q_t}$), so erhalten wir in beiden Fällen das gleiche Ergebnis. Wir bekommen also ein vorhersagbares, stabiles Verhalten für P, wenn wir ausschließen, dass P und Q den gleichen Wert haben.
Jetzt untersuchen wir auf die gleiche Weise das Verhalten von Q. Tabelle 14 folgend ergibt sich

$$\begin{aligned} Q_{t+2\Delta} &= \overline{S\,P_{t+2\Delta}} = \overline{S} \vee \overline{P_{t+2\Delta}} = \overline{S} \vee \overline{\overline{R\,Q_{t+\Delta}}} = \overline{S} \vee R\,Q_{t+\Delta} = \overline{S} \vee R\left(\overline{S\,P_{t+\Delta}}\right) \\ &= \overline{S} \vee R\left(\overline{S} \vee \overline{P_{t+\Delta}}\right) = \overline{S} \vee R\,\overline{P_{t+\Delta}} = \overline{S} \vee R\,\overline{\left(\overline{R\,Q_t}\right)} = \overline{S} \vee R\,(R\,Q_t) = \overline{S} \vee R\,Q_t \end{aligned}$$

während sich Tabelle 15 folgend

$$\begin{aligned} Q_{t+2\Delta} &= \overline{S\,P_{t+\Delta}} = \overline{S} \vee \overline{P_{t+\Delta}} = \overline{S} \vee \overline{\overline{R\,Q_{t+\Delta}}} = \overline{S} \vee R\,Q_{t+\Delta} = \overline{S} \vee R\left(\overline{S\,P_t}\right) \\ &= \overline{S} \vee R\left(\overline{S} \vee \overline{P_t}\right) = \overline{S} \vee R\,\overline{P_t} \end{aligned}$$

ergibt. Wir erhalten also auch für Q genau dann das wünschenswerte, stabile und vorhersagbare Verhalten, wenn $P \neq Q$ gilt. Wir sehen den ersten Zeilen beider Tabellen an, dass wir $P = Q$ erhalten, wenn wir als Eingabe $R = S = 0$ wählen. Wir verbieten jetzt also diese Eingabe und erhalten die verkürzte Tabelle 16.

R_t	S_t	$P_{t+2\Delta}$	$Q_{t+2\Delta}$
0	1	1	0
1	0	0	1
1	1	$P_t = \overline{Q_t}$	$Q_t = \overline{P_t}$

Tabelle 16: Wertetabelle zur NAND-Kippstufe (oben langsamer)

Das Verhalten der Schaltung ist ausgesprochen interessant. Mit den Eingaben $(0,1)$ und $(1,0)$ können wir die Ausgaben zu jedem der beiden erlaubten Ausgabeverhalten zwingen. Mit der Eingabe $(1,1)$ wird die vorherige Ausgabe mit Verzögerung 2Δ wieder ausgegeben. Wir haben also eine Schaltung realisiert, die als 1-Bit-Speicher dienen kann: Es kann ein Bit mit beliebigem Wert gespeichert und dann beliebig lange im Speicher gehalten werden. Problematisch ist, dass die Verzögerung von 2Δ etwas unklar ist und wir absolut sicher stellen müssen, dass niemals $R_t = S_t = 0$ anliegt. Wenn wir von der Eingabe $(0,1)$ zur Eingabe $(1,0)$ wechseln wollen, kann das aber passieren, wir erinnern uns an das Thema Hazards (Abschnitt 4.6). Wir werden uns darum in Abschnitt 5.2 mit einer grundlegenden Verbesserung der gleichen Schaltungsidee beschäftigen.

5.1 Modellbildung

Wir haben schon gesehen, dass der Übergang von Schaltnetzen zu Schaltwerken uns einiges an zusätzlicher Komplexität und Kompliziertheit eingebracht hat. Wir haben lange und hart arbeiten müssen, bis wir herausgefunden hatten, dass die ja im Grund sehr einfache Schaltung aus Abbildung 26 ein ganz passabler 1-Bit-Speicher ist. Wäre unsere Aufgabe gewesen, einen solchen 1-Bit-Speicher zu entwerfen, hätten wir uns sicher noch schwerer getan. In der Praxis ist es aber nicht so, dass Schaltungen vom Himmel fallen und man die Aufgabe hat herauszufinden, was sie eigentlich tun. Die bei weitem typischere Aufgabe ist der Entwurf einer Schaltung. Dabei wird es oft so sein, dass ein Schaltnetz, also die Realisierung einer booleschen Funktion nicht ausreicht. Wir wollen Schaltungen realisieren, die auf Eingaben reagieren und geeignete Ausgaben erzeugen; dabei sind die Eingaben natürlich über die Zeit verteilt. Beispiele sind die Steuerung einer Ampelanlage, die auf Anforderungen durch Fußgänger und Autofahrer (per Induktionsschleife) geeignet reagieren muss, die Steuerung einer Waschmaschine, die bei gewähltem Waschprogramm und Status der Waschmaschine die geeigneten Aktionen (pumpen, Trommel drehen, schleudern, ...) durchführen muss oder der Getränke-Automat, der Bargeld und Getränkewunsch entgegennimmt und anschließend Getränk und passendes Wechselgeld ausgibt. Um nicht von der Komplexität der Aufgaben erschlagen zu werden, wollen wir einen strukturierten Ansatz wählen, der auf einer relativ hohen Abstraktionsebene beginnt. Wir definieren darum zunächst rein formal einen Automaten, der es uns erlaubt, die genannten Beispiele zu formalisieren. Erst in einem zweiten und späteren Schritt (in Abschnitt 5.2) wollen wir dann darüber sprechen, wie wir solche Automaten dann tatsächlich realisieren.

Der Automat, den wir gleich auch noch ganz formal beschreiben wollen arbeitet in diskreten Schritten, die wir *Takte* nennen. In jedem Takt wird ein Symbol der Eingabe gelesen und in Abhängigkeit vom aktuellen Zustand des Automaten verarbeitet. Bei dieser Verarbeitung kann sich der interne Zustand des Automaten ändern, außerdem wird eine Ausgabe erzeugt. Damit ist jetzt im Grunde klar, aus welchen Komponenten sich so ein Automat zusammensetzt.

Definition 19. *Ein* Mealy-Automat $M = (Q, q_0, \Sigma, \Delta, \delta, \lambda)$ *ist definiert durch eine endliche Menge von Zuständen Q, einen Startzustand $q_0 \in Q$, ein endliches Eingabealphabet Σ, ein endliches Ausgabealphabet Δ, eine Zustandsüberführungsfunktion $\delta\colon Q \times \Sigma \to Q$ und eine Ausgabefunktion $\lambda\colon Q \times \Sigma \to \Delta \cup \{\varepsilon\}$. Er verarbeitet eine endliche Eingabe $w = w_1 \cdots w_l \in \Sigma^*$ zeichenweise, startend im Zustand q_0. Ist der aktuelle Zustand $q \in Q$ und das aktuelle Eingabesymbol $w_i \in \Sigma$, so wechselt der Automat in den Zustand*

$\delta(q, w_i)$ und schreibt das Ausgabesymbol $\lambda(q, w_i)$. Das spezielle Ausgabesymbol ε kennzeichnet, dass keine Ausgabe erzeugt wird.

Wichtig ist, dass der Automat nur eine endliche Menge von Zuständen hat; darum genügt uns auch bei der Realisierung eine endliche Anzahl von Bits, um den aktuellen Zustand zu speichern. Außerdem legt die Übergangsfunktion zusammen mit der Ausgabefunktion das Verhalten des Automaten fest: bei einer festen Eingabe durchläuft der Automat eine feste Folge von Zuständen und erzeugt dabei eine feste Ausgabe. Es gibt keinerlei Wahlmöglichkeiten und keinen Zufall. Das entspricht unserer üblichen Vorstellung von einem Automaten und passt zu den oben genannten Beispielen.
In der Regel ist es nicht günstig, sich einen Mealy-Automaten als ein abstraktes 6-Tupel vorzustellen. Man bekommt leichter eine Vorstellung davon, was der Automat eigentlich macht, wenn man ihn graphisch darstellt. Dafür haben sich folgende Konventionen eingebürgert. Wir zeichnen für jeden Zustand $q \in Q$ einen Kreis (einen Knoten), in dem wir den Namen des Zustandes q notieren. Den Startzustand $q_0 \in Q$ umkreisen wir zur Kennzeichnung doppelt. Für jedes Eingabesymbol $w \in \Sigma$ zeichnen wir von jedem Zustand q aus einen Pfeil (eine gerichtete Kante) zum Nachfolgezustand, der sich ja als $\delta(q, w)$ ergibt. An die Kante schreiben wir ein Paar bestehend aus einem Eingabebuchstaben und einem Ausgabebuchstaben, nämlich $w/\lambda(q, w)$. Ein Beispiel für einen solchen Automaten mit den 15 Zuständen $q_0, q_1, \dots, q_{14}$, dem Eingabealphabet $\Sigma = \{0, 1\}$ und dem Ausgabealphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}\}$ findet man in Abbildung 27.
Es ist nicht schwer sich klar zu machen, dass der in Abbildung 27 dargestellte Mealy-Automat eine Zahl aus der Binärdarstellung in Hexadezimaldarstellung umrechnet, wenn die Länge der Zahl in Binärdarstellung durch vier teilbar ist; dabei können Zahlen anderer Länge mit führenden Nullen aufgefüllt werden.
Manchmal ist es günstiger, die Ausgabe eines Symbols nicht direkt mit einem Eingabezeichen und einem Zustand zu verknüpfen, sondern die Ausgabe gedanklich stärker mit den Zuständen zu verbinden. Man führt dazu ein anderes Automatenmodell ein, dass man *Moore-Automat* nennt.

Definition 20. *Ein* Moore-Automat $M = (Q, q_0, \Sigma, \Delta, \delta, \lambda)$ *ist definiert durch eine endliche Menge von Zuständen Q, einen Startzustand $q_0 \in Q$, ein endliches Eingabealphabet Σ, ein endliches Ausgabealphabet Δ, eine Zustandsüberführungsfunktion $\delta\colon Q \times \Sigma \to Q$ und eine Ausgabefunktion $\lambda\colon Q \to \Delta \cup \{\varepsilon\}$. Er verarbeitet eine endliche Eingabe $w = w_1 \cdots w_l \in \Sigma^*$ zeichenweise, startend im Zustand q_0. Ist der aktuelle Zustand $q \in Q$ und das aktuelle Eingabesymbol $w_i \in \Sigma$, so wechselt der Automat in den Zustand $\delta(q, w_i)$ und*

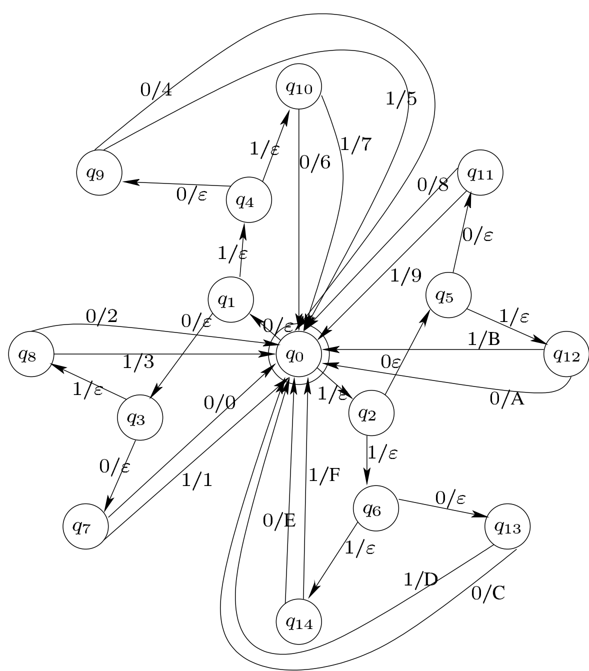


Abbildung 27: Mealy-Automat zur Umwandlung von Binärdarstellung in Hexadezimaldarstellung

schreibt das Ausgabesymbol $\lambda(\delta(q, w_i))$*. Das spezielle Ausgabesymbol* ε *kennzeichnet, dass keine Ausgabe erzeugt wird.*

Die graphische Darstellung eines Moore-Automaten ist der eines Mealy-Automaten sehr ähnlich. Nur werden jetzt die Kanten nur noch mit dem betreffenden Eingabebuchstaben gekennzeichnet, dafür schreibt man an die Zustände zusätzlich das Symbol, das bei Erreichen des Zustandes erzeugt wird. Es ist nicht schwer, sich klar zu machen, dass man zu jedem Mealy-Automaten einen äquivalenten Moore-Automaten finden kann und das auch umgekehrt gilt. Äquivalent heißt dabei, dass beide Automaten auf dem gleichen Eingabe- und Ausgabealphabet operieren und für alle möglichen Eingaben $w \in \Sigma^*$ die gleiche Ausgabe $w' \in \Delta^*$ erzeugt wird. Die Zustandsmengen und Zustandsüberführungsfunktionen der beiden äquivalenten Automaten sind dabei im Allgemeinen verschieden. Es ist eine gute Übung, zu dem Mealy-Automaten aus Abbildung 27 einen äquivalenten Moore-Automaten zu konstruieren und *zu beweisen*, dass tatsächlich die beiden Automaten äquivalent sind. Allerdings sollte man sich dafür ein ziemlich großes Blatt Papier gönnen; die Zustandsmenge des Moore-Automaten wird vermutlich etwa doppelt so groß sein.

Das Wort „vermutlich“ im letzten Satz deutet es schon an: auch zwei Mealy-Automaten können im oben beschriebenen Sinn äquivalent sein, obwohl Zustandsmenge und Überführungsfunktionen verschieden sind. Das kann man sich ganz leicht klar machen: man braucht nur zu einem bestehenden Automaten Zustände (und entsprechende Kanten) hinzuzufügen; die neuen Zustände sind dann natürlich gar nicht erreichbar und können an der Ausgabe folglich nichts ändern. Das gilt natürlich genau so auch für Moore-Automaten. Wir sehen, dass die Verkleinerung von solchen Automaten ein interessantes Thema ist, dass wir aber aus Zeitgründen in eine einführende Vorlesung über theoretische Informatik verweisen wollen.

5.2 Synchrone Schaltwerke

Wir wollen zunächst versuchen, unsere doch recht konkreten Überlegungen zur NAND-Kippstufe (Abbildung 26) und die ja eher abstrakten Modellierungsansätze aus Abschnitt 5.1 miteinander zu verknüpfen. Die Funktion der NAND-Kippstufe haben wir ganz ausführlich charakterisiert. Wie können wir diese Charakterisierung mit Hilfe eines Automaten ausdrücken?
Als Zustandsmenge genügt uns $Q = \{q_0, q_1\}$, da unsere NAND-Kippstufe ja nur zwei Zustände haben kann: entweder ist $P = \overline{Q} = 0$ oder $P = \overline{Q} = 1$. Als Eingabe wählen wir sinnvollerweise die Belegung von R und S, also ist unser Eingabealphabet $\Sigma = \{01, 10, 11\}$; man beachte, dass wir „01“, „10“ und „11“ jeweils als nur *einen* Buchstaben auffassen. Als Ausgabe genügt uns der Wert von P, da sich ja der Wert von Q daraus ergibt. Wir könnten aber genau so gut auch $\Delta = \{01, 10\}$ als Ausgabealphabet wählen. Es ist im Grunde egal, ob wir einen Mealy- oder Moore-Automaten angeben. Wir entscheiden uns hier für einen Mealy-Automaten, den wir in Abbildung 28 angegeben haben.

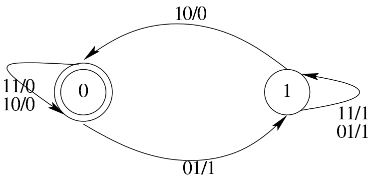


Abbildung 28: Mealy-Automat zur NAND-Kippstufe

Der von uns beschriebene Automat ist getaktet; das ist in der Definition des Mealy-Automaten so vorgeschrieben. Unsere NAND-Kippstufe (Abbildung 26, Seite 91) ist aber nicht getaktet. Das ändern wir jetzt. Wir führen einen zusätzlichen „Eingang“ T ein, der einen festen, von außen vorgegebenen Takt vorgibt. Es gibt verschiedene Möglichkeiten, das Taktsignal zur

Steuerung einzusetzen. Wenn jeweils geschaltet werden soll, wenn das Taktsignal eine 1 liefert, spricht man von Pegelsteuerung: der Pegel des Taktsignals steuert direkt die getaktete Schaltung. Alternativ kann man aber auch auf den Wechsel des Taktsignals achten; dann spricht man von Flankensteuerung. Ist der Wechsel vom Nullsignal zum Einssignal ausschlaggebend, nennt man das positive Flankensteuerung. Ist es der Wechsel vom Einssignal zum Nullsignal, so heißt das negative Flankensteuerung. Unserem Ansatz, uns nicht für technische Details unterhalb der digital-logischen Ebene zu interessieren, treu bleibend, wollen wir die konkrete Entscheidung bezüglich des Taktsignals und sich daraus ergebende Konsequenzen hier ignorieren.
Wir können jetzt sehr einfach die Eingangssignale R und S mit dem Taktsignal T verbinden und damit gleichzeitig auch in der Schaltung sicherstellen, dass niemals $R = S = 0$ versehentlich anliegt. Dazu führen wir für R und S jeweils ein weiteres NAND-Gatter ein, das als zweiten Eingang das Taktsignal erhält. Nur wenn die Werte von R und S stabil anliegen, schaltet das Taktsignal auf 1, so dass an unserer NAND-Kippstufe eine von $(1, 1)$ verschiedene Eingabe anliegen kann. Die komplette Schaltung haben wir noch einmal in Abbildung 29 wiedergegeben. Durch Verwendung der zusätzlichen NAND-Gatter vertauschen 0 und 1 die Rollen. Die nicht erlaubte Eingabe ist jetzt also $(R, S) = (1, 1)$. Man nennt die Schaltung R-S-Flip-Flop. Dabei ist die Bezeichnung „Flip-Flop“ der englischen Sprache entnommen; der Begriff bildet lautmalerisch einen Schalter mit zwei möglichen Stellungen nach.

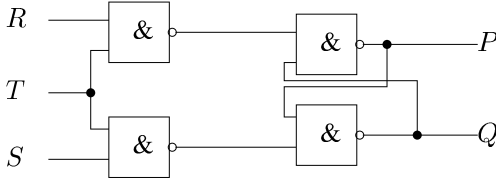


Abbildung 29: RS-Flip-Flop

Wir können natürlich weiterhin den Zustand des RS-Flip-Flops durch Anlegen geeigneter Eingaben auf 0 oder 1 festlegen. Die entsprechende Tabelle geben wir in Tabelle 17 an.

R	S	Q
0	0	Q
0	1	1
1	0	0
1	1	nicht erlaubt

Tabelle 17: Zustandtabelle des RS-Flip-Flop

Der Übergang von der ungetakteten NAND-Kippstufe zum RS-Flip-Flop ist wesentlich. Wir sind von einem ausgesprochen schwierig zu analysierenden und auch schwierig zu verstehendem ungetaktetem Schaltwerk, das ein komplexes Schwingungsverhalten haben kann, zu einem klar strukturierten und mit einem einfachen Mealy-Automaten gut und ausreichend exakt zu beschreibenden getakteten Schaltwerk gekommen. Wir beschäftigen uns darum von jetzt an nur noch mit solchen getakteten Schaltwerken, die wir aus nahe liegenden Gründen auch synchrone Schaltwerke nennen.
Wenn wir uns die tabellarische Darstellung des Verhaltens des RS-Flip-Flops (Tabelle 17) noch einmal ansehen, fällt uns vermutlich als erstes die Zeile „1 1 | nicht erlaubt" ins Auge. Es ist natürlich ausgesprochen unschön und unpraktisch, ein Bauteil zu verwenden, dass durch das Anlegen nicht zulässiger Eingaben in einen undefinierten Zustand gerät. Darum liegt es nahe, das RS-Flip-Flop, auf dessen Funktionalität wir natürlich nicht verzichten wollen, in geringfügig größere Schaltwerke einzubinden, bei denen dieses Problem dann nicht mehr besteht.
Eine Variante erzwingt $R = \overline{S}$, verwendet also nur noch ein einziges Eingangssignal D und erzeugt die Eingangssignale für das intern verwendete RS-Flip-Flop mit Hilfe einer Negation. Die entsprechende Schaltung ist in Abbildung 30 angegeben. Offenbar kann hier nicht mehr ein Wert gesetzt und beliebig lange gehalten werden. Stattdessen wird der Eingabewert zeitversetzt wiedergegeben. Daraus erklärt sich auch der Name D-Flip-Flop: der Buchstabe D steht für das englische Wort Delay, also Verzögerung.

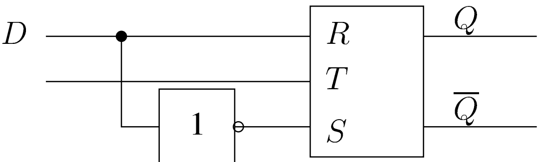


Abbildung 30: D-Flip-Flop

Eine andere sinnvolle Ergänzung des RS-Flip-Flops zu einem Schaltwerk, das für jede Eingabe sinnvolles Verhalten bietet, ist das so genannte JK-Flip-Flop, dessen Verhalten wir in Tabelle 19 wiedergeben. Man legt fest, dass die beim RS-Flip-Flip nicht erlaubte Eingabe 11 den aktuellen Speicherinhalt invertieren soll. Um zu einer Schaltung zu kommen, die das gewünschte Verhalten realisiert, machen wir uns mit Hilfe von Tabelle 18 klar, wie das Zusammenspiel von J, K, R, S, P und Q genau aussehen soll.
Wir sehen, dass wir das gewünschte Verhalten erreichen, wenn wir $R = J\,Q'$ und $S = K\,P'$ wählen. Die entsprechende Realisierung des JK-Flip-Flops ist in Abbildung 31 dargestellt.

J	K	R	S	P	Q
0	0	0	0	P'	Q'
0	1	0	1	1	0
1	0	1	0	0	1
1	1	P'	Q'	$\overline{P'}$	$\overline{Q'}$

Tabelle 18: Zusammenhang von J, K, R, S und Zustand beim JK-Flip-Flop

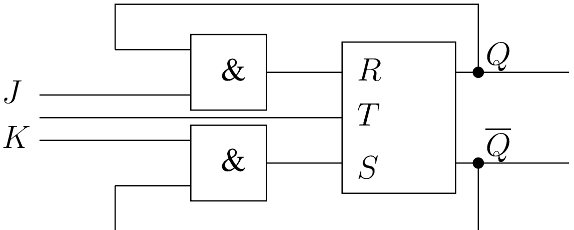


Abbildung 31: JK-Flip-Flop

J	K	Q
0	0	Q'
0	1	0
1	0	1
1	1	$\overline{Q'}$

Tabelle 19: Zustandtabelle des JK-Flip-Flops

Manchmal ist es besonders der einfache Zustandswechsel, der am JK-Flip-Flop so interessant ist. Es gibt darum noch einen weiteren nenneswerten Flip-Flop-Typ, der auf dem JK-Flip-Flop aufsetzt und entweder den Zustand unverändert beibehält oder ihn invertiert. Weil ein JK-Flip-Flop so ein Verhalten zeigt, wenn die beiden Eingaben gleich belegt sind, genügt uns hier ein Eingang und die Schaltung aus Abbildung 32 leistet das Gewünschte. Man bezeichnet dieses Flip-Flop als T-Flip-Flop, dabei steht „T“ für Toggle, als umschalten. Um Verwirrung zu vermeiden, haben wir in Abbildung 32 dann das Taktsignal in Clk (für Clock) umbenannt.

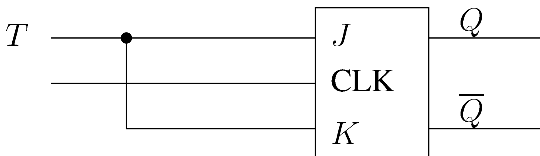


Abbildung 32: T-Flip-Flop

Für den Einsatz von Flip-Flops ist meist eine andere Darstellung sinnvoll. In der Regel ist man daran interessiert, den aktuellen Zustand des Flip-Flops auf kontrollierte Art und Weise zu ändern und fragt sich, wie dazu die Eingänge des Flip-Flops belegt werden müssen. Diese Information entnimmt man am einfachsten einer *Ansteuertabelle*, die genau diese Abhängigkeit zeigt. Wir sehen uns zunächst die Ansteuertabelle des D-Flip-Flops an (Tabelle 20, dabei bezeichnet im Folgenden stets Q den aktuellen Zustand und Q' den gewünschten neuen Zustand.

Q	Q'	D
0	0	0
0	1	1
1	0	0
1	1	1

Tabelle 20: Ansteuertabelle D-Flip-Flop

Wir sehen, dass die Ansteuerung des D-Flip-Flops besonders einfach ist. Es muss einfach der gewünschte neue Zustand als Eingabe angelegt werden, der Wert des alten Zustands spielt gar keine Rolle.
Für das RS-Flip-Flop ist die Lage schon etwas komplizierter, wie wir der Ansteuertabelle 21 entnehmen können. Wenn sich der aktuelle Zustand Q ändern soll, so gibt es auf jeden Fall nur eine Belegung der Steuerleitungen R und S, die das erreichen kann. Wenn aber der Zustand unverändert bleiben soll, so führen zwei verschiedene Belegungen der Steuerleitungen zum gewünschten Resultat: zum einen kann der Zustand explizit wie gewünscht gesetzt werden, zum anderen kann durch Wahl von $R = S = 0$ der Zustand unverändert belassen werden. Dieser Freiheitsgrad schlägt sich in der Ansteuertabelle 21 in zwei Don't-Care-Symbolen $*$ nieder. Wir wissen schon von der Diskussion unvollständig definierter Funktionen in Abschnitt 4.5, dass diese Wahlfreiheit manchmal zum Entwurf einfacherer Schaltungen verhelfen kann. Wir haben dort auch schon geeignete Strategien besprochen, um für solche Funktionen Minimalpolynome zu finden.

Q	Q'	R	S
0	0	$*$	0
0	1	0	1
1	0	1	0
1	1	0	$*$

Tabelle 21: Ansteuertabelle RS-Flip-Flop

Für das JK-Flip-Flip ist die Lage in gewisser Weise sehr ähnlich. Weil wir zusätzlich mit der Belegung $J = K = 1$ den aktuellen Zustand invertieren können, haben wir in der Ansteuertabelle des JK-Flip-Flips (Tabelle 22) noch zwei Don't-Care-Symbole mehr. Dieser relativ große Freiheitsgrad macht JK-Flip-Flops bei der Realisierung von Schaltwerken oft zu einer ausgesprochen angenehmen Wahl.

Q	Q'	J	K
0	0	0	$*$
0	1	1	$*$
1	0	$*$	1
1	1	$*$	0

Tabelle 22: Ansteuertabelle JK-Flip-Flop

Entwurf von Schaltwerken

Flip-Flops haben wir uns natürlich nicht nur aus Spaß angesehen und auch die Einführung der beiden Automatentypen (Mealy- und Moore-Automat) hatte nicht nur die Absicht, ein RS-Flip-Flop zu entwerfen. Tatsächlich haben wir jetzt alle notwendigen Arbeitsmittel und Methoden zusammen, um Schaltwerke für alles zu entwerfen, was wir in Form eines Mealy- oder Moore-Automaten formalisieren können. Wir werden sehen, dass nun unser Wissen über den Entwurf von Schaltnetzen und die Realisierung von 1-Bit-Speichern mit Hilfe von Flip-Flops dabei zusammenfließen und sich in ihrer Gesamtheit als mächtige Werkzeuge erweisen werden.

Grundsätzlich kann ein Schaltwerk wie in Abbildung 33 dargestellt werden. Man erkennt zwei deutlich getrennte Teile. Zum einen gibt es ein Schaltnetz, das als Eingabe zum einen die eigentlichen Eingaben des Schaltwerkes $x_1, \ldots, x_n$ erhält, andererseits aber auch weitere Eingaben $w_1, \ldots, w_l$ aus dem Speicher. Dieser Speicher stellt die zweite große Komponente des Schaltwerkes dar. Er wird gesteuert von einem Teil der Ausgänge des Schaltnetzes $(y_1, \ldots, y_k)$. Außerdem berechnet das Schaltnetz natürlich auch noch die eigentliche Ausgabe des Schaltwerkes $z_1, \ldots, z_m$.

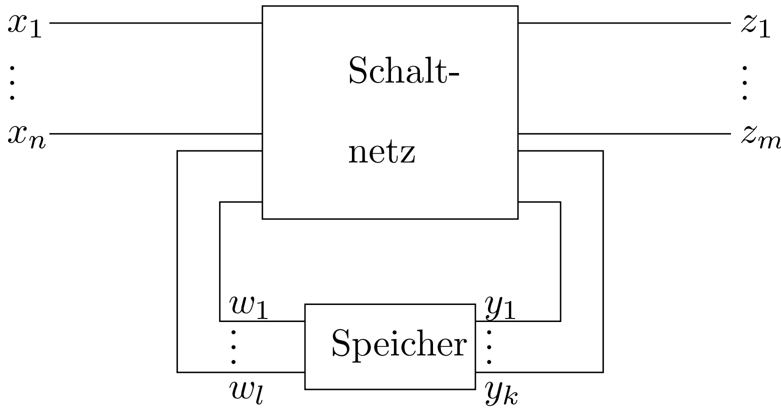


Abbildung 33: Modell eines Schaltwerks

Man kann dieses Modell noch etwas präziser fassen, wie das in Abbildung 34 geschieht. Man trennt das Schaltnetz in zwei Teile, einen Teil der für die Steuerung des Speichers zuständig ist, der zweite Teil berechnet die Ausgabe. Die eigentliche Eingabe des Schaltwerkes $x_1, \ldots, x_n$ dient sowohl dem Ansteuerschaltnetz als auch dem Ausgabeschaltnetz als Eingabe. Beide erhalten ebenfalls zusätzlich als Eingabe den Ausgang des Speichers $w_1, \ldots, w_l$. Die Steuersignale für den Speicher $y_1, \ldots, y_k$ werden ausschließlich vom Ansteuerschaltnetz berechnet, die eigentliche Ausgabe des Schaltwerkes $z_1, \ldots, z_m$ wird ausschließlich vom Ausgabeschaltnetz berechnet.

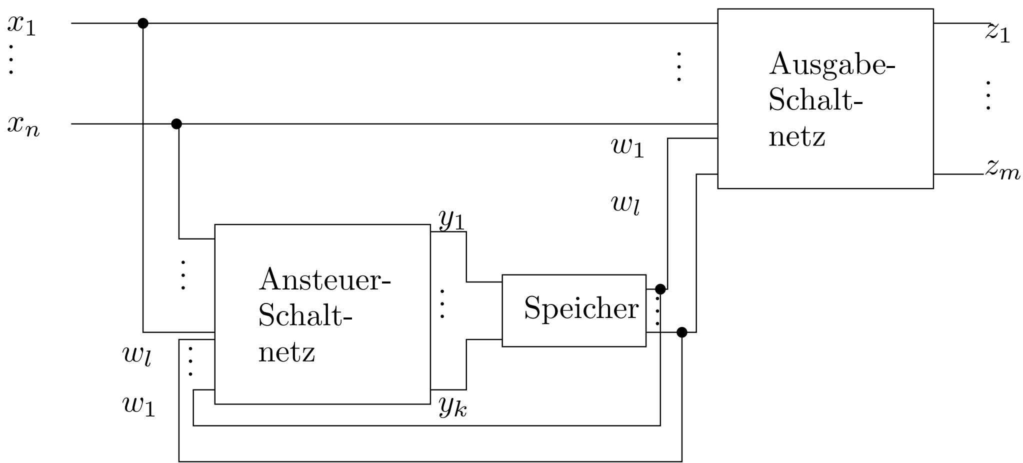


Abbildung 34: Detaillierteres Modell eines Schaltwerks

Die Korrespondenz zwischen Automaten und diesem Schaltwerk-Modell ist vielleicht nicht ganz offensichtlich. Die verschiedenen Zustände des Automaten werden durch den Speicher modelliert, der sich ja auch in verschiedenen

Zuständen befinden kann; jede Belegung des Speichers mit Bit-Werten fassen wir als einen Speicherzustand auf. Wir werden also eine Beziehung (eine bijektive Abbildung) zwischen den Zuständen des Automaten und (einigen) Zuständen des Speichers des Schaltwerkes definieren. Der Speicher muss also mindestens so viele verschiedene Zustände annehmen können (also mindestens so viele verschiedene Werte speichern können), wie der zu realisierende Automat Zustände hat. Wir wissen, dass wir mit b 1-Bit-Zuständen 2^b verschiedene Werte speichern können, wir brauchen folglich $2^b \geq |Q|$. Es genügt also, $b = \lceil \log_2 |Q| \rceil$ zu wählen, da ja $2^{\lceil \log_2 |Q| \rceil} \geq |Q|$ gilt. Wenn das Eingabealphabet und das Ausgabealphabet nicht schon binärcodiert vorliegt, muss auch hier auf analoge Art und Weise eine Binärcodierung erfolgen. Allerdings muss man nicht zwingend so sparsam codieren. Es ist möglich, dass die Verwendung von größere Alphabeten und größeren Speichern zur wesentlich kleineren Schaltwerken führt.
Der Entwurf von komplexen Schaltwerken, die komplexe Automaten realisieren, ist vermutlich nicht vollständig formalisierbar. Es gibt jedenfalls keine befriedigende mathematische Durchdringung, die ein rein schematisches oder gar automatisiertes Vorgehen erlaubt. Darum spielen nach wie vor beim Schaltwerkentwurf Erfahrung, intuitive Einsicht, Gefühl und Heuristik eine Rolle. Allerdings kann man natürlich das grundsätzliche Vorgehen vernünftig beschreiben und Daumenregeln nennen. Wir beginnen mit einer solchen grundsätzlichen Darstellung und werden dann zunächst ein recht kleines Beispiel besprechen, um am konkreten Objekt besser zu verstehen, was gemeint ist und wie man vorgeht.
Vor dem eigentlichen Beginn des Entwurfs des Schaltwerks steht das Verstehen der zu lösenden Aufgabe, ein in der Praxis nicht zu unterschätzendes Problem. Dieses Verstehen mündet schließlich im Entwurf eines Mealy- oder Moore-Automaten, der die Lösung des Problems formalisiert und exakt beschreibt. Ist dieser Schritt gelungen, muss der Automat schrittweise in ein Schaltwerk übersetzt werden. Dazu schreibt man zunächst in Form einer Tabelle auf, wie Ausgaben und neue Zustände von Eingaben und aktuellen Zuständen abhängen. Weil wir die Zustände schließlich als Betragszahlen in Binärdarstellung repräsentieren werden, ist es hilfreich, diese Zuordnung schon jetzt vorzunehmen. Dabei müssen wir wie schon diskutiert mindestens $\lceil \log_2 |Q| \rceil$ Bits verwenden. Allerdings ist es wie erwähnt gut möglich, dass die Wahl von mehr Bits zur Zustandscodierung am Ende in einen wesentlich kleineren Schaltwerk-Entwurf mündet. An dieser Stelle sind Intuition und Erfahrung gefragt. Jetzt müssen wir uns entscheiden, mit welchen Flip-Flop-Typen wir den Speicher realisieren wollen. Manchmal bieten sich gewisse Typen an, etwa ein D-Flip-Flop, wenn einfach direkt eine Ausgabe gespeichert werden soll oder ein T-Flip-Flop, wenn der aktuelle Zustand oft invertiert

werden muss. Bietet sich kein Flip-Flop-Typ direkt an, sind JK-Flip-Flops oft eine gute Wahl, weil, wie wir bereits gesehen haben, ihre Ansteuertabelle besonders viele Don't-Care-Stellen enthält, was bei der Realisierung mehr Freiheiten lässt, die oft einen dann letztlich einfacheren Entwurf erreichbar machen. Es ist natürlich möglich, verschiedene Flip-Flop-Typen in einem Schaltwerk gleichzeitig zu verwenden. Nach der Wahl der Flip-Flop-Typen soll man die vorher erstellte Tabelle um die Ansteuersignale der Flip-Flops erweitern. Jetzt ergibt sich also eine vollständige Wertetabelle einer (unter Umständen unvollständig definierten) booleschen Funktion. Diese boolesche Funktion muss jetzt in Form eines Schaltnetzes realisiert werden. Dabei kommen natürlich die Methoden, die wir für den Schaltnetzentwurf besprochen haben, zum Einsatz. Weil sich in der Regel (zumindest auf den ersten Blick) schlecht strukturierte Funktionen ergeben, denen man keine besonderen Eigenschaften ansieht, bieten sich eher die Methoden für den Schaltnetzentwurf allgemeiner boolscher Funktionen an, also vor allem die Berechnung von Minimalpolynomen mit KV-Diagrammen bzw. dem Algorithmus von Quine/McCluskey und dem Einsatz der PI-Tafel. Weil wir in aller Regel boolesche Funktionen mit vielen Ausgängen realisieren müssen, ist auf die Verwendung von Primimplikanten in mehreren Disjunktionen zu achten, was das Überdeckungsproblem schon erheblich ändert. Dann können schließlich das Schaltnetz und die gewählten Flip-Flops passend miteinander verbunden werden, so dass wir am Ende ein vollständiges Schaltwerk haben.

Die Ausführungen zum Schaltwerkentwurf waren lang und abstrakt. Wir konkretisieren sie jetzt und betrachten ein einfaches und eher kleines Beispiel. Dazu kommen wir auf ein Thema zurück, mit dem wir uns schon ausgiebig beschäftigt haben und das wir darum auch gut verstanden haben. Das erleichtert das Kennenlernen des Neuen, dem wir im Schaltwerkentwurf begegnen. Wir haben gesehen (Kapitel 4.4), dass wir die Addition von zwei Betragszahlen in Binärdarstellung der Länge n nach der Schulmethode in Größe etwa $5n$ und Tiefe etwa $4n$ realisieren können. Mit dem Entwurf des Carry-Look-Ahead-Addierers haben wir die Tiefe auf etwa $2 \log n$ reduzieren können, allerdings haben wir dabei die Größe auf etwa n^2 vergrößert. Jetzt wollen wir dieses Problem mit einem Schaltwerk lösen, dabei sollen die Bits der beiden Summanden schrittweise eingegeben und die Summe schrittweise ausgegeben werden. Wir sehen sofort, dass das nur vernünftig funktionieren kann, wenn wir die Summanden von rechts nach links stellenweise eingeben und die Summe auch in dieser Schreibrichtung ausgeben lassen.

Um ein Bit der Summe berechnen zu können, brauchen wir die beiden entsprechenden Bits der beiden Summanden und das Übertragsbit, das eine Stelle vorher entstanden ist. Es ist vernünftig, in jedem Schritt gleiche beide Summandenbits einzulesen und dann auch direkt das Summenbit der

Position auszugeben. Damit das auch funktioniert und ein eventuell vorne auftretendes Übertragsbits ausgegeben werden kann, muss als letztes (also am weitesten links gelegenes) Bit der beiden Summanden ein Nullbit eingegeben werden.

Als Eingabealphabet haben wir also $\Sigma = \{00, 01, 10, 11\}$, wobei zu beachten ist, dass es sich dabei um genau vier Zeichen handelt, 00 zum Beispiel also als ein Eingabesymbol zu interpretieren ist, das freilich zwei Eingabebits codiert. Als Ausgabealphabet ergibt sich dann $\Delta = \{0, 1\}$. Speichern müssen wir nur das Übertragsbit, das ja dann bei der nächsten Stelle zur Summe beiträgt. Als Zustandsmenge genügt uns also $Q = \{0, 1\}$, wobei der Startzustand $q_0 = 0$ ist; anfangs gibt es keinen Übertrag, was einem Carrybit mit Wert 0 entspricht. Die Zustandsüberführungsfunktion δ und die Ausgabefunktion λ können wir bei diesem kleinen Automaten tatsächlich explizit in Tabellenform angeben (Tabellen 23 und 24). Wir geben aber wie gewohnt den Automaten auch in Form eines Diagramms an (Abbildung 35).

q	w	$\delta(q, w)$
0	00	0
0	01	0
0	10	0
0	11	1
1	00	0
1	01	1
1	10	1
1	11	1

Tabelle 23: Zustandsüberführungsfunktion für den Serienaddierer

q	w	$\lambda(q, w)$
0	00	0
0	01	1
0	10	1
0	11	0
1	00	1
1	01	0
1	10	0
1	11	1

Tabelle 24: Ausgabefunktion für den Serienaddierer

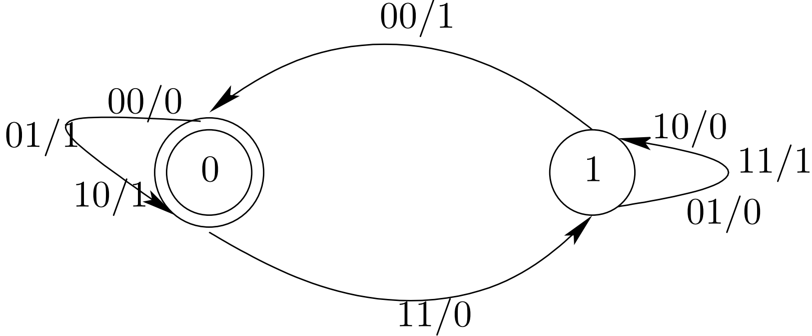


Abbildung 35: Mealy-Automat des Serienaddierers

Das Ausgabealphabet Δ ist schon binärcodiert, auch die Zustandsmenge Q kann man so auffassen. Für das Eingabealphabet bietet sich eine triviale Übertragung in zwei Bits offensichtlich an. Wir können darum leicht die jetzt benötigte Tabelle angeben, die den Zusammenhang zwischen Eingaben und aktuellem Zustand auf der einen Seite und Ausgaben und neuem Zustand auf der anderen Seite exakt darstellt. Tabelle 25 leistet das.

c_{alt}	x	y	c_{neu}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Tabelle 25: Eingabe-Zustands-Tabelle des Serienaddierers

Zur Realisierung des Speichers genügt ein einziges Flip-Flop; für den genauen Typ müssen wir uns jetzt entscheiden. Wir wählen ein JK-Flip-Flop, eine Wahl die auch anders hätte getroffen werden können. Als Begründung können wir anführen, dass Tabelle 25 jedenfalls keine andere Wahl unmittelbar nahe legt, wir bei JK-Flip-Flops besonders viele Freiheiten bei der Wahl der Ansteuerfunktion haben und darum auf ein besonders einfaches Ansteuerschaltnetz hoffen dürfen. Wir schauen uns noch einmal die Ansteuertabelle des JK-Flip-Flop an (Abbildung 22) und ergänzen dann Tabelle 25 entsprechend. Das Ergebnis ist in Tabelle 26 dargestellt.

c_{alt}	x	y	c_{neu}	s	J	K
0	0	0	0	0	0	*
0	0	1	0	1	0	*
0	1	0	0	1	0	*
0	1	1	1	0	1	*
1	0	0	0	1	*	1
1	0	1	1	0	*	0
1	1	0	1	0	*	0
1	1	1	1	1	*	0

Tabelle 26: Eingabe-Zustands-Tabelle des Serienaddierers, um Flip-Flop-Ansteuerung erweitert

Jetzt sind wir schon beinahe am Ziel. Wir müssen nun für eine Funktion $f\colon B^3 \to B^3$, die in Form einer Wertetabelle gegeben ist, ein Schaltnetz entwerfen. Die Funktion hängt von den Eingaben x, y und dem aktuellen Zustand c_{alt} ab, ihr Ausgangswert $f(x, y, c_{\text{alt}}) \in B^3$ beschreibt gleichzeitig den Ausgangswert s und die Ansteuerung des JK-Flip-Flops mit den Werten für J und K. Eine zunächst äquivalente Beschreibung spricht natürlich von drei Funktionen $f\colon B^3 \to B$, alle drei von den gleichen Variablen abhängend, jeweils eine für s, J und K. Man darf aber nicht vergessen, dass diese drei Funktionen in *einen* Schaltnetz realisiert werden, also sich Gatter teilen können, die dadurch selbstverständlich nicht mehrfach realisiert werden müssen. Dadurch kann ein Schaltnetz für die drei Funktionen zusammen (bzw. die eine Funktion $f\colon B^3 \to B^3$) kleiner werden als drei getrennte Schaltnetze für die drei „kleineren“ Funktionen.

Wir beginnen mit den Funktionen für die Ansteuerung des JK-Flip-Flops und sehen uns als erstes die Funktion für J an. Wir erstellen ein KV-Diagramm (Tabelle 27) und erkennen, dass wir durch die vielen Don’t-Care-Werte erhebliche Wahlfreiheit haben. Das kürzeste zulässige Monom, dass die einzige 1 im Diagramm überdeckt, ist $x\,y$. Weil wir keine weiteren Einsen zu überdecken haben, brauchen wir nicht weiter nach Primimplikanten zu suchen und haben $J(c_{\text{alt}}, x, y) = x\,y$ als boolesche Funktion.

c_{alt} \ xy	00	01	11	10
0			1	
1	*	*	*	*

Tabelle 27: KV-Diagramm für Ansteuerfunktion J

Das KV-Diagramm zur Ansteuerfunktion für K sieht sehr ähnlich aus (Tabelle 28). Analog wählen nur das Monom $\overline{x}\,\overline{y}$ und erhalten $K(c_{\text{alt}}, x, y) = \overline{x}\,\overline{y}$ als boolesche Funktion.

c_{alt} \ xy	00	01	11	10
0	*	*	*	*
1	1			

Tabelle 28: KV-Diagramm für Ansteuerfunktion K

Ein Blick auf das KV-Diagramm für die Ausgabefunktion s (Tabelle 29) lässt uns erkennen, dass wir es mit einem schlechtesten Fall zu tun haben. Das Minimalpolynom für s ist gleich der disjunktiven Normalform (DNF), wir müssen für jede der vier Einsen einen Minterm benutzen, um sie abzudecken. Wir erhalten folglich $s(c_{\text{alt}}, x, y) = \overline{c_{\text{alt}}}\,\overline{x}\,y \vee \overline{c_{\text{alt}}}\,x\,\overline{y} \vee c_{\text{alt}}\,\overline{x}\,\overline{y} \vee c_{\text{alt}}\,x\,y$. Man sieht direkt, dass man auf verschiedene Arten ausklammern könnte und wir entscheiden uns dafür, sowohl c_{alt} als auch $\overline{c_{\text{alt}}}$ auszuklammern. Das führt uns zu $s(c_{\text{alt}}, x, y) = \overline{c_{\text{alt}}}\,(\overline{x}\,y \vee x\,\overline{y}) \vee c_{\text{alt}}\,(\overline{x}\,\overline{y} \vee x\,y)$ und der Zweck dieses Ausklammerns wird jetzt unmittelbar erkennbar. Wir müssen sowohl $\overline{x}\,\overline{y}$ als auch $x\,y$ sowieso berechnen, da sie für die Ansteuerung des JK-Flip-Flops benötigt werden. Wir können also tatsächlich Gatter einsparen.

c_{alt} \ xy	00	01	11	10
0		1		1
1	1		1	

Tabelle 29: KV-Diagramm für Ausgabefunktion s

Die Funktionen sind alle so einfach, dass wir sie direkt in ein Schaltnetz überführen können, wie man es in Abbildung 36 sehen kann. Wir behandeln hier den aktuellen Zustand c_{alt} formal als weitere Eingabe, wie wir das bei der Bestimmung der Funktion ja auch getan haben. Wenn wir dann im letzten Schritt das Schaltwerk zusammensetzen, entfällt dieser Eingang. Stattdessen entnehmen wir den aktuellen Zustand c_{alt} aus dem JK-Flip-Flop; die Funktionen J und K, die wir im Schaltnetz formal als zusätzliche Ausgaben dargestellt haben, verwenden wir jetzt wie vorgesehen für die Ansteuerung des Flip-Flops. Da ein JK-Flip-Flop neben dem aktuellen Zustand Q auch direkt das Komplement $\overline{Q}$ zur Verfügung stellt, können wir im Vergleich zum Schaltnetz noch ein Negationsgatter einsparen.

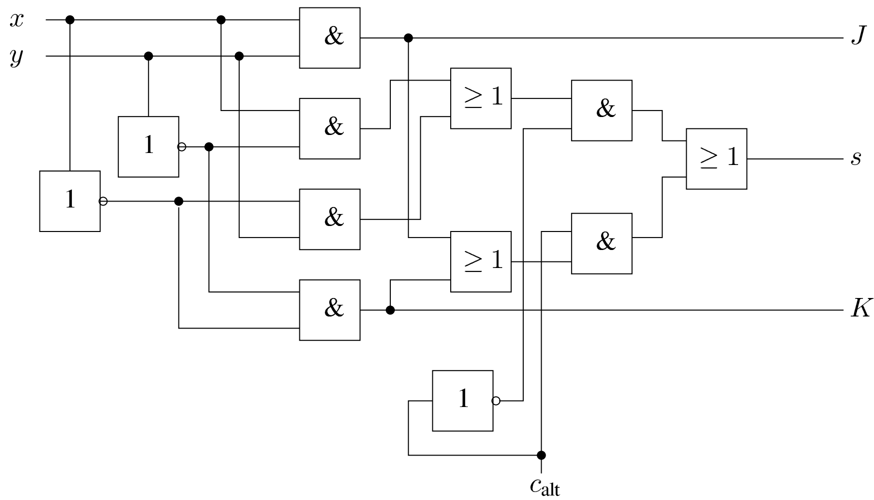


Abbildung 36: Schaltnetz für Ansteuerung und Ausgabe des Serienaddierers

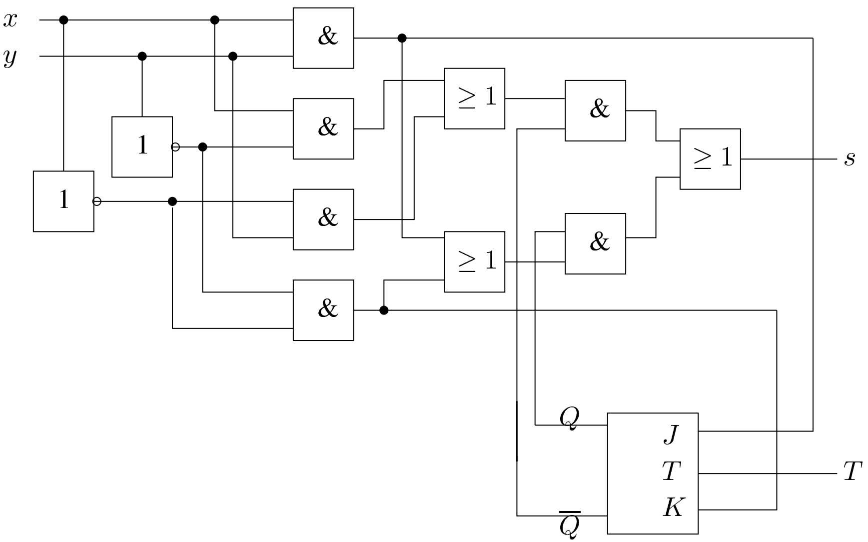


Abbildung 37: Schaltwerk Serienaddierer

Was haben wir jetzt eigentlich mit der Realisierung des Serienaddierers als Schaltwerk (Abbildung 37) erreicht? Wir haben eine kleine, kompakte und nicht sehr tiefe Schaltung, die beliebig lange Betragszahlen in Binärdarstellung addieren kann. Das klingt natürlich sehr beeindruckend. Allerdings sollte man bedenken, dass man in jedem Takt nur je ein Bit der beiden Summanden einspeisen kann und darum n Takte auf die Ausgabe warten muss. Das bewegt sich in der gleichen Größenordnung wie unser Schaltnetz, das wir nach der Methode der Schuladdition entworfen haben – schon dort haben wir erwähnt,

dass Tiefen (also Rechenzeiten), die linear in der Anzahl der Eingabevariablen sind, absolut inakzeptabel in Schaltungen sind, wenn die Anzahl der Eingabevariablen nicht sehr klein ist.
Wir wollen bei beiden Themen noch etwas verweilen, also sowohl beim Schaltwerkentwurf als auch bei der Addition. Wir wissen alle sehr genau, dass das eigentliche Problem bei der Realisierung einer schnellen Addition die Übertragsbits sind. Wenn ein Carry von ganz hinten nach ganz vorne durchgereicht werden muss, dann liegt der Verdacht nahe, dass das lange dauert – man möchte naiv vielleicht glauben, dass die Zeitspanne, die das dauert, sich proportional zu der Anzahl von Stellen verhält. Wir wissen schon, dass das ganz und gar nicht stimmt: der Carry-Look-Ahead-Addierer bewältigt das Problem in dramatisch kürzerer Zeit, nur proportional zum Logarithmus der Anzahl Stellen. Wir wollen das Problem hier trotzdem noch einmal etwas naiver angehen. Es kann sein, dass ein Carrybit von ganz hinten nach ganz vorne durchgereicht werden muss. Wenn ein Verfahren die Übertragsbits stellenweise nach vorne schiebt, wie das bei der Schulmethode und beim Serienaddierer der Fall ist, müssen solche Eingaben natürlich lange Bearbeitungszeit in Anspruch nehmen. Aber selbstverständlich gibt es viele Zahlen, bei deren Addition das nicht so ist. Könnte man in solchen Fällen nicht vielleicht schon vorher ein Ergebnis bereit stellen? Wir wollen diese Idee beim Entwurf eines Schaltwerkes für die Addition verfolgen. Es ist klar, dass wir die Eingabe (also die beiden Summanden) einem solchen Schaltwerk gleich am Anfang der Berechnung zur Verfügung stellen müssen, schließlich kann die Summe nur dann vollständig berechnen, wenn man die Summanden vollständig kennt. Weil wir bei Automaten immer endliche Alphabete voraussetzen, rücken wir von dem nicht mehr realisierbaren Wunsch ab, ein Addierwerk für beliebig lange Zahlen zu konstruieren, wie es der Serienaddierer ist. Wir geben uns stattdessen mit der Addition von Betragszahlen in Binärdarstellung fester Länge zufrieden. Bei Hardware-Realisierungen ist das ja auch keine ungewöhnliche Einschränkung. Die Eingabe wird also im ersten Takt eingegeben, danach folgt eine mehr oder weniger lange Berechnung, deren Länge – das hatten wir vereinbart – variabel von der konkreten Eingabe abhängen darf. Das passt nun leider nicht gut zu unseren Automaten-Modellen. Mealy- und Moore-Automat lesen in jedem Takt ein Symbol der Eingabe und stellen die Arbeit ein. Das passt nicht zu unseren Wünschen, wir müssen darum die Definition der Automaten ein wenig ändern. Wir hatten schon immer bei der Ausgabe ein zusätzliches Symbol ε erlaubt, das anzeigt, dass eigentlich keine Ausgabe geschrieben wird. Nun erlauben wir die Verwendung dieses Symbols ε auch in der Eingabe und verlangen, dass dieses Symbol so lange eingegeben wird, bis der Automat signalisiert, dass die Berechnung beendet ist. Dann darf eine neue echte Eingabe angelegt werden. Das Signal, dass der Automat die

Berechnung beendet hat, realisieren wir als ein zusätzliches Ausgabebit, dass den Wert 0 hat, so lange der Automat noch in der Berechnungsphase ist und den Wert 1 annimmt, wenn der Automat bereit ist, eine echte Eingabe entgegenzunehmen.

Jetzt müssen wir konkretisieren, wie die Berechnung eigentlich durchgeführt werden soll. Wir haben zwei Summanden $x = (x_{n-1}x_{n-2}\cdots x_0)_2$ und $y = (y_{n-1}y_{n-2}\cdots y_0)_2$, die wir uns passend stellenweise untereinander geschrieben vorstellen können. Wir wollen in jedem Takt die beiden Zahlen x, y durch zwei andere Zahlen x', y' mit gleicher Summe ersetzen, es soll also $x + y = x' + y'$ gelten. Dann ist zumindest sichergestellt, dass keine wesentlichen Informationen verloren gegangen sind. Am Ende soll die untere Zahl y den Wert 0 haben; dann können wir an der oberen Zahl x die Summe ablesen. Wenn wir in einem Takt stellenweise jeweils einen Halbaddierer verwenden, der uns Summenbit s_i und Übertragsbit c_i bereitstellt, so wissen wir, dass $x_i + y_i = s_i + 2c_i$ gilt. Wir könnten also x_i durch s_i ersetzen und y_{i+1} durch c_i ersetzen. Wenn wir zusätzlich $y_0 = 0$ setzen und einen eventuellen Übertrag in y_n (oder o für overflow) auffangen, so haben wir sicher noch nichts falsch gemacht. Tatsächlich haben wir sogar schon alles erreicht, was wir wollten: An der hintersten Stelle setzen wir auf jeden Fall $y_0 = 0$. Im nächsten Takt kann darum an der Stelle 0 bei Addition von x_0 und $y_0 = 0$ mit dem Halbaddierer kein Übertrag entstehen, so dass wir nach zwei Takten $y_1 = y_0 = 0$ garantieren können. Aus gleichem Grund haben wir mit Sicherheit $y_2 = y_1 = y_0 = 0$ nach drei Takten; induktiv schließen wir (und könnten es notfalls auch durch vollständige Induktion beweisen), dass nach i Takten $y_{i-1} = y_{i-2} = \cdots = y_1 = y_0$ gilt. Folglich ist die Rechnung nach spätestens $n+1$ Takten beendet. Wir hatten uns überlegt, dass wir in jedem Takt damit rechnen müssen, einen Übertrag in einem Überlaufbit o speichern zu müssen. Bedeutet das, dass wir insgesamt n solche Überlaufsbits brauchen, um für alle Fälle gerüstet zu sein? Das ist offensichtlich nicht der Fall. Wir erhalten als Eingabe zwei Betragszahlen mit jeweils n Bits, es kann also sicher nicht mehr als $2^n - 1 + 2^n - 1 = 2^{n+1} - 2$ als Ergebnis herauskommen. Das ist aber sicher in $n + 1$ Bits korrekt darstellbar, folglich kommen wir mit einem einzigen Überlaufbit o aus. Eingangs hatten wir angesprochen, dass wir durch ein Zustandsbit d anzeigen wollen, ob die Rechnung beendet ist. Offenbar können wir $d = 1$ setzen, wenn alle Bits von y den Wert 0 haben.

Im Grunde haben wir jetzt das Addierwerk vollständig beschrieben. Der Fairness halber sollten wir erwähnen, dass wir zwar die wesentlichen Ideen Schritt für Schritt erarbeitet haben, wir aber bei weitem nicht die Ersten sind, denen so ein Addierwerk einfällt. Das beschriebene Addierwerk ist unter dem Namen von Neumann-Addierwerk bekannt. John von Neumann (oder auch Johann von Neumann, 1903–1957, ein amerikanischer Mathematiker

österreichisch-ungarischen Ursprungs) hat viel zur Entwicklung der Mathematik, Wahrscheinlichkeitsrechnung und Logik beigetragen. Da er zu Zeiten lebte und arbeitete, als vor allem die theoretischen Grundlagen der Informatik gelegt wurden, ist er eine für die Informatik wichtige historische Persönlichkeit.

Um alle Unklarheiten zu vermeiden, geben wir ein Prinzip-Schaltwerk des von Neumann-Addierers für drei Bits an (Abbildung 38). Wir bezeichnen mit HA den bekannten Halbaddierer, der zwei Eingänge hat und am Ausgang Summenbit s und Übertragsbit c zur Verfügung stellt. Für die zu speichernden Bits malen wir jeweils ein Kästchen mit einem Eingang und einem Ausgang. Am Ausgang kann der gespeicherte Wert ausgelesen werden, am Eingang ein neuer Wert eingeschrieben. Die wenigen bleibenden logischen Zusammenhänge stellen wir mit Hilfe der bekannten Gatter da. Um Ein- und Ausgänge eindeutig zu kennzeichnen, verwenden wir Pfeile (also gerichtete Kanten), um die Richtung des Datenflusses eindeutig darzustellen.

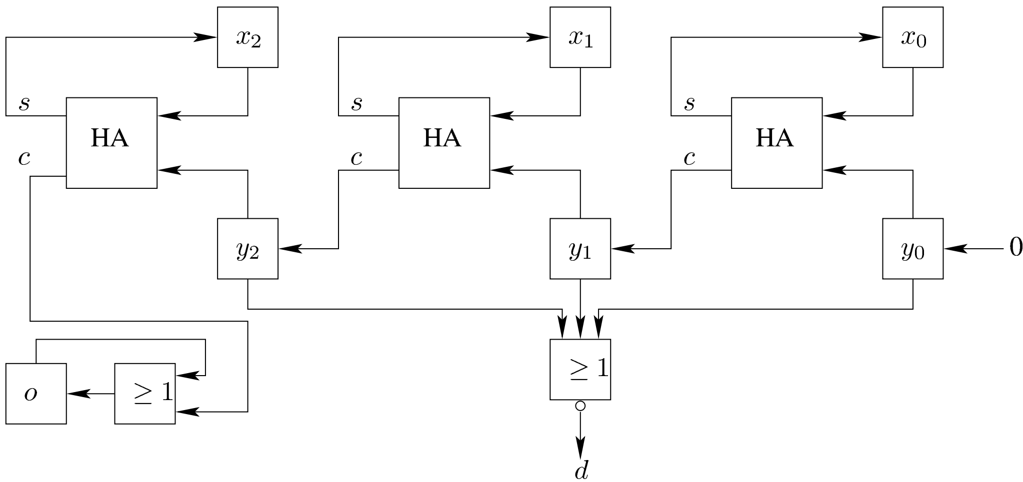


Abbildung 38: Von Neumann-Addierwerk

Was müssen wir ändern, wenn wir aus der Prinzip-Schaltung 38 ein reales Schaltbild konstruieren wollen? Bei allen „Speicherkästchen" ist es so, dass der neue Wert unabhängig vom aktuell gespeicherten Wert eingeschrieben werden kann. Das gilt sogar für das Übertragsbit, was wir durch die Oder-Verknüpfung von aktuellem Zustand und neuem Wert erreicht haben. Darum können wir jedes einzelne dieser „Speicherkästchen" durch ein einfaches D-Flip-Flop ersetzen; natürlich müssen alle diese Flip-Flops gleich getaktet sein. Weitere Änderungen sind nicht erforderlich, so dass wir direkt aus Abbildung 38 ein von Neumann-Addierwerk für Drei-Bit-Zahlen erhalten. Die

Verallgemeinerung auf längere (oder auch kürzere) Zahlen ist sehr nahe liegend, wir wollen das hier hier weiter ausführen.
Wir hatten ausgeführt, dass wir das Modell des Mealy-Automaten etwas erweitern, um eine Formalisierung als Mealy-Automat zu ermöglichen. Wir wollen uns jetzt noch die Mühe machen, diese Formalisierung für allgemeines $n \in \mathbb{N}$ auszuführen. Einerseits können wir daran den formalen Umgang mit Mealy-Automaten einüben, andererseits können wir uns noch einmal alle Details des von Neumann-Addierwerks vor Augen führen.
Eingaben sind zwei n-Bit-Zahlen, die wir als Bitstring $w \in B^{2n}$ darstellen können. Um nachher leichter ausdrücken zu können, was wir meinen, vereinbaren wir die Notation $w = xy$ mit $x = x_{n-1} \cdots x_0$ und $y = y_{n-1} \cdots y_0$ für alle $w \in B^{2n}$. Zusammen mit der leeren Eingabe, die wir ja brauchen, haben wir also $\Sigma = B^{2n} \cup \{\varepsilon\}$. Ausgeben wollen wir einerseits die Summe als n-Bit-Zahl und das Übertragsbit o, andererseits aber auch unser Statusbit d, das mit dem Wert 1 signalisiert, dass die Rechnung beendet ist. Wenn die Rechnung noch nicht beendet ist, kommen wir mit der Ausgabe des Statusbits $d = 0$ aus. Insgesamt genügt es uns also, $\Delta = B^{n+2} \cup \{0\}$ festzulegen. Wir vereinbaren, dass wir für $w \in \Delta$ die Notation $w = dox$ mit $x = x_{n-1} \cdots x_0$ verwenden. Das Addierwerk speichert die Eingabe der Länge $2n$ und das aktuelle Übertragsbit, wir kommen also mit $Q = B^{2n+1}$ genau aus. Für einen Zustand $q \in Q$ verwenden wir die Notation $q = oxy$, wieder mit $x = x_{n-1} \cdots x_0$ und $y = y_{n-1} \cdots y_0$. Startzustand ist $q_0 = 0^{2n+1}$, wir beginnen also mit den Zahlen 0 und ohne Übertrag.
Nachdem wir Eingabealphabet Σ, Ausgabealphabet Δ, Zustandsmenge Q und Startzustand q_0 beschrieben haben, müssen jetzt noch die Zustandsüberführungsfunktion δ und die Ausgabefunktion λ beschreiben. Das ist nicht ganz einfach, wir gehen darum schrittweise und systematisch vor. Das von Neumann-Addierwerk akzeptiert nur dann Eingaben, wenn für den aktuellen Zustand $q = oxy$ gerade $y = 0^n$ gilt. Wir wollen trotzdem das Verhalten vollständig definieren und legen fest, dass der Automat die aktuelle Rechnung dann abbricht und mit der neuen Eingabe neu startet. Wenn also $w = xy \neq \varepsilon$ die aktuelle Eingabe ist, dann legen wir $\delta(q, w) = 0xy$ als neuen Zustand fest. Für diesen Fall müssen wir entscheiden, ob wir erst eine Rechnung ausführen müssen, oder ob die Eingabe schon in Form eines Ergebnisses vorliegt. Es ist also $\lambda(q, w) = 0$, falls $y \neq 0^n$ ist und $\lambda(q, w) = 10x$ sonst. Das ist sinnvoll, denn wenn die Eingabe schon aus x und y mit $y = 0^n$ besteht, ist x direkt die korrekte Summe.
Jetzt müssen wir noch über den ja eigentlich interessanten Fall sprechen, dass die Eingabe $w = \varepsilon$ ist. Nun sind wir in der Rechenphase und müssen einen Rechenschritt durchführen. Wir legen $\delta(q, \varepsilon) = q'$ fest wie folgt. Es sei $q = oxy$ und $q' = o'x'y'$. Dann ist $x'_i = x_i \oplus y_i$ für alle $i \in \{0, 1, \ldots, n-1\}$ und $y'_i =$

$x_{i-1} \wedge y_{i-1}$ für $i \in \{1, \ldots, n-1\}$, außerdem $y'_0 = 0$. Wir erkennen natürlich die booleschen Funktionen wieder, die ein Halbaddierer realisiert. Außerdem ist $o' = o \vee x_{n-1} \wedge y_{n-1}$. Damit ist die Zustandsüberführungsfunktion schon komplett spezifiziert. Es fehlt noch die Ausgabefunktion λ für den Fall, dass die Eingabe $w = \varepsilon$ ist. Falls der aktuelle Zustand $q = oxy$ mit $y = 0^n$ ist, dann ist $\lambda(q, \varepsilon) = 1ox$, da die Rechnung dann ja zu Ende ist und das Ergebnis in x (und eventuell im Übertragsbit o) steht. Andernfalls ist $\delta(q, \varepsilon) = 0$.

Das von Neumann-Addierwerk ist sehr viel größer als der Serienaddierer, den wir vorher entwickelt haben. Trotzdem ist es unter Umständen nicht schneller als der Serienaddierer. Im ungünstigsten Fall (den man Worst Case nennt), brauchen beide Schaltwerke $n + 1$ Takte zur Addition von zwei n-Bit-Zahlen. Warum sollten wir also den größeren Hardware-Aufwand des von Neumann-Addierers in Kauf nehmen? Es gibt einen ganz wesentlichen Unterschied zwischen den beiden Addierwerken: für den Serieraddierer gibt es keine besonders ungünstige Eingabe; es gibt auch keine besonders günstige Eingabe. Völlig unabhängig von der konkreten Eingabe steht das Ergebnis der Addition von zwei n-Bit-Zahlen nach $n + 1$ Takten zur Verfügung. Das ist gänzlich anders beim von Neumann-Addierwerk. Hier wird im günstigsten Fall gar kein Takt benötigt, das Addierwerk erkennt dann unmittelbar, dass die Eingabe gleichzeitig das Ergebnis ist und signalisiert am Statussignal d, dass die Rechnung beendet ist. Den ungünstigsten Fall hatten wir bisher mit $n + 1$ Takten abgeschätzt mit dem Argument, dass ja sicher die Anzahl der Nullen im unteren Summanden y von rechts nach links um mindestens 1 zunimmt in jedem Takt. Aber gibt es wirklich eine Eingabe, bei der so viele Takte benötigt werden? Wir stellen einmal exemplarisch eine Folge von Summanden dar, wie sie bei der Rechnung im von Neumann-Addierwerk auftreten. Dabei bekommen wir nicht nur ein besseres Gefühl für das konkrete Funktionieren des von Neumann-Addierwerks, wir lernen auch etwas über ungünstigste Eingaben, also Eingaben, die besonders große Anzahl von Rechentakten erfordern.

	0	1	1	1	1	$\leadsto$	0	1	1	1	0	$\leadsto$	0	1	1	0	0
		0	0	0	1			0	0	1	0			0	1	0	0
$\leadsto$	0	1	0	0	0	$\leadsto$	1	0	0	0	0						
		1	0	0	0			0	0	0	0						

Wir sehen, dass wenn der Summand x den Wert $2^n - 1$ hat und der Summand y den Wert 1, sich das einzelne 1-Bit von y in jedem Takt eine Position weiter nach links schiebt, bis es schließlich nach $n+1$ Takten im Übertragsspeicher o landet. Wir könnten jetzt also mit Hilfe der Methode der vollständigen Induktion auch formal beweisen, dass im Worst Case ein von Neumann-Addierwerk tatsächlich $n + 1$ Schritte rechnen muss. Ist das von Neumann-Addierwerk also tatsächlich nicht besser als der Serienaddierer? Wie wir gesehen haben,

kommt das auf die Eingaben an. Wenn wir immer nur $(2^n - 1) + 1$ rechnen wollen, können wir genau so gut einen Serienaddierer benutzen. Welche Additionen in der Praxis durchzuführen sind, lässt sich natürlich nicht gut vorhersagen. Eine übliche Methode der Informatik, in so einer Situation zu einer Einschätzung der Effizienz des Verfahrens zu kommen, ist eine so genannte Average Case Analyse. Der „Average Case", der durchschnittliche Fall, soll das typische Verhalten beschreiben, das durchschnittliche Verhalten, das Verhalten bei typischen oder durchschnittlichen Eingaben – oder das Verhalten im Durchschnitt betrachtet über eine große Anzahl von Eingaben. Natürlich muss man über die Eingaben immer noch irgend eine Annahme machen, sonst muss man ja wiederum damit rechnen, dass jede einzelne Eingabe gerade der schlechteste Fall ist. Die in solchen Situation typische Annahme ist die so genannte Gleichverteilung: man nimmt an, dass jede Eingabe mit gleicher Wahrscheinlichkeit auftritt. Eine praktisch äquivalente Aussage ist, dass über lange Zeit betrachtet jede mögliche Eingabe ungefähr gleich häufig auftritt. Im Einzelfall muss diese Annahme natürlich nicht zutreffen. Aber sie ist in vielen Situationen eine vernünftige Annahme.
Wenn wir das Verhalten unter dieser Annahme betrachten, welche durchschnittliche Taktzahl erwarten wir dann? Für eine formale und korrekte Analyse fehlen uns im Moment leider noch die wahrscheinlichkeitstheoretischen Möglichkeiten. Aber wir können anschaulich unserer Intuition folgend argumentieren und so zu einem Ergebnis kommen, das man dann auch formal als korrekt nachweisen kann.
Die Gleichverteilung entspricht der zufälligen, unabhängigen und gleichverteilten Wahl jedes einzelnen Eingabebits. Wenn man so vorgeht, dann erwartet man im Durchschnitt $n/2$ Einsen in beiden Summanden. Nach einem Takt hat man im unteren Summanden eine 1, wenn an den Positionen vor Beginn des Taktes in beiden Summanden eine 1 stand. Weil das mit Wahrscheinlichkeit $1/4$ der Fall ist (11 ist einer von vier gleich wahrscheinlichen Fällen), erwarten wir nach dem ersten Takt noch $n/4$ Einsen im unteren Summanden. Einen Takt weiter haben wir Einsen in y unter gleichen Bedingungen. Bei Einsen im oberen Summanden liegt die Lage etwas anders. Die Wahrscheinlichkeit für eine 1 oben nach einem Takt liegt an jeder Stelle bei $1/2$, da 01 und 10 eben zwei von vier gleich wahrscheinlichen Fällen sind. Darum erwarten wir nach dem zweiten Takt noch $n/8$ Einsen im unteren Summanden y, da im oberen Summanden x die Wahrscheinlichkeit für eine 1 unverändert bei $1/2$ liegt, während sie sich im unteren Summanden etwa halbiert hat. Die Argumentation setzt sich fort, so dass wir nach i Takten noch $n/2^{i+1}$ Einsen in y erwarten. Nach etwa $\log_2 n$ Takten erwarten wir also, dass unten keine Einsen mehr zu finden sind und die Rechnung des von Neumann-Addierers beendet wird. Tatsächlich kann man formal beweisen, dass $\log_2 n$ Takte recht

genau die durchschnittliche Taktzahl des von Neumann-Addierwerks bei rein zufälligen Eingaben ist. Wir haben jetzt also ein Addierwerk, dass weiterhin ziemlich klein ist, das zwar im ungünstigsten Fall sehr langsam addiert, das aber nicht nur im günstigsten Fall sehr schnell ist, sondern das auch im Durchschnitt schnell rechnet. Darum ist das von Neumann-Addierwerk mit seiner einfachen Struktur ein für die Praxis durchaus interessantes Schaltwerk.

5.3 Speicher

Wir haben in Abschnitt 5.2 verschiedene Flip-Flops kennen gelernt und uns schon überlegt, dass wir damit ein Bit speichern können. Natürlich kommt man in einem halbwegs vernünftigen Rechner nicht mit einem 1-Bit-Speicher aus. Es ist auch nicht akzeptabel, Bits nur zur Zustandsspeicherung zu verwenden und dann jeweils ad hoc einzuplanen. Was man sich wünscht, ist ein kompakter Baustein, der Speicher realisiert. Natürlich kann dieser Baustein aus Flip-Flops aufgebaut sein, er soll aber nach außen einfach die Funktionalität „Speicher" haben, also leicht mit Daten beschreibbar sein und die Daten müssen leicht wieder auslesbar sein.

Wir verdeutlichen uns das zunächst an einem „Baustein", der drei Bits speichern soll. Dabei fassen wir die drei Bits gedanklich zu einer Einheit, die wir *Wort* nennen, zusammen, sie werden also immer gemeinsam gelesen und geschrieben. Das ist prinzipiell realistisch; die Größe von *drei* Bits ist es natürlich nicht. Das ändert aber nichts an der Funktionsweise.

Wir brauchen für den Baustein drei Eingänge I_0, I_1 und I_2, an denen gleichzeitig die zu speichernden Werte der drei Bits anliegen, wenn wir schreiben wollen. Zum Auslesen werden dementsprechend drei Ausgänge benötigt, die wir O_0, O_1 und O_2 nennen. Außerdem müssen wir noch auswählen können, ob gerade geschrieben oder gelesen werden soll. Dazu sehen wir noch einen weiteren Eingang RD vor, der den Wert 1 haben soll, wenn ein Lesezugriff erfolgen soll und den Wert 0, wenn geschrieben werden soll. Eine mögliche Schaltung mit der gewünschten Funktionalität sieht man in Abbildung 39.

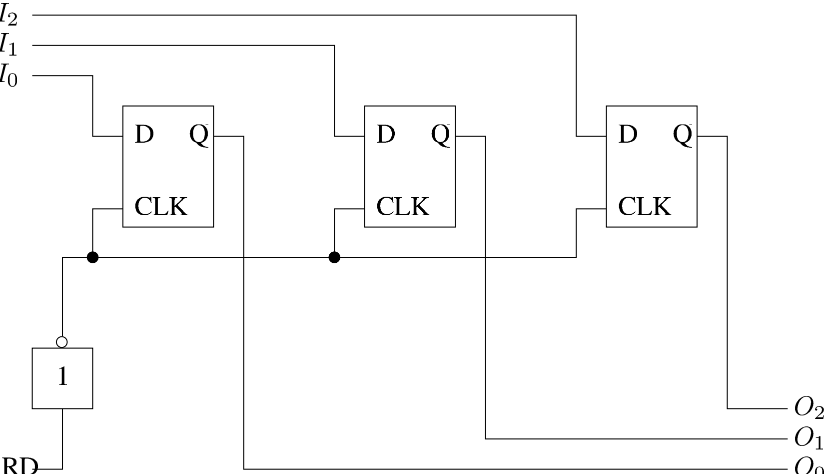


Abbildung 39: Speicher für ein Wort der Länge drei

Der Speicher ist einfach und direkt mit D-Flip-Flops realisiert. Man macht sich leicht klar, dass die gewünschte und oben beschriebene Funktionalität tatsächlich realisiert ist. Wir beobachten, dass wir die Takteingänge der D-Flip-Flops nutzen, um zu erreichen, dass der Speicherinhalt beim Schreiben gespeichert und beim Auslesen unverändert erhalten bleibt. Es ist auch ganz offensichtlich, wie dieser Ansatz auf realistische Wortgrößen (zum Beispiel 8 Bits je Wort) erweitert werden kann: es genügt einfach, die Anzahl der Ein- und Ausgaben als auch die Anzahl der D-Flip-Flops entsprechend zu vergrößern und die Ansteuerung jeweils im Prinzip gleich zu gestalten.

Wir wissen alle, dass wir nicht nur ein Wort, sondern sehr viele Worte speichern müssen. Die Verallgemeinerung der Schaltung aus Abbildung 39 auf mehr als ein Wort ist nicht ganz so einfach. Man muss in der Lage sein auszuwählen, welches Wort gerade gelesen oder geschrieben werden muss. Dazu wird auf jeden Fall eine weitere Eingabe benötigt. Wir fangen zunächst einmal klein an und erweitern unseren 3-Bit-Speicher so, dass zwei Worte gespeichert werden können. Dann genügt uns ein weiterer Eingang A_0; dabei wird je nach Belegung von A_0 das eine oder das andere Wort angesprochen. Die Schaltung in Abbildung 40 realisiert genau das.

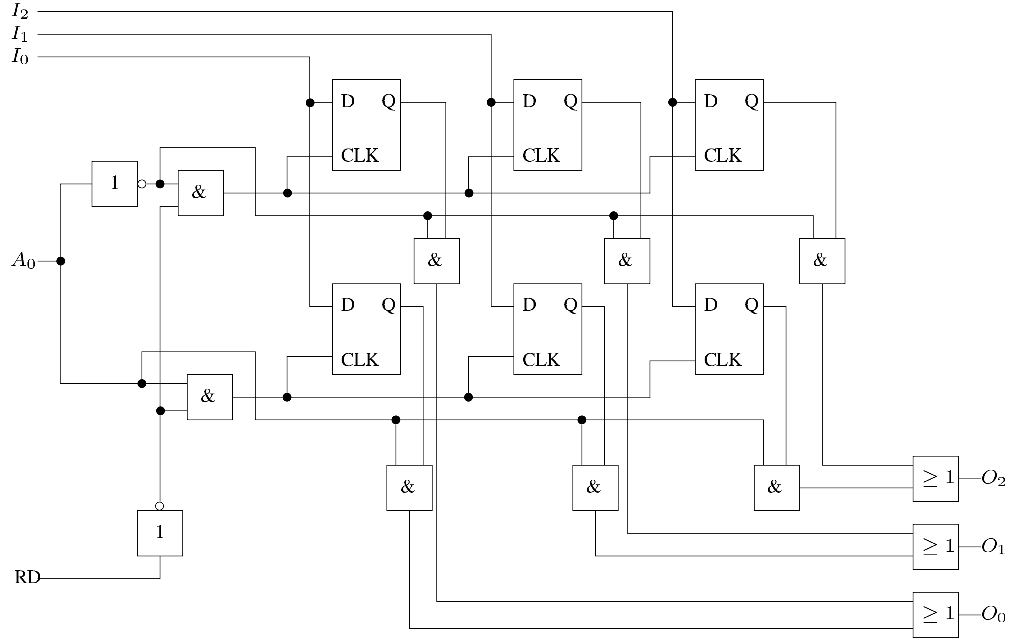


Abbildung 40: Speicher für zwei Worte der Länge drei

Die Adressleitung A_0 benutzen wir, um bei genau einem der beiden Flip-Flop-Reihen Aktivität herzustellen. Die unter den Flip-Flop-Reihen dargestellten Reihen der Und-Gatter erreichen, dass eine im Flip-Flop gespeicherte 1 nur dann weitergegeben wird, wenn die entsprechende Flip-Flop-Reihe durch das Adressbit A_0 ausgewählt ist. Darum können wir vor den Ausgängen die Ausgabe dann mit Oder-Bausteinen berechnen. Da stets nur eine Flip-Flop-Reihe adressiert ist, könnten wir auch XOR-Bausteine verwenden, ohne dass das einen Unterschied machte.

Es ist hilfreich, wenn man in einem Rechner mehr als einen Speicherbaustein verwenden kann. Dann muss zur Realisierung von größerem Speicher nicht auf einen größeren Speicherbaustein zurückgegriffen werden. Um die Schaltungen zum Zugriff auf die einzelnen Speicherbausteine nicht für jeden Speicherbaustein neu realisieren zu müssen, wäre es schön, wenn man einen Speicherbaustein insgesamt „ein- oder abschalten“ könnte, wenn es also einen weiteren Eingang gäbe, der dem Speicherbaustein insgesamt mitteilt, ob er jetzt angesprochen ist oder nicht. Wir erweitern darum den Speicherbaustein aus Abbildung 40 um einen weiteren Eingang CS (für Chip Select). Wenn $CS = 1$ gilt, soll sich der neue Baustein wie die Schaltung aus Abbildung 40 verhalten, wenn $CS = 0$ ist, wollen wir weder Daten aus dem Speicher lesen noch welche schreiben. Die entsprechend erweiterte Schaltung stellen wir in

Abbildung 41 dar. Die Realisierung ist denkbar einfach und bedarf keiner weiteren Kommentierung.

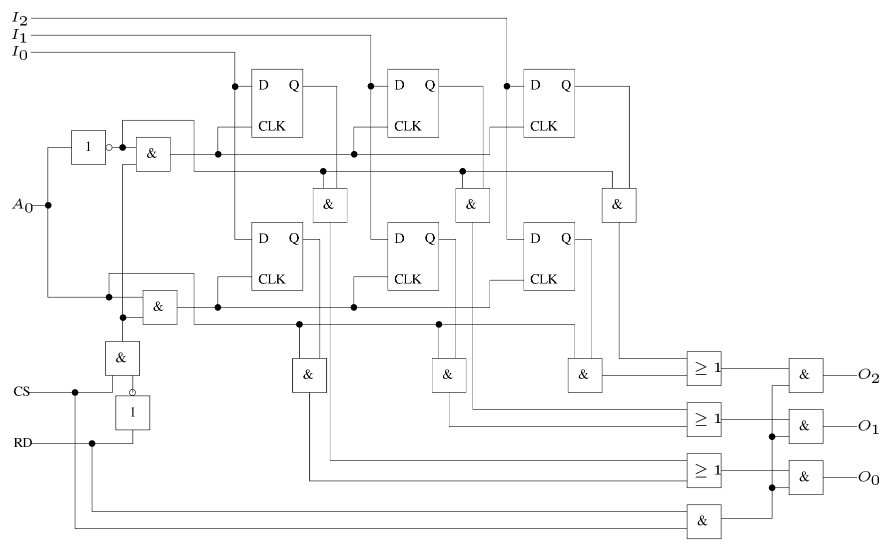


Abbildung 41: Speicher für zwei Worte der Länge drei mit „Chip Select"

Wenn man sich Speicherbausteine in der Praxis ansieht, fällt auf, dass sie nicht getrennte Datenleitungen für Ein- und Ausgaben haben. Stattdessen gibt es einfach nur Datenleitungen, die je nach Belegung von RD als Ein- oder Ausgabe dienen. Wir sehen uns darauf hin noch einmal Abbildung 41 an. Wir können in dieser Schaltung die Ein- und Ausgänge nicht einfach zusammenlegen. Wenn $RD = 0$ ist, so liegen auf den Ausgängen überall 0 an; das kann man in einer Schaltung nicht unbedingt einfach mit einem Eingangssignal überlagern. Man entledigt sich in der Praxis dieses Problems, indem man so genannte Tri-State Buffer einsetzt, deren Symbol wir in Abbildung 42 sehen, wo wir die drei Und-Gatter an den Ausgängen entsprechend ersetzt haben. Wenn $RD = 1$ gilt, verhalten sich diese Tri-State-Buffer, als ob sie gar nicht da wären; eine anliegende 0 wird unverändert weitergeleitet, das gleiche gilt für eine anliegende 1. Wenn jedoch $RD = 0$ ist, so haben die Tri-State-Buffer hohen Widerstand; das bedeutet, dass anliegende Signale nicht weitergeleitet werden und die anliegende Leitung für den Rest der Schaltung praktisch nicht vorhanden ist.

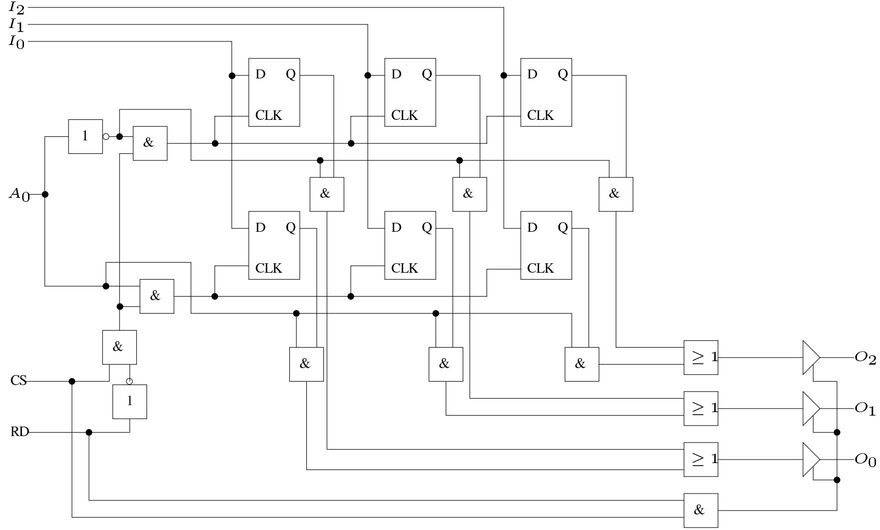


Abbildung 42: Speicher für zwei Worte der Länge drei mit „Chip Select“ und Tri-State-Buffer

In Abbildung 42 könnten wir jetzt also die Ein- und Ausgänge jeweils zusammen auf eine Leitung legen, ohne dass es zu Störungen käme. Wir verzichten hier auf die Darstellung einer entsprechend modifizierten Schaltung.

Natürlich ist ein 2-Wort-Speicher für die Praxis immer noch viel zu klein. Wir müssen unsere Ideen hier also noch einmal verallgemeinern. Das ist aber nicht schwer. Wir sehen uns noch einmal die Schaltung aus Abbildung 42 an und erkennen, wie die Auswahl des Wortes realisiert ist (Abbildung 43). Recht direkt auf die Adressleitung A_0 folgen zwei Leitungen, von denen jeweils genau eine das Signal 1 trägt und damit anzeigt, dass das entsprechende Wort ausgewählt worden ist. Man nennt diese Leitungen *Word Select Line*.

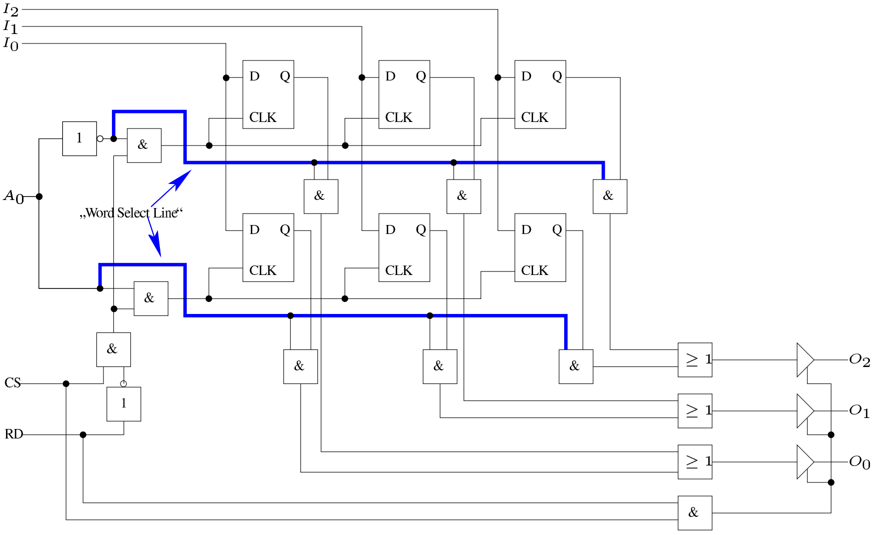


Abbildung 43: Word Select Line im Speicher für zwei Worte der Länge drei mit „Chip Select“ und Tri-State-Buffer

Wir wollen das Konzept der Word Select Line auf mehr als zwei Worte erweitern. Wir wissen schon vom Multiplexer, dass wir mit d Adressleitungen genau 2^d Daten adressieren können. Es bietet sich also an, als Anzahl der in einem Speicherbaustein zu speichernden Worte eine Zweierpotenz zu wählen (also 2^k für ein $k \in \mathbb{N}$) und dann mit k Adressleitungen A_0, A_1, …, A_{k-1} zu arbeiten. Wir brauchen also eine boolesche Funktion $f\colon B^k \to B^{2^k}$, für die $f(A_0, A_1, \ldots, A_{k-1}) = (w_0, w_1, \ldots, w_{2^k} - 1)$ gilt, wobei $w_i = 1$ ist, wenn $i = (A_{k-1}\, A_{k-2} \cdots A_0)_2$ gilt und $w_i = 0$ sonst. Es wird also in den 2^k Ausgabebits $w_0, w_1, \ldots, w_{2^k-1}$ immer genau eine 1 erzeugt. Die Position dieser 1 wird durch die in den k Adressvariablen $A_0, A_1, \ldots, A_{k-1}$ gegebene Adresse bestimmt. Man nennt diese Funktion, die ja in gewisser Weise einen Multiplexer umkehrt, $k \times 2^k$-Decoder. Für $k = 2$ kann man eine Schaltnetzrealisierung eines 2×4-Decoders leicht direkt angeben (Abbildung 44). Die Verallgemeinerung auf größere k ist nicht schwierig. Es lohnt sich an dieser Stelle, sich daran zu erinnern, dass wir einen festen Speicher (ROM) mit Hilfe eines PLAs realisieren konnten. Wenn man sich jetzt die Realisierung in Tabelle 12 (Seite 89) noch einmal ansieht, erkennt man direkt, dass im oberen Teil ein Decoder realisiert wird.

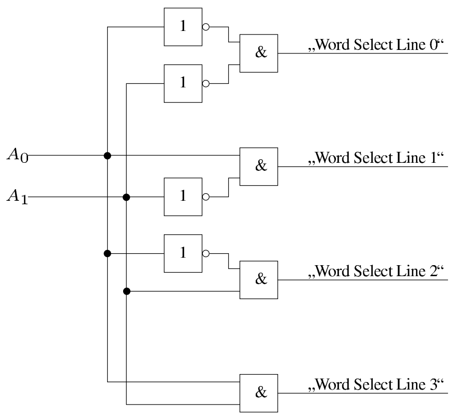


Abbildung 44: 2×4-Decoder

Wir wissen jetzt also, wie wir einen Speicher für 2^k Worte mit je b Bits realisieren können. Die Schaltung hat im Wesentlichen lineare Größe in $2^k \cdot b$, dabei werden genau $2^k \cdot b$ D-Flip-Flops benötigt. Ein so aufgebauter Speicher hat den Vorteil, dass der Speicherinhalt ohne besonderen Aufwand erhalten bleibt (natürlich nur, so lange die Schaltung mit Strom versorgt wird) und der Speicher ausgesprochen schnell geschrieben und gelesen werden kann. Die Eigenschaft, dass der Speicherinhalt bis zum nächsten Beschreiben erhalten bleibt, hat diesem Typ von Speicherbausteinen den Namen SRAM (für Static Random Access Memory) eingebracht. Unangenehm ist, dass die Realisierung verhältnismäßig teuer ist und auch nicht beliebig kompakt. Darum wird für die Realisierung von wirklich großen Speichern, wie sie in modernen Computern als Hauptspeicher eingesetzt werden, nicht auf SRAM-Bausteine zurückgegriffen. Stattdessen kommen so genannte DRAM-Bausteine zum Einsatz; dabei steht das D in DRAM für dynamic. Wir wollen hier nicht wirklich auf die technischen Details eingehen, aber doch den prinzipiellen Unterschied erläutern. Die Schaltung eines DRAM-Bausteins sieht der eines SRAM-Bausteins zunächst einmal sehr ähnlich, jedoch werden die Flip-Flops durch Kondensatoren ersetzt. Es dürfte allgemein bekannt sein, dass man auf einen Kondensator eine Spannung laden kann, so dass auch auf diese Weise prinzipiell ein Wert gespeichert werden kann: Die Codierung steckt im Zustand des Kondensators, der entweder aufgeladen oder entladen ist. In realen Schaltungen verlieren Kondensatoren aber aufgrund nicht perfekter Isolation zwischen ihren Anschlüssen oder parallel geschalteter Widerstände im Laufe der Zeit nach und nach ihre Ladung. Darum reicht es nicht, einen Wert einmal zu schreiben. Es ist vielmehr erforderlich, die Speicherung in relativ kurzen Abständen (wir bewegen uns hier im Millisekundenbereich) zu wiederholen. Man muss also zusätzlich eine Schaltung realisieren, die in recht

kurzen Abständen die Inhalte des Speichers liest und dann unverändert wieder zurückschreibt. Einen solchen Zyklus nennt man *Refresh*. Während eines Refreshs steht der Speicherbaustein natürlich nicht zum Lesen zur Verfügung. Darum sind DRAM-Bausteine erkennbar langsamer als SRAM-Bausteine. Ihr geringer Preis, die geringe Größe und großen erreichbaren Speicherdichten machen sie aber dennoch zu den Speicherbausteinen, mit denen der Hauptspeicher von Computern realisiert wird.

Moores Gesetz

Wir wissen alle, dass moderne Hauptspeicher sehr groß sind und dass auch moderne Cache Speicherbausteine (also SRAMS) sehr groß sind. Man kann leicht einen SRAM-Baustein kaufen, der zum Beispiel 128K Worte (also $128 \cdot 1024 = 2^{17}$) Worte von jeweils 8 Bit speichert, der also intern $2^{17} \cdot 8 = 2^{20} = 1\,048\,576$ Flip-Flops enthält. Wir sehen unmittelbar ein, warum heutzutage solche Schaltungen nicht mehr mit der Hand entworfen und gezeichnet werden können und der Computer als Werkzeug aus dem Schaltungsentwurf nicht mehr wegzudenken ist. Wir wissen aber auch, dass es nicht immer so große Speicherbausteine gegeben hat. Das legt die Frage nahe, wie sich die Technik in der Zukunft entwickeln wird.
Fragen über zukünftige Entwicklungen sind inhärent schwierig und nicht immer wirklich seriös beantwortbar. Um so erstaunlicher ist, dass eine Antwort auf diese Frage, die schon 1965 gegeben wurde, heute zumindest noch diskussionswürdig ist. Gorden Moore (der an der Gründung von Intel beteiligt war) hat 1965 in Electronics einen Artikel mit dem Titel „Cramming more components onto integrated circuits“ veröffentlicht. Er betrachtet darin die Entwicklung der Chiptechnik in den vergangenen Jahren und stellt fest (also empirisch), dass sich die Anzahl der Transistoren (als bestimmende Bausteine), die auf einem Chip realisierbar sind, etwa alle 18 Monate verdoppelt. Über diese rein empirische Beobachtung hinaus formuliert Moore in diesem Artikel aber auch die Vermutung, dass dieser Trend noch für 10 Jahre anhalten wird. Diese Behauptung ist kühn, weil „Verdopplung“ offenbar exponentielles Wachstum bedeutet und uns allen klar ist, dass es in realen Systemen exponentielles Wachstum immer nur eine sehr begrenzte Zeit geben kann. Wir sind heute in der Lage auf das Jahr 1975 zurückzublicken und festzustellen, dass Moore mit seiner Vermutung Recht hatte. Betrachtet man zum Beispiel die Anzahl der Transistoren auf den Prozessoren der Firma Intel angefangen beim 4004 aus dem Jahr 1971 bis hin zum Pentium III aus dem Jahr 1999 (siehe Tabelle 30) so stellt man tatsächlich ein vermutlich exponentielles (wenn auch nicht ganz Moores Vorhersage erfüllendes) Wachstum fest. Man kann an der Spalte „minimale Bauteilgröße“ auch einen der Gründe

dafür erkennen, warum exponentielles Wachstum über 30 Jahre überhaupt möglich war: die Bauteilgröße hat sich drastisch verkleinert, so dass immer mehr Transistoren auf der gleichen Chipfläche untergebracht werden können.

Jahr	Prozessor	Anzahl Transistoren	minimale Bauteilgröße
1971	4004	2.300	10μm
1972	8008	2.500	10μm
1974	8080	4.500	6μm
1976	8085	6.500	3μm
1978	8086	29.000	3μm
1982	80286	134.000	1,, μm
1985	80386	275.000	1,5μm
1989	80486 DX	1.200.000	1μm
1993	Pentium	3.100.000	0,8μm
1997	Pentium II	7.500.000	0 35μm
1999	Pentium III	9.500.000	0,17μm

Tabelle 30: Anzahl Transistoren auf Intel-Chips zum Vergleich mit Moores Gesetz

Moores Prophezeiung ist viel beachtet und diskutiert worden, sie ist unter dem Namen „Moore's Law" geradezu berühmt geworden. Sie ist auf viele andere Bereiche übertragen worden, etwa auf Festplattengrößen oder die Anzahl von Seiten im WWW – man sollte sich daran erinnern, dass Moore selbst nur von der Anzahl von Transistoren in einer integrierten Schaltung spricht. Wir wollen uns nicht an der spekulativen Debatte beteiligen, ob sich dieser Trend noch in den nächsten Jahren fortsetzt. Aber wir sollten trotzdem noch einen Moment bei der Debatte verweilen. Moores Gesetz ist sowohl als „inspirierend" als auch als „Fluch der Computer-Industrie" bezeichnet worden. Für beides gibt es gute Gründe. Eine wichtige Perspektive auf Moores Gesetz betrachtet die Aussage als „self-fulfilling prophecy", als eine Vorhersage, die eintrifft, eben weil sie vorhergesagt worden ist. Das ist nicht so ganz fern liegend. Wenn Designer und Ingenieure wissen, dass im Allgemeinen erwartet wird, dass sich die Anzahl der Transistoren bei jedem neuen Prozessor verdoppelt, so ist das eben auch eine Zielvorgabe, an der man sich orientieren kann. Negativ an dieser Vorhersage ist, dass Firmen sich trauen können, Software nicht-optimiert in einem Zustand auf den Markt zu bringen, der für die noch vorherrschende Computergeneration eigentlich zu groß und zu langsam ist, darauf vertrauend, dass schon in kurzer Zeit die nächste Rechnergeneration ausreichend groß und schnell sein wird.

Register

Die Recheneinheit eines Computers führt ihre Berechnungen in aller Regel weder direkt im Hauptspeicher noch direkt im Cache aus – zumal der Unterschied zwischen Hauptspeicher und Cache der Recheneinheit ohnehin in der Regel verborgen bleibt. Es gibt spezielle Speicherzellen, die der Recheneinheit zugeordnet sind und in denen die einzelnen Rechnungen ausgeführt werden. Auch die Steuerung des Rechenwerks, das so genannte Leitwerk, verfügt über solche eigenen Speicherzellen. Natürlich werden auch diese Speicherzellen, die den Namen *Register* tragen, einfach durch Flip-Flops realisiert. Sie stellen aber in der Regel besondere, zusätzliche Funktionalität zur Verfügung, die eine gesonderte Betrachtung im Abschnitt 5.3 „Speicher“ lohnt.

Eine häufig benutzte Funktion, mit der wir uns bisher noch gar nicht beschäftigt haben, ist das Verschieben von Speicherinhalten. Wir stellen uns einige linear angeordnete gespeicherte Bits $b_0, b_1, \dots, b_l$ vor. Wenn wir den Inhalt nach links verschieben, so weisen wir der Speicherzelle von b_0 den Wert von b_1 zu, der Speicherzelle von b_1 den Wert von b_2, allgemein also der Speicherzelle von b_i den Wert von b_{i+1}. Unberücksichtigt bleibt dabei die Zelle ganz rechts, also b_l. Man unterscheidet zwei Möglichkeiten. Entweder füllt man b_l mit einem festen Wert x (in der Regel $x = 0$), oder man speichert in b_l den alten Wert von b_0 (der im ersten Fall ja einfach verloren geht). Den zweiten Fall nennt man zyklisches Verschieben, den ersten Fall nicht zyklisches Verschieben (oder auch Bit Shift). Ganz analog kann auch (zyklisch oder nicht zyklisch) nach rechts verschoben werden, dann wird also allgemein b_i mit dem Wert von b_{i-1} gefüllt.

Wir wollen hier exemplarisch ein *Schieberegister* entwerfen, das jeweils nichtzyklisch nach links und rechts verschieben kann, wobei die frei werdenden Bits mit einem zu wählenden Bit x aufgefüllt werden. Wir vereinfachen unseren Entwurf und beschäftigen uns nur mit den eigentlichen Verschiebungen, das Lesen und Schreiben der Registerinhalte schließen wir also explizit aus.

Wir haben darum zwei Eingabebits, ein Bit d, das die Schieberichtung bestimmt (bei $d = 0$ schieben wir nach links, bei $d = 1$ nach rechts) und ein Bit x, das den Wert des leer gewordenen Bits bestimmt. Wir führen den Entwurf hier nur für ein 3-Bit-Register durch, was natürlich eine nicht realistisch kleine Größe hat. Es reicht aber aus, um alle Überlegungen vorzuführen: Offensichtlich haben das linke und das rechte Bit eine Sonderstelle, alle anderen Bits sind aber gleichartig, so dass es ausreicht, von diesen „Mittelbits“ nur eines zu haben, um alle nötigen Überlegungen anstellen zu können.

Wir definieren das gewünschte Verhalten formal durch Angabe eines Mealy-Automaten und können dann die klassischen Schritte der Schaltwerk-Synthese zur Realisierung durchlaufen. Wir kommen vereinbarungsgemäß mit dem

Eingabealphabet $\Sigma = \{00, 01, 10, 11\}$ aus, wobei das linke Bit eines Eingabebuchstaben $w \in \Sigma$ die Schieberichtung d angibt und das rechte Bit den Wert von x. Da wir keine Ausgabe brauchen, legen wir $\Delta = \emptyset$ fest. Als Zustände genügen uns $Q = \{000, 001, 010, 011, 100, 101, 110, 111\}$, da wir ja nur die gespeicherten Bits kennen müssen. Die Ausgabefunktion λ für die leere Ausgabe ist leicht definiert, es ist $\lambda(q, w) = \varepsilon$ für alle $q \in Q$ und alle $w \in \Sigma$. Welchen Zustand $q \in Q$ wir zum Startzustand machen, spielt keine Rolle. Es bleibt also nur noch die Zustandsüberführungsfunktion δ festzulegen, das können wir zum Beispiel in Form einer Tabelle machen. Da wir acht Zustände und vier Eingaben haben, hätte eine solche Tabelle aber 32 Zeilen. Wenn man etwas geschickter vorgeht, kann man aber eine sehr kurze Tabelle aufschreiben, die auch δ exakt spezifiziert und gleichzeitig unser Strukturwissen über Schieberegister widerspiegelt. Dazu benutzt man die Notation $q_1q_2q_3 \in Q$ für einen Zustand, in dem die drei Bits eben einzeln mit q_1, q_2, q_3 bezeichnet sind. Analog schreiben wir $dx \in \Sigma$. Mit dieser Notation ergibt sich dann Tabelle 31. Man kann für nur drei Bits auch gut die Zustandsüberführungsfunktion in einem Diagramm wiedergeben (Abbildung 45), aus dem die Symmetrien gut deutlich werden.

q			w		$\delta(q, w)$		
q_1	q_2	q_3	d	x	q_1'	q_2'	q_3'
q_1	q_2	q_3	0	x	q_2	q_3	x
q_1	q_2	q_3	1	x	x	q_1	q_2

Tabelle 31: Vereinfachte Tabelle zum Mealy-Automaten zum Schieberegister

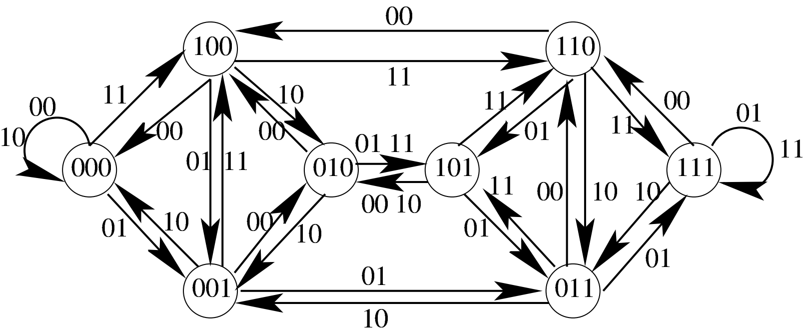


Abbildung 45: Diagramm des Mealy-Automaten zum Schieberegister

Der nächste Schritt bei der Schaltwerk-Synthese besteht in der Auswahl der Flip-Flop-Typen und der Aufstellung der Funktionen für die Flip-Flop-Ansteuerung. Wir wählen die besonders einfachen RS-Flip-Flops und kommen zu recht einfach strukturierten Ansteuertabellen, die wir getrennt für die

drei Zustandsbits angeben (Tabelle 32 für das linke Zustandsbit, Tabelle 33 für das rechte Zustandsbit und Tabelle 34 für das mittlere Zustandsbit).

q			w		q_l	R	S
q_1	q_2	q_3	d	x			
q_1	q_2	q_3	0	x	q_2	$\overline{q_2}$	q_2
q_1	q_2	q_3	1	x	x	$\overline{x}$	x

Tabelle 32: Vereinfachte Tabelle zum Mealy-Automaten zum Schieberegister, linkes Zustandsbit

q			w		q_r	R	S
q_1	q_2	q_3	d	x			
q_1	q_2	q_3	0	x	x	$\overline{x}$	x
q_1	q_2	q_3	1	x	q_2	$\overline{q_2}$	q_2

Tabelle 33: Vereinfachte Tabelle zum Mealy-Automaten zum Schieberegister, rechtes Zustandsbit

q			w		q_m	R	S
q_1	q_2	q_3	d	x			
q_1	q_2	q_3	0	x	q_3	$\overline{q_3}$	q_3
q_1	q_2	q_3	1	x	q_1	$\overline{q_1}$	q_1

Tabelle 34: Vereinfachte Tabelle zum Mealy-Automaten zum Schieberegister, mittleres Zustandsbit

Die Ansteuerung der Flip-Flops ergibt sich ganz einfach, wenn man sich klar macht, dass es bei einem RS-Flip-Flop genügt, $(R, S) = (\overline{y}, y)$ zu wählen, um einen Wert y zu speichern. Es ist sehr hilfreich zu beobachten, dass wir in allen Fällen $R = \overline{S}$ haben. Es genügt also auf jeden Fall, die Ansteuerfunktion für S zu realisieren, die Ansteuerfunktion für R ergibt sich bei allen drei Flip-Flops dann einfach durch Negation. Eine Umsetzung der einfachen Ansteuerfunktionen liegt auf der Hand, so dass wir die in Abbildung 46 dargestellte Schaltung bekommen als Realisierung eines nicht-zyklischen 3-Bit-Schieberegisters.

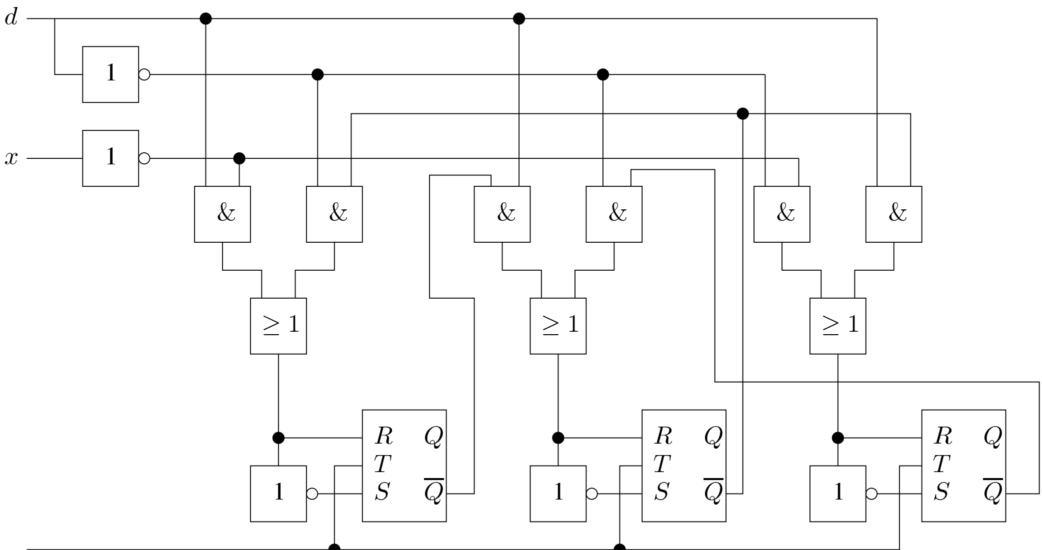


Abbildung 46: 3-Bit-Schieberegister

Zähler und Taktableitung

Vielfach möchte man gerne Schaltwerke realisieren, die vorwärts und vielleicht auch rückwärts zählen können. Wie kann man das bewerkstelligen? Natürlich müssen wir den aktuellen Zählerstand im Zustand codieren. Da die Zustandsmenge eines Automaten vereinbarungsgemäß endlich ist, können wir auch nur endlich viele Zahlen abzählen. Was macht man beim Erreichen des größten erlaubten Wertes? Die übliche Antwort auf diese Frage besteht darin, dann wieder bei Null anzufangen. Man spricht aus nahe liegenden Gründen dann von einem Modulo Zähler.

Bevor wir uns an die Realisierung machen, wollen wir noch einen anderen Grund dafür nennen, aus dem man sich in einem Rechner einen Zähler wünschen kann. Wir wissen, dass moderne Computer grundsätzlich als getaktete Schaltwerke beschrieben werden können. Es gibt also eine „Uhr“, die einen Takt vorgibt. Es sollte aber bis jetzt klar geworden sein, dass es Sinn machen kann, nicht in allen Systemkomponenten den gleichen Takt zu verwenden. Natürlich kann man einen fest vorgegebenen Takt nicht beschleunigen; es ist aber leicht denkbar, daraus langsamere Takte abzuleiten. Eine einfache und nahe liegende Möglichkeit dafür besteht in der Verwendung eines Modulo Zählers, der bei jedem neuen Erreichen der Null ein Taktsignal abgibt. So kann dann in jedem d-ten eigentlichen Takt ein Taktsignal eines langsameren Taktes erzeugt werden. Wir wollen jetzt erst exemplarisch besprechen, wie ein Modulo Zähler realisiert werden und uns dann noch kurz

überlegen, dass man die Ableitung neuer Takte auch einfacher bewerkstelligen kann.

Wir wollen also ein Schaltwerk realisieren, das in jedem Takt entweder um 1 vorwärts oder um 1 rückwärts zählt. Das geschieht in Abhängigkeit von einem Eingabebit d, wobei $d = 0$ das Rückwärtszählen und $d = 1$ das Vorwärtszählen implizieren soll. Wir benutzen n Zustände, also $Q = \{0, 1, \ldots, n-1\}$, wobei wir den Zustand mit dem Zählerstand identifizieren. Wie gesagt haben wir $\Sigma = \{0, 1\}$. Auf die ja ziemlich triviale Ausgabe verzichten wir hier in unserer Beschreibung. Die Zustandsüberführungsfunktion lässt sich einfach als $\delta(q, 1) = q + 1$ für $q < n - 1$, $\delta(q, 0) = q - 1$ für $q > 0$ sowie $\delta(0, 0) = n - 1$ und $\delta(n - 1, 1) = 0$ vollständig beschrieben.

Wir wollen das Schaltwerk mit RS-Flip-Flops realisieren und können konkret für das Beispiel $n = 8$ direkt die Wertetabelle einschließlich Flip-Flop-Ansteuerungen angeben (Tabelle 35). Wir codieren den Zustand mit drei Bits und bezeichnen das linke, mittlere und rechte Bit jeweils mit dem Anfangsbuchstaben (also l, m, oder r), dazu passend bezeichnen wir die Flip-Flop-Ansteuerung.

d	q			q_{neu}			R_l	S_l	R_m	S_m	R_r	S_r
0	0	0	0	1	1	1	0	1	0	1	0	1
0	0	0	1	0	0	0	*	0	*	0	1	0
0	0	1	0	0	0	1	*	0	1	0	0	1
0	0	1	1	0	1	0	*	0	0	*	1	0
0	1	0	0	0	1	1	1	0	0	1	0	1
0	1	0	1	1	0	0	0	*	*	0	1	0
0	1	1	0	1	0	1	0	*	1	0	0	1
0	1	1	1	1	1	0	0	*	0	*	1	0
1	0	0	0	0	0	1	*	0	*	0	0	1
1	0	0	1	0	1	0	*	0	0	1	1	0
1	0	1	0	0	1	1	*	0	0	*	0	1
1	0	1	1	1	0	0	0	1	1	0	1	0
1	1	0	0	1	0	1	0	*	*	0	0	1
1	1	0	1	1	1	0	0	*	0	1	1	0
1	1	1	0	1	1	1	0	*	0	*	0	1
1	1	1	1	0	0	0	1	0	1	0	1	0

Tabelle 35: Wertetabelle für den Zähler (mod 8)

Mit den üblichen Mitteln finden wir zum Beispiel

$$
\begin{aligned}
R_l &= \overline{d}\, q_l\, \overline{q_m}\, \overline{q_r} \lor d\, q_l\, q_m\, q_r \\
S_l &= \overline{d}\, \overline{q_l}\, \overline{q_m}\, \overline{q_r} \lor d\, \overline{q_l}\, q_m\, q_r \\
R_m &= \overline{d}\, \overline{q_m}\, q_r \lor \overline{d}\, q_m\, \overline{q_r} \lor d \overline{q_m}\, \overline{q_r} \lor q_m\, q_r \\
S_m &= \overline{R_m} \\
R_r &= q_r \\
S_r &= \overline{R_r}
\end{aligned}
$$

für die Ansteuerfunktionen. Das ist offenbar nicht besonders elegant; wir hatten beim Entwurf sehr viele Freiheiten, die wir jeweils ohne viel Überlegung praktisch willkürlich genutzt haben. Es ist durchaus möglich zu besseren Realisierungen zu kommen. Mit der eben genannten Wahl ergibt sich die Schaltung aus Abbildung 47.

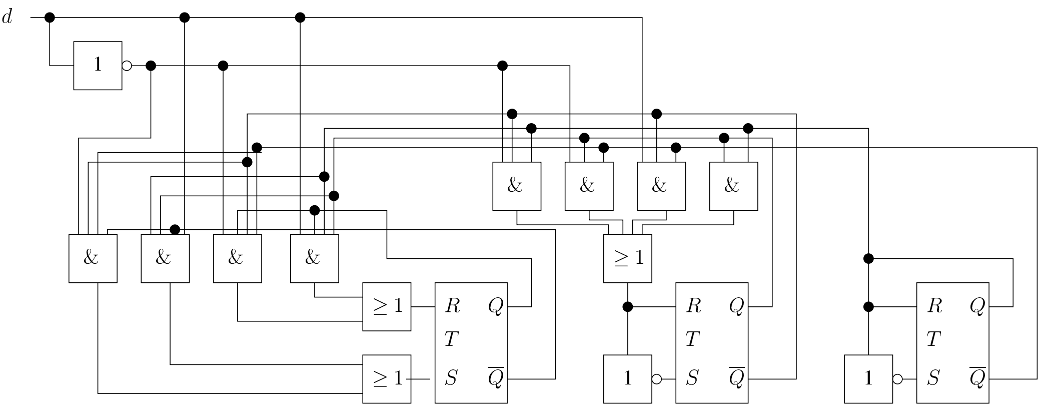


Abbildung 47: Vor- und Rückwärtszähler (mod 8)

Wenn es nur darum geht, zu einem langsameren Takt zu kommen, ist ein Zähler aber zu großer Aufwand. Betrachten wir zum Beispiel die Schaltung aus Abbildung 48.

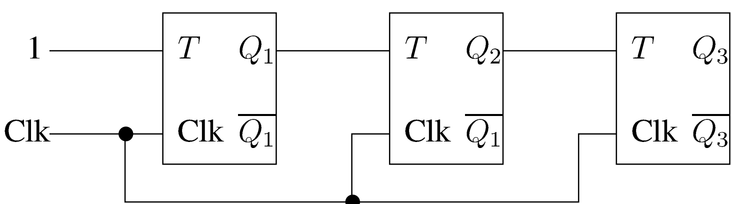


Abbildung 48: Beispiel zur Taktableitung

Wir benutzen ausschließlich T-Flip-Flops und erinnern uns daran, dass bei Eingabe von $T = 0$ der Zustand erhalten bleibt, während Eingabe von $T = 1$ zum Zustandswechsel führt. Die Schaltung ist synchron, das bedeutet, dass bei allen drei Flip-Flops der gleiche Takt Clk anliegt. Wir nehmen an, dass anfangs alle T-Flip-Flops im Zustand 0 sind und betrachten die Situation nach t Takten für $t \in \{0, 1, \ldots, 8\}$. Man kann leicht nachvollziehen, dass sich dann die Zustände ergeben, wie sie in Tabelle 36 angegeben sind.

t	Q_1	Q_2	Q_3
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	1
4	0	0	0
5	1	0	0
6	0	1	0
7	1	1	1
8	0	0	0

Tabelle 36: Zustände zur Abbildung 48

Wenn man noch etwas mehr Aufwand betreibt, kann man auch noch mehr erreichen. Die Schaltung aus Abbildung 49 zeigt, wie man erreichen kann, Takt gleicher Geschwindigkeit mit verschobener Phase zu haben. Die resultierenden Zustände sind in Tabelle 37 wiedergegeben, man überzeuge sich von der Richtigkeit.

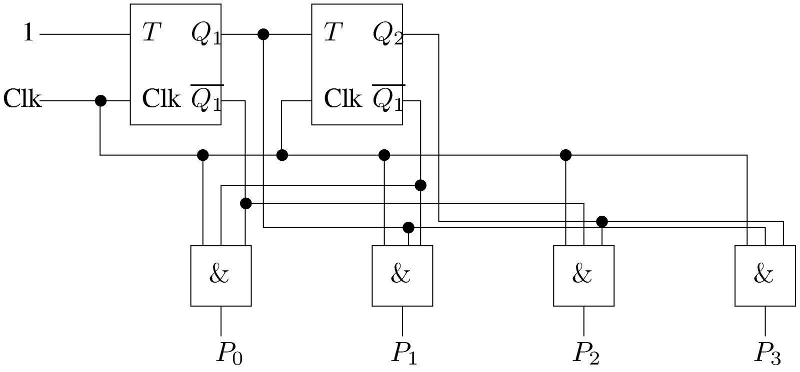


Abbildung 49: Beispiel zur Phasenableitung

t	Q_1	Q_2	P_0	P_1	P_2	P_3
0	0	0	1	0	0	0
1	1	0	0	1	0	0
2	0	1	0	0	1	0
3	1	1	0	0	0	1
4	0	0	1	0	0	0
5	1	0	0	1	0	0
6	0	1	0	0	1	0
7	1	1	0	0	0	1
8	0	0	1	0	0	0

Tabelle 37: Zustände zur Abbildung 49