

Skriptum zur Vorlesung
Einführung in die Informatik

Prof. Dr. Gordon Wassermann

Wintersemester 2006/07

Inhaltsverzeichnis

Vorwort	v
1 Einleitung	1
1.1 Technische Grundbegriffe	7
1.1.1 Darstellung von Zahlen im Binärsystem und anderen nichtdezimalen Systemen	9
1.1.2 Umwandlung von Zahlendarstellungen zwischen ver- schiedenen Basen	12
1.1.3 Rechnen im Binärsystem	16
1.2 Die Organisationsstruktur eines Rechners	27
2 Digitale Logik	41
3 Rechnen mit Zahlen	107
3.1 Nachtrag über Binärbrüche	161
4 Eine einfache CPU	165
5 Realisierung und Regelung	187
6 Formale Sprachen	261
7 Turing Maschinen	307
Literatur	327

Abbildungsverzeichnis

2.1	Ein Feldeffekttransistor	42
2.2	Schaltkreissymbole für Transistoren	43
2.3	Ein Transistor als Inverter	44
2.4	Ein NAND Gatter aus zwei Transistoren	44
2.5	Ein NOR Gatter aus zwei Transistoren	46
2.6	Schaltkreissymbole für Gatter	48
2.7	Ein NOT Gatter durch Zusammenlegung der Eingänge eines NAND Gatters	48
2.8	Ein NOT Gatter als NAND mit einem Eingang	48
2.9	Ein Puffer Gatter aus zwei NOT Gattern	49
2.10	Ein AND Gatter aus einem NAND Gatter gefolgt durch ein NOT Gatter	49
2.11	Ein OR Gatter aus einem NAND Gatter mit NOT Gattern vor den Eingängen	50
2.12	Ein AND mit mehreren Eingängen	51
2.13	Ein AND mit mehreren Eingängen als einzelnes Gatter	52
2.14	Die Schaltung zu Ausdruck (2.1)	56
2.15	Eine DN Schaltung, für den Ausdruck $\bar{a}bc + \bar{a}c$	67
2.16	Ein <i>SR</i> -Flipflop	85
2.17	Ein gesteuertes <i>SR</i> -Flipflop	88
2.18	Ein gesteuertes <i>D</i> -Flipflop	88
2.19	Ein Multiplexer (schematisch)	90
2.20	Ein Multiplexer mit 4 Eingängen	91
2.21	Ein Impulsgeber für die steigende Flanke der Uhr	93
2.22	Ein Mikrozähler mit 6 Zuständen	96
2.23	PLA Gitterpunkttypen	99
2.24	PLA Zellen mit Dioden zur Programmierung	101
2.25	Eine programmierte Spalte der Und-Ebene	102
2.26	Eine programmierte Zeile der Oder-Ebene	103
2.27	Ein PLA für den Mikrozähler mit 6 Zuständen und ein Steu- ersignal	105

3.1	Ein Halbaddierer	136
3.2	Ein Volladdierer aus zwei Halbaddierern	137
3.3	Ein serieller Addierer	138
3.4	Ein von Neumann Addierer	141
3.5	Ein Ripple Carry Addierer	144
3.6	Ein Carry Select Addierer	146
3.7	Die Überlauferkennungsschaltung für eine Bitstelle eines asyn- chronen Carry Ripple Addierers	147
3.8	Schaltung für den Übertrag c_3	152
3.9	Ein Wallace Tree für 8 Summanden	156
4.1	Ein Befehlswort unseres fiktiven Rechners	166
4.2	Die Datenwege in unserer CPU	170
4.3	Vorstufe: Holen des Befehls	172
4.4	Befehle ohne weiteren Speicherzugriff	174
4.5	Befehle mit einem zweiten Speicherzugriff	175
5.1	Das Huffman Modell eines sequentiellen Automaten	189
5.2	Der Zustandsgraph von f	194
5.3	Der Zustandsgraph des Cola-Automaten	199
5.4	Flussdiagramm eines Multiplikationsprogramms	202
5.5	Das markierte Flussdiagramm des Multiplikationsprogramms .	204
5.6	Flussdiagramm einer while -Schleife	207
5.7	Ein Petri-Netz (zeichnerisch)	221
5.8	Das Schalten des Petri-Netzes aus Abbildung 5.7	228
5.9	Eine Schleife in einem Petri-Netz	230
5.10	Alternative Pfade in einem Petri-Netz	230
5.11	Parallele Pfade in einem Petri-Netz	231
5.12	Ein Petri-Netz vom Typ „mutex“	236
5.13	Die Steuerung eines Gerätetreibers mit Semaphoren	248
5.14	Schematischer Aufbau eines Monitors	250
5.15	Flussdiagramm eines Monitoraufrufs	253
6.1	Alle Ableitungsbäume der Grammatik G_7	287
7.1	Eine Turingmaschine	310

Vorwort

Das beiliegende Skriptum begleitete die Vorlesung *Einführung in die Informatik*, die ich im Wintersemester 2006/07 an der Ruhr-Universität Bochum gehalten habe, und gibt deren Inhalt wieder.

Diese Vorlesung, die im Studienplan für Mathematik integriert ist aber gleichzeitig als Serviceveranstaltung für andere Fakultäten fungiert, zieht immer viele Hörer aus verschiedenen Interessensbereichen und mit einem sehr unterschiedlichen Stand an technischen Begabungen und Vorkenntnissen an.

Sie wird besucht von Studenten, die Mathematik mit oder ohne Schwerpunkt Informatik studieren, von Naturwissenschaftlern, und am entgegengesetzten Ende des Spektrums von Geisteswissenschaftlern und anderen fachfremden Interessenten, die die Vorlesung im Optionalbereich hören.

Diese Umstände machen die *Einführung in die Informatik* zu einer sehr wichtigen Veranstaltung, die normalerweise auf kompetente Weise von den Informatikern Prof. Dr. E. Bertsch oder Prof. Dr. H. Simon an unserer Fakultät abgehalten wird. Ich habe Herrn Bertsch dieses eine Mal vertreten, um ihm ein Forschungsfreisemester zu ermöglichen.

Das Skriptum entstand zunächst als meine eigene Vorlesungsvorbereitung und als eine Möglichkeit für mich, meine Gedanken zu dem für mich ungewohnten Thema zu sortieren und klar zu formulieren. Aber natürlich war auch der (stark motivierende) Hintergedanke dabei, dass auch die Hörer von dem Ergebnis profitieren könnten und so auch nicht gezwungen wären, das manchmal nicht so schöne Tafelbild in den Vorlesungsstunden abzuschreiben. Dieser Zweck hat sich anscheinend ausgezahlt, auch wenn das Skriptum nur selten schon am Tag der Vorlesung verfügbar war, denn mit dem Skriptum hat eine große Anzahl der Hörer aus allen Bereichen einschließlich des Optionalbereichs es bis zum Ende des Semesters mit mir ausgehalten, und ich bezweifle, dass das sonst der Fall gewesen wäre.

Als „außerplanmäßiger“ Dozent für diese Veranstaltung bildete ich auch dadurch eine Ausnahme, dass ich noch nicht einmal Informatiker, sondern Topologe bin. Zwar war ich nicht *ganz* ohne Qualifikation, denn ich interessiere mich seit meiner Studienzeit in den sechziger Jahren für Computer und

ihre Anwendung und habe sowohl eine gewisse Ausbildung wie auch „historische“ Erfahrungen auf diesem Gebiet, aber trotzdem waren meine Kenntnisse nicht auf dem neuesten Stand, und ich hätte diese Aufgabe ohne den Rat und die großzügig gewährte Hilfe der Informatiker wohl kaum adäquat bewältigen können.

Prof. Bertsch hat mir eine Kopie der sorgfältig ausgearbeiteten Folien überlassen, die er in seiner *Einführung in die Informatik* auflegt und im Internet veröffentlicht und die mir als Leitfaden für die Themenauswahl und die Vorbereitung der Vorlesungsstunden dienten (auch wenn ich die Reihenfolge gelegentlich verändert habe). Er hat mich auch auf passende Begleitliteratur aufmerksam gemacht, insbesondere auf das Buch von Tannenbaum ([14] oder [15]). Und er hat mich bei der Ankündigung der Vorlesung im Kommentierten Vorlesungsverzeichnis und im Optionalbereich beraten, sowie bei der Anforderung einer geeigneten Hilfskraft für die Korrektur der Übungsaufgaben.

Dieser Korrektor war Michael Kallweit, der nicht nur bei der Korrektur eine sachkundige und zuverlässige Hilfe war, sondern auch das Skriptum aufmerksam gelesen hat und mich auf viele Fehler hingewiesen hat, wofür ich ihm sehr dankbar bin.

Auch andere Personen haben viel zum Gelingen meines Ausflugs in die Informatik beigetragen. Dr. Edgar Korthauer (wie auch Herr Kallweit) hat mir eine umfangreiche ausführliche Ausarbeitung einer früheren Vorlesung von Herrn Simon überlassen und war auch bereit, bei allen mündlichen Prüfungen am Ende des Semesters als Beisitzer zu fungieren. Daniel Reinert hatte drei Jahre lang die Übungen zu den Vorlesungen *Einführung in die Informatik* von Prof. Bertsch gehalten und war so freundlich, mir die gesammelten Übungsaufgaben, die er gestellt hatte, zuzuschicken, so dass ich auf einen reichen Fundus zurückgreifen konnte und kaum noch eigene Aufgaben erfinden musste.

Für den unverzichtbaren Beistand aller oben genannten Personen möchte ich meinen herzlichen Dank aussprechen.

Der *Text* des Skriptums wurde natürlich mit L^AT_EX geschrieben, aber diese Danksagungen wären nicht vollständig ohne anerkennende Erwähnung des L^AT_EX Makropakets „Circuit_macros“ von Prof. J. D. Aplevich an der University of Waterloo in Kanada, das für die bequeme Erstellung *sämtlicher* Diagramme und Zeichnungen (bis auf die allererste) verwendet wurde.

Bochum, im September 2007

Gordon Wassermann

Kapitel 1

Einleitung

Was ist Informatik? Hier ist eine Definition aus der durch freie Zusammenarbeit im Internet entstehenden Enzyklopädie **Wikipedia** (die WWW Adresse des Eintrags ist <http://de.wikipedia.org/wiki/Informatik>):

Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen, insbesondere der automatischen Verarbeitung mit Hilfe von Rechenanlagen. Historisch hat sich die Informatik als Wissenschaft aus der Mathematik entwickelt, während die Entwicklung der ersten Rechenanlagen ihre Ursprünge in der Elektrotechnik und Nachrichtentechnik hat.

Dennoch stellen Computer eigentlich nur ein Werkzeug und Medium der Informatik dar, um die theoretischen Konzepte praktisch umzusetzen.

Von dem niederländischen Informatiker Edsger Dijkstra stammt der Satz „In der Informatik geht es genauso wenig um Computer wie in der Astronomie um Teleskope.“

Diesem Zitat folgen in der Wikipedia einige Hinweise auf den Ursprung des Wortes **Informatik**, das im deutschen Sprachraum erstmals 1957 verwendet wurde (von Karl Steinbuch) und als Bezeichnung für die genannte Wissenschaft 1968 Einzug fand (unter anderem Verwendung durch Bundesforschungsminister G. Stoltenberg bei der Eröffnung einer Tagung in Berlin). Das Wort *informatique* aus *information* und *automatique* wurde ein Jahr früher schon im Französischen verwendet. Die englische Bezeichnung, seit Anfang der 60er Jahre, ist ganz anders: **computer science**.

Wie dem auch sei, es geht in der Informatik also um die *automatische* Verarbeitung von *Information*, wobei aus diesen, den Begriff bestimmenden Merkmalen sich weitere Eigenschaften ableiten lassen oder zumindest sich als üblich oder im Lichte der heutigen Technik als angebracht herausstellen.

Die Information, die verarbeitet wird, kann von jeder beliebigen Art sein: es kann sich um Zahlen handeln, mit denen eine Berechnung durchzuführen ist, um Texte, die geschrieben, formatiert, vervielfältigt, übersetzt, katalogisiert, oder geprüft werden sollen, um Bilddokumente, die auf verschiedene Weisen bearbeitet werden können, aus Datenlisten, in denen Daten gesucht werden oder die auf beliebige Weise auszuwerten sind, oder um jede andere nicht in diese Beschreibung passende Art von Information (wobei dieses Wort eine technische Bedeutung hat, auf die ich nicht weiter eingehen möchte, außer zu sagen, dass nicht jedes wahrnehmbare Phänomen tatsächlich Information enthalten muss — ein zufälliges Rauschen enthält zum Beispiel aus der Sicht der Nachrichtentechnik keine Information, aber die genaue Bedeutung des Wortes „Information“ hängt von der Disziplin ab, in der es verwendet wird, und ist nicht leicht festzulegen).

Auch die Verarbeitung der Information soll ein sinnvolles Ergebnis haben und alte Information in neue verwandeln. Ein Schredder kann man zum Beispiel auch als eine Maschine betrachten, die Information „verarbeitet“, aber ihre Funktion ist nicht wirklich der Gegenstand der Informatik (oder ist bestenfalls ein sehr ausgearteter Spezialfall).

Ein wesentliches Merkmal für die Informatik ist, dass Information *automatisch*, d. h., maschinell, verarbeitet wird. Ein Mensch, der Gedichte schreibt oder Bilder malt, ist kein Gegenstand der Informatik. Ein Mensch, der mit Bleistift und Papier rechnet ist im Prinzip auch kein Gegenstand der Informatik, aber die Rechen*methode*, die der Mensch anwendet, könnte einer sein — allerdings nur deshalb, weil auch eine Maschine diese Rechenmethode anwenden könnte und der Aufwand und die Kosten für die Anwendung dieser Methode im Rahmen der Informatik zu untersuchen wäre.

Wie die Maschine, die die Informationsverarbeitung ausführt, physikalisch beschaffen ist spielt keine *prinzipielle* Rolle (wohl aber eine praktische). Schon ein Abacus ist eine mechanische Rechen*hilfe*, auch wenn ein Mensch die Kugelchen mit den Fingern schieben muss. Seit dem 17. Jahrhundert gibt es mechanische Rechenmaschinen, die die Grundrechenarten mit mehr oder weniger menschlicher Hilfe ausführen können, und diese Maschinen waren bis in die 70er Jahre des letzten Jahrhunderts sehr gebräuchlich für die Ausführung einfacher Berechnungen, bis sie von den elektronischen Taschenrechnern und größeren Rechnern völlig verdrängt wurden.

Allerdings kann man nicht wirklich von automatischer Informationsverarbeitung sprechen, wenn eine Maschine nur Einzelschritte eines Verarbeitungsablaufs oder einer Berechnung ausführt, aber ein Mensch die Zwischenergebnisse notieren muss und für die Weiterverarbeitung einzeln wieder in die Maschine eingeben muss. „Automatisch“ soll zumindest beinhalten, dass eine komplizierte Berechnung *insgesamt* ohne menschliches Eingreifen von

der Maschine durchgeführt wird.

Tatsächlich sind auch früher Maschinen (dazu auch die ersten elektronischen Rechner) gebaut wurden, die auf eine einzige spezielle Aufgabe ausgelegt waren und nur diese Berechnung (für verschiedene Zahlen) durchführen konnten.

Technisch ist das auch eine automatische Bearbeitung, aber interessanter und wünschenswerter sind natürlich „universelle“ Maschinen, die ihre Arbeitsweise verändern können und *jede* Rechenaufgabe prinzipiell durchführen können, wenn man ihnen nur beibringt, welche Rechenschritte sie ausführen sollen. Hauptsächlich befasst sich die Informatik also mit *programmierbaren* Rechnern, die auf verschiedene Verarbeitungsprozesse eingestellt werden können, indem man den gewünschten Verarbeitungsablauf irgendwie in die Maschine eingibt.

Die Programmierbarkeit einer Maschine kann auf mehrere Arten implementiert werden. Schon zu Beginn des 19. Jahrhunderts gab es mechanische Webstühle, deren Webmuster durch Lochkarten gesteuert wurden. Frühe elektronische Rechner wurden oft durch ihre Verkabelung gesteuert, d. h., der Ablauf der Rechenschritte wurde festgelegt, in dem man für diesen Zweck gekennzeichnete Steckplätze durch Kabeln verband.

Das ist natürlich nicht sehr praktisch, denn es verlangt einen großen Aufwand und schränkt die Komplexität der Bearbeitung stark ein. Die flexibelste und leistungsfähigste Idee ist die des „Rechners mit gespeichertem Programm“ (stored program computer), bei dem die Rechenschritte *wie Daten* dem Computer zugeführt werden. Nur die schon ablaufende Arbeitsweise des Computers bestimmt, ob die im Arbeitsverlauf betrachteten Informationen als neue Befehle interpretiert werden, die die Ausführung einer bestimmten internen Operation veranlassen, oder als Daten behandelt werden und von den ausgeführten Operationen „verarbeitet“ werden. Ein wesentlicher Unterschied in der physikalischen Beschaffenheit dieser Arten von Information gibt es nicht; beide werden in der gleichen Gestalt in den Hauptspeicherspeicher des Computers abgelegt.

Noch ein letztes Merkmal steckt implizit in unserer Auffassung von „Informatik“ und sollte hier unbedingt erwähnt werden, nämlich das Merkmal „*digital*“: in der Informatik stellen wir uns stillschweigend vor, dass die zu verarbeitende Information *diskret* und nicht kontinuierlich ist, d. h., dass sie in scharf voneinander abgegrenzten Varianten vorliegt, von denen es bei vorgegebenem Umfang der Daten nur endlich viele gibt.

Das Gegenteil davon sind kontinuierlich vorliegende Daten (die man „*analoge*“ Daten nennt), und der Unterschied wird deutlich, wenn man eine Digitaluhr mit einer Zeigeruhr vergleicht. Die Digitaluhr zeigt in der Regel Stunden, Minuten und Sekunden in Klartext an (und davon gibt es 24 mal 60 mal

60 Varianten, immerhin endlich viele), während die Zeigeruhr ein Kontinuum von Zeigerpositionen durchläuft und man prinzipiell auch kleine Bruchteile von Sekunden ablesen könnte, wenn man genügend fein sehen könnte (und wenn der Zeiger gleichmäßig und nicht ruckartig fortbewegt wird).

Vor einem halben Jahrhundert gab es neben den Digitalrechnern (die heute das Bild beherrschen) gleichberechtigt auch **Analogrechner**, sowohl mechanische (der Rechenschieber, der zur Standardausrüstung jedes Ingenieurs gehörte) wie auch elektrische, die Berechnungen mit den Stromstärken oder Spannungen in einfachen elektrischen Stromkreisen ausführten.

Weil die Ergebnisse aber immer mit einer Messungenauigkeit abgelesen werden mussten und weil die digitalen Rechner immer leistungsfähiger wurden, haben die Digitalrechner die Analogrechner praktisch verdrängt. Auch eigentlich „analoge“ Aufgaben wie etwa Bildbearbeitung können von Digitalrechnern mit genügend viel Rechenkapazität bewältigt werden, wenn man die analogen Daten durch ein genügend feines und genügend fein abgestimmtes Muster von diskreten Daten „digitalisiert“. Weil die Anzahl der Zustände endlich ist, sind sie sicher voneinander zu unterscheiden, und die Daten können ohne Verfälschung oder Verzerrung gespeichert oder übertragen und im genau gleichen Zustand später wieder abgerufen werden.

Wie schon gesagt, spielt die physikalische Beschaffenheit einer Maschine, die die obigen Forderungen erfüllt, für die Informatik keine wesentliche inhaltliche Rolle. Die Maschine muss noch nicht einmal eine physikalische Beschaffenheit haben — es wird in dieser Vorlesung sehr oft von *virtuellen* Maschinen die Rede sein, die nur in unserer Vorstellung existieren. Solche gedachte Maschinen haben trotzdem eine genau festgelegte Funktionsweise und genau festgelegte Fähigkeiten. Manchmal kann man sie mit wirklichen Maschinen (aus Silizium und Draht) realisieren, manchmal kann man sie mit wirklichen Maschinen *und ihrer Software* simulieren, und manchmal könnte man sie überhaupt nicht (oder nur näherungsweise) durch eine echte Maschine realisieren, z. B., weil eine gedachte Maschine einen unendlichen Speicher haben kann, den man in der Wirklichkeit nicht nachbauen könnte.

Neben den virtuellen Maschinen interessieren wir uns natürlich auch für echte physikalische Exemplare, und hier können wir, nicht aus theoretischen sondern alleine aus praktischen Gründen, eine letzte Einschränkung machen. Die physikalischen Maschinen, über die wir sprechen werden, werden immer *elektronische* Rechenanlagen sein, obwohl nichts in der Grundidee der Informatik dies zwingend vorschreibt. Denn die Erfahrung mehrerer Jahrhunderte hat gezeigt, dass mechanische Rechner unpraktikabel sind, weil komplexe mechanische Recheneinheiten wegen der auftretenden physikalischen Kräfte (Reibung, Biegekräfte usw.) nicht zuverlässig in Bewegung zu bringen sind (oder nur mit übermäßigem und unbezahlbarem Aufwand). Obwohl es schon

im 19. Jahrhundert Versuche gab, programmierbare Rechner zu bauen, sind sie letztendlich an diesem Problem gescheitert. Erst die Erfindung des Transistors 1948 ermöglichte den Bau stabil und schnell laufender Rechenanlagen, die für praktische Aufgaben wirklich einsetzbar waren.

Um die oben genannten Punkte zusammenzufassen: womit wir uns in der Informatik und speziell in dieser Vorlesung befassen, sind

- programmierbare
- und in der Regel durch gespeicherte Programme gesteuerte
- elektronische (oder virtuelle)
- Digitalrechner

und ihre Funktionsweise und Organisation, d. h., die Kombination aus ihrer **Hardware** (die physikalischen Bausteine, aus denen Sie aufgebaut sind) und ihrer **Software** (die Programme und Befehlsfolgen, die ihre Funktion steuern), sowie das Zusammenspiel dieser beiden Komponenten.

Die Themen, die uns interessieren, zergliedern sich in drei Bereiche:

Technische Informatik: Hier geht es um die innere Grundstruktur von Rechnern und um die elementaren Operationen auf einfachstem Niveau, die sie ausführen können. Zu diesem Bereich gehören logische Schaltungen, der zentrale Prozessor eines Rechners, und die direkte Programmierung der einfachen Befehle und Rechenanweisungen, die der Rechner unmittelbar „verstehen“ und ausführen kann.

Praktische Informatik: Hier geht es um die Systemprogramme, die den „Verkehr“=Datenfluss innerhalb einer Rechenmaschine regeln und die die Schnittstelle zwischen Mensch und Maschine bilden und die Maschine für die praktische Benutzung erst verfügbar machen, indem sie einen bequemen Zugang ermöglichen.

Die Programmierung eines Rechners in seiner eigenen „Maschinensprache“, wie oben beschrieben, ist für Menschen sehr unbequem und deshalb unpraktikabel. Aus diesem Grund werden Rechnersysteme hierarchisch organisiert, wobei jede Stufe in der Hierarchie unmittelbar mit der vorhergehenden Stufe verbunden ist aber sich immer weiter von dem direkten Kontakt mit der internen Rechnerstruktur entfernt, dafür sich immer mehr der menschlichen Denkweise annähert und immer mehr Strukturen aufweist, die für Menschen leicht zu verstehen und bequem zu handhaben sind.

Auf der obersten Stufe dieser Hierarchie stehen die Systemprogramme, darunter die Betriebssysteme, die den unmittelbaren Zugang des Benutzers zum Rechner gestalten, seine Befehle zur Ausführung weiterleiten und das Zusammenspiel der Benutzer untereinander und mit den Ressourcen der Maschine regeln.

Zum Bereich der praktischen Informatik gehören auch Konzepte zur **parallelen Verarbeitung**. Hier geht es darum, Rechner zu beschleunigen, indem man mehrere Rechenprozesse gleichzeitig ablaufen lässt, wobei diese parallelen Prozesse koordiniert werden müssen (da manche Rechenschritte die Ergebnisse früherer Rechenschritte benötigen).

Theoretische Informatik: Hier geht es nicht um die praktische Realisierung von Rechnern und ihrem tatsächlichen Aufbau, sondern um das „theoretisch Machbare“. Welche Fragen können prinzipiell durch systematische Berechnung beantwortet werden und welche nicht? (Ja, es gibt tatsächlich Aufgaben, die durch Rechner **prinzipiell** nicht lösbar sind!)

Wie beschreibt man Berechnungen, so dass man darüber theoretische Überlegungen anstellen kann? Welche Design Möglichkeiten gibt es für Programmiersprachen (in denen man die auszuführenden Aktionen eines Rechners beschreiben und festlegen kann)? Die Theorie, die darüber Auskunft gibt, ist die Theorie der **formalen Sprachen und ihrer Grammatiken**, die von Linguistikern wie Noam Chomsky entwickelt wurde.

Wir werden diese drei Themenbereiche in der Vorlesung im Detail behandeln, und dabei mit der Theorie der logischen Schaltungen beginnen. Bevor wir das tun und richtig mit der Vorlesung loslegen, sollten wir dennoch die von diesem einführenden Kapitel gebotene Gelegenheit noch ausnutzen, um ein paar allgemeine Dinge zu erörtern, die dem Ganzen einen Rahmen geben. Diese Vorbereitung wird nur wenig Zeit kosten und wird das richtige Verständnis des eigentlichen Themas erst ermöglichen.

Erstens: Es nützt wenig zu lernen, was die Bauteile eines Rechners tun, d. h., wie sie Daten verarbeiten, wenn man nicht verstanden hat, in welcher Form diese Daten dem Rechner vorliegen. Das wäre, als wenn man den Magen einer Kuh und seine zweckdienlichen Besonderheiten untersuchen würde, ohne zu wissen, dass Kühe Gras fressen. Da diese Vorlesung Hörer mit sehr unterschiedlichen Vorkenntnissen hat, wollen wir, damit alle mitkommen, kurz ein paar sehr einfache technische Grundbegriffe erläutern, ohne die man praktisch überhaupt nicht über Rechner und ihren Fähigkeiten reden kann.

Detailkenntnisse eines Rechners und seiner Bauteile nützen uns wenig, wenn wir nicht wissen, wie sie sich nachher zu einer Einheit zusammenfügen. Deshalb wollen wir einen kurzen und sehr allgemeinen Überblick über die Gesamtstruktur eines Rechnersystems geben, ohne auf Spezialisierungen oder Details einzugehen (die kommen ja ohnehin später).

Und zuletzt: um die gegenwärtige Ausprägung der Informatik und die gegenwärtigen Fähigkeiten von Rechnern richtig einschätzen und würdigen zu können, sollte man wissen, wo diese Technologie herkam und wie sie sich zu ihrem jetzigen Stand entwickelt hat, zumal die frühe Entwicklung und die damaligen Zwänge in manchem Detail noch fortlebt. Das heißt, wir werden diesen Abschnitt mit einem historischen Überblick abschließen, damit wir unter Kenntnis der Entwicklung mit einem „geschulten Auge“ die moderne Rechnertechnik betrachten können.

1.1 Technische Grundbegriffe

Menschen lernen schon als Kinder, Daten in verschiedenen Formen aufzunehmen, zu entziffern und weiterzuverarbeiten, als Töne, als Bilder, als Schrift mit einem Alphabet (wenige Schriftzeichen) oder sogar als „Hieroglyphen“ (viele Schriftzeichen, die auch eine Bedeutung tragen, wie bei der sinojapanischen Schrift).

Dies ist für Maschinen sehr viel schwerer, da sie an die Komplexität des menschlichen Gehirns bei weitem nicht herankommen; obwohl es jetzt schon Software gibt, das zum Beispiel relativ zuverlässig gesprochene Sprache erkennen kann, müssen Daten in der Regel in einer sehr einfachen „vorgekauften“ Form dem Rechner zugefügt werden, damit er sie weiterverarbeiten kann (und auch erkannte Sprache wird zunächst von einem Programm in eine einfachere Form übersetzt, bevor sie weiterverarbeitet wird),

Es gibt verschiedene Möglichkeiten, Daten in einfacher Form in einem Rechner darzustellen. Mechanische Rechner, die meistens mit Zahlen umgehen, stellen diese Zahlen durch die Stellung eines Ziffernrades, eines Schiebers oder von Zahnrädern dar, d. h., durch die Position eines beweglichen Objekts, und für diese Position gibt es oft im Konstruktionsschema des Geräts 10 Grundstellungen, weil die Menschen die Gewohnheit haben, Zahlen im Dezimalsystem mit 10 Ziffern zu schreiben.

Elektronische Rechner müssen Daten mittels unterscheidbarer elektrischer Eigenschaften kodieren, und sie müssen in der Lage sein, die erlaubten Zustände dieser Merkmale sicher voneinander zu unterscheiden. Beispiele geeigneter elektrischer Merkmale sind Spannungen, Stromsträrken, oder Widerstände elektrischer Verbindungen.

Desto mehr Zustände erlaubt sind, desto schwieriger und kostenspieler ist die technische Realisierung ihrer Unterscheidung — am Besten geht es, wenn man nur *zwei* Zustände unterscheiden muss. Deshalb werden in der Regel *bipolare Bausteine*, die nur zwei verschiedene Zustände zulassen, zur Konstruktion elektronischer Rechner eingesetzt. Die Merkmale, die unterschieden werden, können zwei Spannungspegel (niedrig und hoch) sein, oder die Polarisierung eines Magnets (frühe Speicherelemente funktionierten so), oder die Stromleitfähigkeit einer Verbindung (geschlossen=leitfähig, offen=kein oder nur wenig Strom fließt) — dieses Merkmal funktioniert wie die Stellung eines Stromschalters und kann auch durch elektromechanische Elemente, nämlich Relais, realisiert werden, oder als rein elektrischer Schalter durch Vakuumröhren oder Transistoren.

Heute werden wegen der schnellen Schaltbarkeit, der Zuverlässigkeit und der billigen Massenproduktion auch in schon funktionsfähigen komplizierten Schaltkreisen hauptsächlich Transistoren zu diesem Zweck verwendet, und diese nicht als Einzelstücke, sondern zu Millionen Stück in *integrierte Schaltkreise* vereint (die neuesten Testchips von Intel tragen über eine Milliarde Transistoren).

Dennoch: die funktionelle Grundeinheit ist immer noch ein *einzelner* bipolarer Baustein (egal wie geartet) und er kann nur zwei Zustände darstellen. Wie kann man mit einer so einfachen Grundlage Daten darstellen?

Viele Konstrukteure früher elektronischer Rechner haben versucht, die von mechanischen Rechnern gewohnte Dezimaldarstellung von Zahlen beizubehalten. Der erste programmierbare elektronische Digitalrechner, ENIAC, benutzte 10 Röhren zur Darstellung einer Ziffer (jede Röhre entsprach einer der Möglichkeiten 0-9, nur eine Röhre war „an“ und sie entsprach der jeweils gespeicherten Ziffer). Für das Rechnen wurde die Position der eingeschalteten Röhre weitergeschoben und gezählt. In anderen Rechnern wurden mehrere Röhren zusammengefasst um eine kodierte Dezimalziffer darzustellen. Aber schon vor der Konstruktion der ersten elektronischen Rechnern war einigen Konstrukteuren klar, dass für Maschinen, die von Natur aus mit einem Universum aus nur *zwei* Zuständen am besten umgehen können, eine andere Zahlendarstellung viel bequemer ist: das **Dualsystem** oder **Binärsystem**, das schon Leibniz ausführlich beschrieb und für alle mögliche auch philosophische Zwecke anwenden wollte, ist eine positionelle Zahlennotation wie das bekannte Dezimalsystem, aber kommt mit nur zwei Ziffernzeichen 0 und 1 aus. Somit kann jede Ziffer mit einem einzigen bipolaren Element dargestellt werden.

Diese Vereinfachung hat für Rechenmaschinen *und für ihre Rechenwerke* große Vorteile, für die menschlichen Benutzer aber die Nachteile, dass man nicht gewohnt ist, in Binärzahlen zu denken, und dass man die Vereinfachung

der Binärnotation damit erkaufen muss, dass die Ziffernfolge zur Darstellung einer Zahl viel länger ist, als bei Dezimalzahlen.

Der erste Nachteil wird dadurch behoben, dass elektronische Rechenmaschinen in der Regel ihre Daten in der für Menschen gewohnten Form annehmen und ausgeben, und sie übernehmen selber die Aufgabe, die Daten für die interne Verarbeitung in die Binärdarstellung umzuwandeln. Die Bauweise der Rechner kann so problemlos auf die Binärdarstellung ausgerichtet sein.

Den zweiten Nachteil (der längeren Zahlendarstellung) behebt man, indem man Binärziffern gruppiert und somit zu einer kürzeren Darstellung (mit mehr Ziffern) übergehen kann, aus der man aber unmittelbar die Binärziffern ablesen kann.

1.1.1 Darstellung von Zahlen im Binärsystem und anderen nichtdezimalen Systemen

Für diejenigen von Ihnen, denen die Darstellung von Zahlen in anderen Basen nicht geläufig ist, hier eine kurze Erläuterung.

In Prinzip kann man Zahlen mit einem Zeichenvorrat aus einer beliebigen endlichen Anzahl n von Zahlensymbolen (Ziffern) in positioneller Notationsweise darstellen, und diese Darstellung erfolgt ganz analog zur üblichen Dezimalnotation, die einfach den Spezialfall $n = 10$ darstellt.

Zum Beispiel, die Zahl 137 hat nur drei Stellen und die größte Ziffer ist 7, aber sie stellt eine wesentlich größere Zahl als $1 \times 3 \times 7$ dar, denn wie jeder weiss ist sie zu verstehen als

$$137 = 1 \times 100 + 3 \times 10 + 7 = 1 \times 10^2 + 3 \times 10^1 + 7 \times 10^0.$$

Jede andere natürliche Zahl kann man durch eine Folge $a_k a_{k-1} \dots a_2 a_1 a_0$ von Ziffern a_i schreiben, wo jede Ziffer eines der zehn Symbole 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ist (und wo normalerweise, außer bei der Zahl 0, die erste Ziffer a_k nicht 0 sein sollte), und diese Zahl hat den Wert

$$a_k a_{k-1} \dots a_2 a_1 a_0 = a_k \times 10^k + a_{k-1} \times 10^{k-1} + \dots + a_2 \times 100 + a_1 \times 10 + a_0$$

Ein solches Zehnersystem wird von den meisten (aber nicht von allen) menschlichen Kulturen benutzt, wahrscheinlich weil man am Anfang der Rechnung die 10 Finger zum Zählen benutzte, aber bis auf diese Ursache gibt es keinen zwingenden technischen Grund, Zahlen im Zehnersystem zu notieren.

Man kann Zahlen nach dem gleichen Prinzip notieren unter Verwendung eines beliebigen endlichen Vorrats aus Ziffernsymbolen, solange wir mehr als nur ein Symbol zur Verfügung haben (mit nur einem Symbol geht eine

positionelle Notation wie oben nicht). Ist $n \geq 2$ die Anzahl der Symbole im Vorrat, so übernimmt n die Rolle der 10 in der Beschreibung oben, aber das ist die einzige Anpassung, die man vornehmen muss.

Die n Ziffern aus dem Zahlenvorrat stehen, in irgendeiner Zuordnung, für die ersten n natürlichen Zahlen (beginnend bei 0!), d.h. für die Zahlen 0, 1, 2, ..., $n-1$. Weil es keine wesentliche Rolle spielt, wie die Zeichen aussehen, werden wir die Ziffern aus dem Zeichenvorrat vorerst einfach 0, 1, 2, ..., $n-1$ nennen.

In dem „ n -er System“ schreibt man jetzt $a_k a_{k-1} \dots a_2 a_1 a_0$ (wobei die a_i aus dem Ziffernvorrat kommen und die erste Ziffer a_k nicht 0 ist, außer wenn sie die einzige Ziffer ist) als Notation für die Zahl

$$a = a_k \times n^k + a_{k-1} \times n^{k-1} + \dots + a_2 \times n^2 + a_1 \times n + a_0.$$

Jede natürliche Zahl a hat bei festem Ziffernvorrat eine *eindeutige* solche Darstellung, aber natürlich ändert sich die Darstellung, wenn man die Anzahl n der Symbole verändert.

Die Zahl n , die die Anzahl der verfügbaren Ziffern wiedergibt, nennt sich die **Basis** dieser Zahlendarstellung, und sie muss, wie gesagt, mindestens 2 sein.

Hat man eine Zahl durch eine Ziffernfolge $a_k a_{k-1} \dots a_2 a_1 a_0$ zur Basis n dargestellt und will man die Basis explizit angeben (zum Beispiel, weil man gleichzeitig mit mehreren Basen hantiert und sonst nicht klar wäre, welche man für diese spezielle Darstellung verwendet hat), so kann man die Basis in Dezimaldarstellung als Index an die letzte Ziffer der Ziffernfolge setzen.

Unsere Beispielzahl 137 von vorhin würden wir in dieser genaueren Schreibweise als

$$137_{10}$$

schreiben, und die gleiche Zahl in Basis 6 wäre

$$345_6$$

(wie man auf diese Ziffern kommt, werden wir weiter unten erklären).

Für die Informatik sind ganz spezielle Basen wichtig. Das **Dual-** oder **Binärsystem** mit $n = 2$ ist für das Rechnen mit elektronischen Rechnern wie geschaffen, da es für jede Stelle in der positionellen Notation einer Zahl nur zwei Möglichkeiten gibt, und man Zahlen in binärer Schreibweise direkt in den bipolaren Bausteinen des Rechners darstellen kann; um eine k -stellige Zahl darzustellen (deren maximale Größe $2^k - 1$ ist) braucht man k solche Bausteine, aber das ist die effizienteste Schreibweise, die überhaupt möglich ist, denn alle 2^k möglichen Gesamtzustände der k Bausteine stellen wirklich eine Zahl dar, d.h., keiner bleibt ungenutzt oder wird verschwendet.

Für Informatiker ist das Binärsystem deshalb das wichtigste Zahlendarstellungssystem überhaupt, und wir werden gleich etwas über das effiziente Rechnen in diesem System sagen.

Jede Ziffer im Binärsystem hat nur die möglichen Zustände 0 und 1, und eine Binärzahl (d. h., eine Zahl geschrieben im Binärsystem) ist also eine Anreihung von Ziffern 0 und 1. Die kleinsten natürlichen Zahlen lauten in diesem System

0, 1, 10, 11, 100, 101, 110, 111, 1000,
1001, 1010, 1011, 1100, 1101, 1110, 1111, 10000, ...

Sie sehen, dass beim Hochzählen der Zahlen eine 0 zu einer 1 wird, aber eine 1, die hochgezählt wird, verursacht sofort einen Übertrag auf die nächste Stelle. Sie erkennen auch, dass die Zahlendarstellung sehr schnell sehr lang wird, und für Menschen deshalb schlecht lesbar.

Wir werden keine Verwendung für „exotische“ Zahlenbasen haben, aber neben dem gewohnten Dezimalsystem und dem Binärsystem werden wir noch das **Hexadezimalsystem** mit der Basis 16 verwenden, weil dieses System eine bequeme Kurzdarstellung für lange Binärzahlen bietet.

Die letzte sichtbare Zahl in der Liste oben ist $2^4 = 16$, und die maximal 4-stelligen Binärzahlen liefern also genau die Dezimalzahlen 0–15, die die Ziffern des 16-er Systems ergeben. Man kann eine Binärzahl deshalb sehr leicht in die äquivalente Hexadezimalzahl verwandeln, indem man die Binärziffern in Gruppen von 4 zusammenfasst und als eine Hexadezimalziffer liest; umgekehrt muss man die Ziffern einer Hexadezimalzahl einfach *einzel*n als 4-ziffrige Binärzahlen (mit führenden Nullen, wenn nötig) hinschreiben und aneinanderreihen, und man erhält die entsprechende Binärzahl.

Wenn wir Hexadezimalziffern immer in der Binärform schreiben würden, hätten wir allerdings nichts gewonnen, denn die Zahl wäre ja nicht kürzer geworden. Stattdessen schreibt man Hexadezimalzahlen mit den gewohnten Ziffern 0–9, und weil diese aber für die 16 Möglichkeiten nicht ausreichen, benutzt man für die zusätzlich benötigten sechs Zahlzeichen die ersten Buchstaben des Alphabets, also

A = 10, B = 11, C = 12, D = 13, E = 14 und F = 15.

In der Frühzeit der Entwicklung elektronischer Rechner war auch die Verwendung des **Oktalsystems** (zur Basis 8) üblich, und es erfüllte den gleichen Zweck, wobei man jetzt nicht vier sondern *drei* Binärziffern zu einer Oktalziffer zusammenfasste. Es gab aber auch eine Wechselwirkung zwischen der Zahlennotation und der inneren Organisation der Rechner, und praktische

Gründe, die wir unten erläutern, führten schließlich zu einem Sieg des Hexadezimalsystems, so dass das Oktalsystem kaum noch verwendet wird.

Unsere Beispielzahl 137 hat in den oben genannten Basen folgende Darstellungen:

$$137_{10} = 10001001_2 = 211_8 = 89_{16}.$$

1.1.2 Umwandlung von Zahlendarstellungen zwischen verschiedenen Basen

Für Menschen ist immer noch das gewohnte Dezimalsystem am bequemsten; schließlich lernen wir in diesem System Zahlen erst verstehen. Es drängt sich deshalb die Frage auf, wie man Zahlen zwischen verschiedenen Notationssystemen übersetzen kann (und speziell, wie man Zahlen zwischen dem Dezimalsystem und dem Binärsystem am *effizientesten* umwandeln kann).

Um die Frage am allgemeinsten besprechen zu können, gehen wir aus von einer Zahl a , deren schriftliche Darstellung $a_k \dots a_0$ zur Basis n vorliegt und in die Darstellung $b_l \dots b_0$ zur Basis m umzuwandeln ist, wobei n und m natürliche Zahlen ≥ 2 sind.

Für die Anwendung in der Informatik ist es auch meistens der Fall, dass der „Rechner“, der die Umwandlung vorzunehmen hat, in einer der beiden Basen bequemer rechnen kann, als in der anderen — Menschen ziehen es vor, im Dezimalsystem zu rechnen, Maschinen rechnen am direktesten und am einfachsten im Binärsystem. Welche Basis für den Rechnenden bequemer ist wird beeinflussen, welche der möglichen Methoden er wählen wird.

Wenn man in der Basis n bequem rechnen kann, wird man in dieser Basis die Rechenschritte ausführen und folgende Methode verwenden: man dividiert wiederholt die gegebene Zahl (am Anfang a) durch m und man notiert sich den Quotienten und den Rest; der Quotient ist die „gegebene Zahl“ für die nächste Wiederholung, und man führt dies so oft aus, bis der Quotient 0 geworden ist; die Reste ergeben von rechts nach links gelesen die Ziffern der Darstellung von a zur Basis m .

Diese knappe Beschreibung ist vielleicht nicht sehr verständlich. Deshalb wiederholen wir die Beschreibung in einer etwas formaleren Sprache, die aber genau sagt, was wir meinen:

Vor Beginn der Berechnung setze man $q_0 := a$ und $i := 0$ und man wiederhole nun folgende Schritte:

1. Dividiere q_i durch m und nenne den Quotienten q_{i+1} und den Rest (auch wenn er 0 ist) b_i ;
2. Wenn $q_{i+1} = 0$ ist, höre man auf; sonst erhöhe man i um 1 und gehe zu Schritt 1.

Nachdem man aufgehört hat, bilden die berechneten Zahlen b_i die Ziffern der Darstellung $b_l \dots b_0$ von a zur Basis m .

Als Beispiel wandeln wir nach dieser Methode die Dezimalzahl 137 in die Basen 6 und 2.

Für die Umwandlung zur Basis 6:

$$\begin{array}{ll} 137 \div 6 = 22, & \text{Rest 5} \\ 22 \div 6 = 3, & \text{Rest 4} \\ 3 \div 6 = 0 \text{ (jetzt aufhören!)}, & \text{Rest 3} \end{array}$$

Wir lesen die ermittelten Reste in umgekehrter Reihenfolge und erhalten die Ziffern 345_6 .

Für die Umwandlung zur Basis 2:

$$\begin{array}{ll} 137 \div 2 = 68, & \text{Rest 1} \\ 68 \div 2 = 34, & \text{Rest 0} \\ 34 \div 2 = 17, & \text{Rest 0} \\ 17 \div 2 = 8, & \text{Rest 1} \\ 8 \div 2 = 4, & \text{Rest 0} \\ 4 \div 2 = 2, & \text{Rest 0} \\ 2 \div 2 = 1, & \text{Rest 0} \\ 1 \div 2 = 0 \text{ (jetzt aufhören!)}, & \text{Rest 1} \end{array}$$

Wir lesen die ermittelten Reste in umgekehrter Reihenfolge und erhalten die Ziffern 10001001_2 .

Übrigens, die oktale und hexadezimale Darstellung dieser Zahl erhalten wir aus der Binärdarstellung direkt durch Gruppierung von Ziffern:

$$\begin{aligned} 10001001_2 &= 10.001.001_2 = 1000.1001_2 \\ &= 2 \quad 1 \quad 1_8 = 8 \quad 9_{16} \end{aligned}$$

Obwohl das gerade besprochene Umwandlungsverfahren immer funktioniert, hat es den Nachteil, dass man dividieren muss, und Division ist für Menschen zumindest eine unangenehme Rechenoperation.

Für die Umwandlung aus anderen Basen in die **Binärdarstellung** gibt es eine andere bequeme Methode, die speziell für die Binärdarstellung als Ziel einfach anzuwenden ist, weil jede Binärziffer nur zwei Werte annehmen kann, 1 oder 0. Dieses Verfahren kommt ohne Divisionen aus.

Für diese Methode schreibt man alle Zweierpotenzen 2^k hin, die nicht größer als die zu umwandelnde Zahl a sind.

Von a zieht man jetzt die größte Zweierpotenz aus der Liste ab, und fährt mit der Differenz d fort. Von dieser zieht man die größte Zweierpotenz $\leq d$ ab, und wiederholt das Verfahren mit der neuen Differenz immer wieder, bis die Differenz 0 wird. Die hier benötigten Zweierpotenzen muss man nicht für jeden Schritt neu berechnen, sondern man liest sie aus der Anfangs erstellten Liste ab, aber man merkt sich, welche Listeneinträge man abzieht.

Weil das Doppelte eines Listeneintrags gleich dem nächsthöheren Listeneintrag ist, kann man keinen Eintrag in der Liste der Zweierpotenzen mehr als einmal beim Abziehen verwenden. Wenn man jetzt die Zweierpotenzliste in absteigender Reihenfolge liest und für jede abgezogene Stelle eine 1 und für jede übersprungene, nicht abgezogene Stelle eine 0 hinschreibt, erhält man die Binärdarstellung der Zahl a .

Das klappt, weil das Verfahren offensichtlich die Ziffern 1 durch Abziehen von Zweierpotenzen von a so ermittelt, dass die Summe der entsprechenden Zweierpotenzen wieder a ergibt.

Aber die Ziffernfolge von Einsen und Nullen als Binärzahl gelesen hat auch die Bedeutung, dass man die Summe der Zweierpotenzen zu bilden hat, die zu den Stellen in der Ziffernfolge gehören, an denen eine Eins steht. Und diese Summe ist a .

Man beachte, dass beim „Entziffern“ einer Binärzahl *Vielfache* von Zweierpotenzen nicht berücksichtigt werden müssen, weil die binären Ziffern ja nicht größer als 1 sein können.

Wir wandeln die Zahl 137 nach diesem Verfahren in eine Binärzahl um: die ersten Zweierpotenzen (die man beginnend mit 1 durch wiederholtes Verdoppeln erzeugen kann) sind

1, 2, 4, 8, 16, 32, 64, 128

und die Nächste, 256, ist schon größer als 137.

Jetzt führen wir das Verfahren des sukzessiven Abziehens aus:

Differenz	Zweierpotenz	abgezogen? (1 = ja)
137	128	1
9	64	0
	32	0
	16	0
	8	1
1	4	0
	2	0
	1	1

Danach ist die Differenz 0. Wir lesen in der linken Spalte die Ziffernfolge 10001001 ab, die wir als Binärdarstellung von 137 ja schon kennen.

Dieses Verfahren lässt sich auch für andere „Zielbasen“ m verwenden, mit folgenden Abwandlungen, die dafür sorgen, dass es keinen Vorteil und eher Nachteile gegenüber dem ersten Verfahren hat: man bildet wieder eine Liste von Potenzen von m , aber wenn man abziehen kann, muss man anstelle der einfachen Potenzen jeweils das *größte Vielfache* der Potenz abziehen, die sich noch mit nichtnegativem Ergebnis abziehen lässt. Welches Vielfache das war, notiert man neben der Potenz (in der letzten Spalte im obigen Schema). Dann kann man wieder aus der letzten Spalte die Basis m Darstellung von a ablesen.

Wir rechnen hierfür kein Beispiel.

Die beiden schon besprochenen Verfahren setzen voraus, dass man in der Quellbasis n gut und in der Zielbasis m schlecht rechnen kann, oder das man es zumindest vorzieht, in der Quellbasis zu rechnen.

Will man die gleichen Umwandlung in der Zielbasis m durchführen, so verwendet man andere Verfahren. Jetzt kann man die Quellzahl $a_k a_{k-1} \dots a_2 a_1 a_0$ direkt in ihre Bedeutung

$$a_k \times n^k + a_{k-1} \times n^{k-1} + \dots + a_2 \times n^2 + a_1 \times n + a_0 \quad (1.1)$$

übersetzen, indem man diesen Ausdruck in Basis m rechnerisch *ausrechnet*.

Es gibt aber einen Trick, der diese Berechnung verkürzt und der erspart, dass man die Potenzen von n explizit bestimmen muss. Dazu klammert man die einzelnen Faktoren n in den Potenzen in (1.1) aus diesem Ausdruck aus, d. h., man schreibt die Darstellung (1.1) um als

$$\left(\dots \left((a_k \cdot n + a_{k-1}) \cdot n \right) + \dots + a_1 \right) \cdot n + a_0.$$

Dies rechnet man aus, indem man auf folgende Weise Zwischenergebnisse erzeugt:

Das ursprüngliche Zwischenergebnis ist 0. Jetzt verarbeitet man die Ziffern a_i von links nach rechts; bei jeder Ziffer multipliziert man das aktuelle Zwischenergebnis mit n (diesen Schritt kann man sich bei der ersten Ziffer natürlich ersparen) und addiert die aktuelle Ziffer. Das liefert das neue Zwischenergebnis, und man wiederholt diesen Vorgang bis alle Ziffern verbraucht sind.

Das am Ende erzielte Zwischenergebnis ist dann der Wert der Zahl, den man in der Basis- m Darstellung erhält, wenn man diese Berechnung in diesem Zahlensystem durchführt.

Wir wandeln auf diese Weise die Zahl 10001001_2 in eine Dezimalzahl um:

Zwischenergebnis	$\times 2$	+	Ziffer
0	0		1
1	2		0
2	4		0
4	8		0
8	16		1
17	34		0
34	68		0
68	136		1
137			

Man sollte aber sagen, dass gerade für Umwandlungen aus dem Binärsystem die direkte Methode nicht viel schlimmer ist, denn es entfällt die Notwendigkeit, wie in anderen Basen erforderlich die Potenzen von n mit den Ziffern der Basis- n Zahl zu multiplizieren.

Wenn $n = 2$, so sind diese Ziffern entweder 0 (die entsprechenden Potenzen von 2 ignoriert man) oder 1 (die entsprechenden Potenzen von 2 gehen *einfach* in das Ergebnis ein). Man muss zwar zur Vorbereitung eine Liste von Potenzen von 2 erstellen (die man aber nach kurzer Zeit auswendig kennt), aber danach muss man nur noch die zu Ziffern 1 gehörenden Potenzen addieren, um die umgewandelte Zahl zu erhalten.

So ist

$$10001001 = 2^7 + 2^3 + 2^0 = 128 + 8 + 1 = 137,$$

und das ist doch noch ganz einfach.

1.1.3 Rechnen im Binärsystem

Wir interessieren uns für das Binärsystem, weil elektronische Rechenmaschinen sich am einfachsten mit bipolaren Bausteinen realisieren lassen und das Binärsystem deshalb das „natürliche“ Zahlensystem für solche Rechner ist.

Das Binärsystem hat noch den weiteren Vorteil, dass die Grundrechenoperationen in ihm viel einfacher auszuführen sind, als in anderen Zahlenbasen. Dieser Vorteil macht sich auch bei der Funktionsweise der Recheneinheiten von binär rechnenden Rechenmaschinen bemerkbar und wir wollen ihn deshalb näher erläutern.

Die Addition von zwei Summanden im Binärsystem funktioniert prinzipiell auf die gleiche Art, wie im Dezimalsystem. Man bildet die Summe stellenweise (in der Ziffernfolge) von rechts nach links, wobei Überträge in die nächst höhere Position zu berücksichtigen sind. Dabei ist von Vorteil,

dass die Additionstabelle im Binärsystem sehr klein ist und auch die Regel dafür, wann Überträge stattfinden, einfacher nachzuprüfen ist, als in anderen Zahlensystemen.

Hier ist die ganze Additionstabelle (für einzelne Ziffern):

$$0 + 0 = 0, \quad 0 + 1 = 1 + 0 = 1, \quad 1 + 1 = 10$$

Als Rechenvorschrift für die Stellen der Summe haben wir also:

An Stellen ohne Übertrag aus der vorherigen Stelle entsteht eine 0, wenn beide Summanden die gleiche Ziffer zeigen, und eine 1 sonst, und ein Übertrag in die nächste Stelle findet genau dann statt, wenn beide Summanden 1 waren.

An Stellen mit Übertrag aus der vorherigen Stelle entsteht eine 1, wenn beide Summanden die gleiche Ziffer zeigen, und eine 0 sonst, und ein Übertrag in die nächsthöhere Stelle findet genau dann statt, wenn mindestens ein Summand 1 war.

Diese Rechenregel ist auch mit elektronischen Bauteilen, wie wir im nächsten Kapitel sehen werden, nicht schwer zu realisieren.

Rechnende Menschen würden tatsächlich die Stellen der Summanden *hintereinander* abarbeiten; rechnende Maschinen können so gebaut werden, dass sie Teile der Berechnung *gleichzeitig* ausführen können und so die Rechenzeit verkürzen.

Die Subtraktion im Binärsystem kann nach entsprechend einfachen Regeln durchgeführt werden, auf die wir aber hier nicht näher eingehen wollen, weil sie von elektronischen Rechnern gar nicht angewendet werden. Elektronische Rechner *subtrahieren* nicht im eigentlichen Sinne (d. h., nicht nach den analogen Regeln zu denen, die wir für das Dezimalsystem in der Schule gelernt haben), sondern sie *addieren* den Minuend zum Negativen des Subtrahends. Die Details hängen davon ab, wie in der jeweiligen Maschine negative Zahlen dargestellt werden, und diese Frage wollen wir hier noch nicht eingehend besprechen. Aber der wichtige Punkt ist, dass anschließend keine Sonderimplementierung für die Subtraktion mehr nötig ist; die Addiereinheit kann ganz normal benutzt werden, um die Subtraktion zu Ende zu führen.

Also: Computer brauchen eine Addiereinheit, aber keine Subtrahiereinheit — stattdessen benötigen Sie nur ein Negierer, dessen Funktionsweise von der Darstellungsart für negative Zahlen abhängt, aber der auf jeden Fall sehr viel einfacher aufgebaut ist, als der Addierer es ist (oder ein Subtrahierer es wäre).

Eine Addiereinheit hat jeder Computer, und sie reicht aus, um die beiden ersten Grundrechenarten zu implementieren.

Aber auch Multiplikation und Division sind binär sehr einfach auszuführen und sehr einfach auf Rechnern zu implementieren. Frühe Rechner mit sehr begrenzten Ressourcen hatten oft keine „eingebauten“ Befehle für diese Operationen, aber sie liessen sich dann mit einem kurzen und einfachen Programm effektiv realisieren.

Wir wollen hier nur kurz erläutern, warum diese Operationen in binärer Arithmetik viel einfacher sind als im Dezimalsystem.

Wenn man zwei Dezimalzahlen multipliziert, muss man Produkte von Ziffern bilden (und zu diesem Zweck das gar nicht so kleine „kleine Einmaleins“ parat haben) und dann auch noch komplizierte Überträge berücksichtigen.

Wenn man binär multipliziert, muss man gar keine Produkte lernen oder parat haben, denn bei der Multiplikation auch langer Zahlen mit einzelnen Ziffern kann es nur um die Multiplikation mit 0 gehen (mit Ergebnis 0) oder um die Multiplikation mit 1, die als Produkt einfach die andere Zahl liefert.

Eine brauchbare und auch von einem elektronischen Rechner gut auszuführende Regel für die Multiplikation zweier großer Zahlen a und b lautet deshalb schlicht:

- Für jede Ziffer 1 in b schreibe eine Kopie von a hin, mit rechtem Ende ausgerichtet auf diese Stelle in b (das bewirkt man am besten, indem man entsprechend viele Nullen rechts an a anfügt);
- Addiere die gerade hingeschriebenen Kopien von a zusammen; das Ergebnis ist $a \cdot b$.

Für die Implementation auf einem Rechner oder in einem Computerprogramm würde man Zwischenergebnisse berechnen und das Zwischenergebnis vor Beginn der Berechnung zu 0 initialisieren. Dann liest man die Ziffern von b von rechts nach links ab, und bei jedem Übergang zur nächsten Ziffer fügt man eine 0 ans rechte Ende von a an.

Wenn man eine Ziffer 1 in b liest, addiert man das gegenwärtige a zum Zwischenergebnis. Wenn man eine Ziffer 0 in b liest, geht man zur nächsten Stelle, ohne das Zwischenergebnis zu verändern.

Das Endergebnis dieses Verfahrens ist das gesuchte Produkt $a \cdot b$.

Wir illustrieren diese Berechnung am Beispiel $13 \cdot 67 = 1101_2 \cdot 1000011_2$:

Die Zahl $b = 1000011_2$ hat Einsen an der niedrigsten oder 2^0 -er Stelle, an der nächst höheren oder 2^1 -er Stelle, und an der 2^6 -er Stelle. Deshalb ist das Produkt die Summe von $a = 1101$, von $11010 = a$ mit einer rechts

angehängten 0, und von $1101000000 = a$ mit sechs angehängten Nullen:

$$\begin{array}{rcl} 1101 \cdot 1000011 & = & \\ & + & 1101 \\ & + & 11010 \\ & + & \frac{1101000000}{1101100111} \end{array}$$

In Dezimal ist diese Zahl

$$\begin{aligned} 1101100111_2 &= 2^9 + 2^8 + 2^6 + 2^5 + 2^2 + 2^1 + 2^0 \\ &= 512 + 256 + 64 + 32 + 4 + 2 + 1 = 871_{10} \end{aligned}$$

und das ist tatsächlich 13 mal 67.

Bei der Division ist die Ersparnis an Komplexität des binären Rechnens gegenüber dem Rechnen im Dezimalsystem noch beeindruckender. Die binäre Division ist kaum aufwändiger als die Multiplikation, was man in keiner anderen Zahlenbasis behaupten kann.

Um das zu verdeutlichen wollen wir uns kurz daran erinnern, wie man die „lange Division“ im Dezimalsystem ausführt, und dafür einen Algorithmus aufstellen, der der Schulmathematik entspricht.

Es seien zwei natürliche Zahlen $a \neq 0$ und c gegeben und wir wollen c mit Rest durch a dividieren und rechnerisch den Quotienten b und den Rest r bestimmen (diese Zahlen sind bestimmt durch die Bedingung, dass $c = a \cdot b + r$, wobei $0 \leq r < a$). Hier das übliche Rechenverfahren aus der Schule, der Klarheit willen ein bisschen formal aufgeschrieben.

Übrigens, wenn $a > c$, dann ist der Quotient 0 und der Rest ist gleich c . In diesem Fall ist keine Berechnung erforderlich. Wir prüfen das natürlich vorweg und wenden das Rechenverfahren nur an, wenn $a \leq c$.

1. Schreibe c sauber auf ein Blatt Papier und schreibe a so darüber, dass beide Zahlen an ihren linken Enden aneinander ausgerichtet sind. Den linken Abschnitt von c , der bis zum rechten Ende von a reicht, nennen wir \tilde{c} .
2. Vergleiche a mit \tilde{c} .
 - A. Wenn a größer ist als \tilde{c} , schreibe eine 0 als nächste Ziffer des Quotienten b , und gehe zu Schritt 3.
 - B. Wenn a nicht größer ist als \tilde{c} , errate den (einstelligen) Quotienten d von \tilde{c} dividiert durch a , indem du den linken Teil von \tilde{c} , bis einschließlich der Stelle der linken Ziffer von a , durch die linke Ziffer von a teilst. Ist dieser Quotient ≥ 10 , nehme $d = 9$.

C. Berechne $d \cdot a$ und schreibe es direkt unter \tilde{c} , und vergleiche die beiden Zahlen.

- i.** Wenn $d \cdot a > \tilde{c}$, wische $d \cdot a$ wieder weg, vermindere d um 1 und kehre zurück zu Schritt 2C.
- ii.** Wenn $d \cdot a \leq \tilde{c}$, ziehe $d \cdot a$ von \tilde{c} ab und schreibe die Differenz unter $d \cdot a$, mache die Differenz zum neuen Wert von \tilde{c} und notiere d als nächst Ziffer des Quotienten b .

3. Hat c noch weitere Ziffern rechts von \tilde{c} ?

- A.** Wenn ja, füge die nächste Ziffer von c rechts an \tilde{c} an, verschiebe a um eine Stelle nach rechts, so dass a und das neue \tilde{c} wieder an ihren rechten Enden aneinander ausgerichtet sind, und kehre zu Schritt 2 zurück.
- B.** Wenn nicht, so ist b der gesuchte Quotient und \tilde{c} der Rest r (aber wenn b mit einer 0 beginnt, entferne man die führende 0).

Als Beispiel dividieren wir 871 durch 13 nach dieser Methode. Nach dem ersten Schritt haben wir als Rechenschema

$$\begin{array}{r} 13 \\ 871 : 13 = \end{array}$$

und $a = 13$, $\tilde{c} = 87$. Wir führen die Proben in Schritt 2Ci nicht im Rechenschema aus, sondern bemerken nur, dass der zuerst probierte Wert von $d = 8$ ist, dass auch $d = 7$ zu groß ist und dass wir erst beim dritten Durchgang durch Schritt 2C den richtigen Wert $d = 6$ erwisch haben.

Die Rechnung geht dann wie folgt weiter:

$$\begin{array}{r} 871 : 13 = 67 \\ 78 \\ \hline 91 \\ 91 \\ \hline 0 \end{array}$$

Auch bei der Bestimmung der zweiten Ziffer des Quotienten muss man drei Werte von d , beginnend mit 9, untersuchen, bevor man feststellt, dass 7 die richtige Ziffer ist. Natürlich ist es nicht überraschend, dass der Quotient 67 und der Rest 0 ist, da wir ja 871 gerade als Produkt von 13 und 67 ermittelt haben.

Sie erinnern sich vielleicht mit Schrecken an Ihre Schulzeit und die Mühe, die Sie damit hatten, diese Rechenmethode zu lernen.

Wenn man *binär* rechnet, wird alles viel einfacher, und vor allem die aufwändigsten und unangenehmsten Schritte müssen nicht mehr ausgeführt werden.

Man beachte, dass in allen Basen eine Ziffer 0 im Quotienten nur in der Situation von Schritt 2A vorkommt. Egal wie die Proben in Schritt 2C ausgehen, die Ziffer d wird keinesfalls 0, wenn man bis zu diesem Schritt gelangt.

Aber in der binären Arithmetik ist eine Ziffer, die nicht 0 ist, automatisch 1. Das heißt, dass wir in Schritt 2B nicht raten müssen und auch keine Proben wie in Schritt 2C machen müssen; d kann in dieser Situation nur 1 sein.

Und aus diesem Grund müssen wir $d \cdot a$ auch nicht berechnen; es ist automatisch gleich a und wir können in Schritt 2Cii gleich a unter \tilde{c} schreiben und abziehen.

Der ganze Algorithmus für die binäre Division hat jetzt folgende viel kürzere und vor allem viel einfachere Gestalt:

1. Schreibe c sauber auf ein Blatt Papier und schreibe a so *darunter*, dass beide Zahlen an ihren linken Enden aneinander ausgerichtet sind. Den linken Abschnitt von c , der bis zum rechten Ende von a reicht, nennen wir \tilde{c} .
2. Vergleiche a mit \tilde{c} .
 - A. Wenn a größer ist als \tilde{c} , schreibe eine 0 als nächste Ziffer des Quotienten b , und gehe zu Schritt 3.
 - B. Wenn a nicht größer ist als \tilde{c} , schreibe eine 1 als nächste Ziffer des Quotienten b , ziehe a von \tilde{c} ab und schreibe die Differenz darunter, und mache die Differenz zum neuen Wert von \tilde{c} .
3. Hat c noch weitere Ziffern rechts von \tilde{c} ?
 - A. Wenn ja, füge die nächste Ziffer von c rechts an \tilde{c} an, verschiebe a um eine Stelle nach rechts, so dass a und das neue \tilde{c} wieder an ihren rechten Enden aneinander ausgerichtet sind, und kehre zu Schritt 2 zurück.
 - B. Wenn nicht, so ist b der gesuchte Quotient und \tilde{c} der Rest r (aber wenn b mit einer 0 beginnt, entferne man die führende 0).

Als Beispiel wiederholen wir die Division von $871 = 1101100111_2$ durch

$13 = 1101_2$, diesmal im Binärsystem. Hier die ganze Berechnung:

$$\begin{array}{r}
 1101100111 : 1101 = 1000011 \\
 \underline{1101} \\
 10011 \\
 \underline{1101} \\
 1101 \\
 \underline{1101} \\
 0
 \end{array}$$

Wir fassen unsere bisherigen Überlegungen kurz zusammen:

- Elektronische Rechner lassen sich am einfachsten, am effizientesten und am billigsten aus *bipolaren Elementen* zusammenbauen, die gerade zwei elektrische Zustände annehmen können (und diese Behauptung bleibt richtig, auch wenn heutzutage in Wirklichkeit die bipolaren Bausteine im wesentlichen durch Photolithographie massenproduziert werden, und zwar nicht als Einzelteile, sondern schon mitsamt ihren elektrischen Verbindungen untereinander — die Grundelemente einer solchen *integrierten Schaltung* sind nach wie vor die konstituierenden bipolaren Elemente, die auch unabhängig voneinander geschaltet werden können).
- Das Rechnen im Binärsystem gestaltet sich *viel* einfacher, als in anderen Zahlensystemen, und lässt sich aus diesem Grund auch viel leichter und unkomplizierter auf Maschinen implementieren, als zum Beispiel das Rechnen im Dezimalsystem. Diese letzte Aussage gilt natürlich um so mehr, wenn die Maschine so konstruiert ist, dass sie konstruktionsmäßig binäre Daten am besten darstellen kann.

Diese beiden Punkte sind der Grund dafür, dass praktisch alle modernen Rechner Daten intern binär speichern und verarbeiten und auch Zahlenberechnungen binär durchführen.

Jetzt, wo das klar ist, sollten wir zwei wichtige Grundbegriffe einführen, ohne die man sich kaum über Rechner unterhalten kann; es handelt sich um die „räumlichen Maßeinheiten“ der Rechnertechnik.

Die kleinste Dateneinheit der Informationstheorie, die Atome, aus denen jede in einem Rechner verarbeitete Information besteht, wird gegeben durch den Zustand eines einzelnen bipolaren Elements, also abstrakt gesehen durch eine einzelne binäre Ziffer 0 oder 1. Diese „Dateneinheit“ wird ein **Bit** genannt, die Abkürzung oder die zusammengestauchte Version der englischen Bezeichnung **binary digit**.

Ein Computer enthält natürlich nur eine endliche (wenn auch oft sehr große) Anzahl von bipolaren Elementen, oder in der abstrakteren Sprache der Informationstheorie, die nicht auf die materiellen Bestandteile direkt absieht, von Bits, und alle Daten, Programme (also die Beschreibung der auszuführenden Operationen), Zwischenergebnisse, Steuermerkmale müssen in diesen endlich vielen Bits untergebracht und dargestellt werden. Damit diese Daten und andere Merkmale auf eine geordnete Weise verarbeitet werden oder zusammenwirken können, müssen die Bits auf eine gewisse Weise organisiert sein und in eine gewisse Struktur eingebunden sein.

Um das erläutern, nehmen wir einmal an, der Rechner soll die Zahlen 13 und 67 miteinander multiplizieren, das Ergebnis als ein Preis in eine Rechnung einfügen, die Rechnung adressieren und ausdrucken für den Versand an den Empfänger. Schon beim ersten Teil dieser Folge von Operationen gibt es ein Problem: wo in den Millionen oder Milliarden Bits, die es auf dem Rechner gibt, findet man die Zahl 13 dargestellt? Wenn diese Zahl ein Operand einer Multiplikation ist und der Rechner weiss, wo die Zahl in seinem Speicher beginnt, woher weiss er, wann die Zahl endet? Wir haben gesehen, dass $13_{10} = 1101_2$, aber woher weiss der Rechner, dass die Bitfolge genau an der letzten 1 endet und nicht irgendwie weiter geht: 11010010...?

Das gleiche Problem tritt auf bei der Behandlung der Rechnung, die auch irgendwie mit Bits im Rechner dargestellt wird. Wo liegen diese Bits? Wo endet die Darstellung der Rechnung und beginnt die Darstellung anderer Daten oder momentan bedeutungsloser Bits?

Im Wesentlichen laufen diese Fragen darauf hinaus, dass eine unstrukturierte Folge von Bits nicht eindeutig interpretierbar sein kann und deshalb keine verarbeitbare Information darstellen kann.

Die Lösung dieses Problems besteht darin, dass die Bits entweder in fest bestimmten Gruppen ansprechbar sind, so dass eine (von dem Rechner darstellbare) Zahl ganz in eine dieser Gruppen untergebracht werden kann und die wohlbestimmte Bitfolge in dieser Gruppe die Zahl darstellt, oder dass manche kurze Bitfolgen als Datenbegrenzer fungieren, und nur in dieser Funktion in Daten vorkommen dürfen (etwa in der Art, wie „Gänsefüßchen“ ein Zitat begrenzen und kenntlich machen, oder wie der Punkt das Ende eines geschriebenen Satzes markiert). Möglich ist auch die Methode, die Länge (in Bits oder größeren Einheiten) der zu verarbeitenden Daten direkt im auszuführenden Befehl oder am Anfang der Daten anzugeben.

Obwohl alle drei Methoden in Rechnern implementiert wurden und alle für gewisse Datentypen ihre besondere Eignung haben, ist es im allgemeinen so, dass sowohl der Speicherzugriff, um Daten für die Recheneinheit zu holen, wie auch der interne Aufbau der Recheneinheit eine bestimmte Datenlänge in Bits bevorzugt, in dem Sinne, dass der Speicher fest in „Bitgruppen“ oder

Zellen einer bestimmten Länge unterteilt ist und dass Speicherzugriffe immer auf ganze Gruppen zugreifen, aber nicht in einem einzigen Zug Daten ansprechen können, die die Grenzen zwischen zwei Gruppen überschreiten. Ferner haben die internen Speicher (**Register** genannt) der Recheneinheit eine bestimmte Länge und Rechenoperationen wirken fast immer auf ganze Register.

Diese Bitgruppen, die der Rechner wegen seiner internen Organisation direkt ansprechen kann, heißen **Wörter**. Rechnerbefehle, die auf bestimmte Wörter im Speicher operieren sollen, müssen eine Möglichkeit haben, genau das richtige Wort anzusprechen; die Lage dieses Wortes im Speicher wird durch eine Binärzahl beschrieben, deren Muster von Einsen und Nullen bei der Ausführung der Operation bewirkt, dass genau die gewünschte Speicherstelle auf eine Schreib- oder Leseoperation reagiert, und diese der Speicherstelle zugeordnete Zahl nennt man sinnvollerweise die **Adresse** des Wortes.

Die Wortlänge eines Rechners wird von vielen Faktoren bestimmt, darunter den Stand der Technik und die Wirtschaftlichkeit oder praktische Realisierbarkeit von Recheneinheiten, die Daten einer bestimmten Länge in einem Zug bearbeiten können, die Größe des Speichers des Rechners (weil diese die Größe des „Adressraums“ und somit die Länge von Adressen bestimmt, und je nach dem Aufbau des Rechners Adressen in ein Wort hineinpassen müssen), die Struktur der Daten oder der Typ von Berechnungen, für die der Rechner gedacht ist (müssen sehr große Zahlen oder sehr genaue Bruchzahlen schnell verarbeitbar sein oder nicht?) und viele Aspekte mehr.

Aus diesem Grund variiert die Wortlänge historischer Rechner in einem weiten Bereich zwischen 1 Bit und 64 Bit. Besonders die frühen Rechner hatten oft "krumme" Wortlängen wie 36 Bit oder 18 Bit, weil diese Länge noch realisierbar war und ausreichte, eine oder sogar zwei Adressen von Operanden in einem einzigen Wort unterzubringen.

Im Laufe der Zeit hat sich aber eine besondere Datenlänge als „Standard“ herausgebildet: nämlich 8 Bit. Um zu erklären, warum gerade diese Bitlänge besonders „praktisch“ ist, sollten wir kurz über die Darstellung von nichtnumerischen Daten im Rechner reden — die wichtigsten und häufigsten solcher Daten sind Textdaten. Weil es auf einem Rechner im allgemeinen keine andere Datenform als Bitfolgen gibt, müssen auch Texte als Bitfolgen „kodiert“ werden, und das macht man, in dem man jedes Zeichen, das in den Texten vorkommen darf, durch eine bestimmte Bitfolge darstellt. Diese Bitfolgen haben in der Regel für alle Zeichen die gleiche Länge. Längere Texte werden durch Aneinanderreihung der Bitfolgen für ihre Zeichen dargestellt, und natürlich wünscht man, dass die Wortlänge des Rechners ein Vielfaches der Bitlänge eines Zeichens ist, damit kurze Textabschnitte ohne Raumverschwendung genau in ein Wort passen und es ausfüllen.

Weil ein großer Teil der Rechnerentwicklung in den Vereinigten Staaten stattgefunden hat, sind die Rechner in erster Linie darauf abgestimmt, mit Texten in *englischer* Sprache umzugehen.

In der Frühzeit der Rechnergeschichte, als Ressourcen sehr knapp waren, hat man sich mit Großbuchstaben zufrieden gegeben, von denen die englische Sprache 26 benutzt. Hinzu kommen die zehn Ziffern und einige Satzzeichen (Punkt, Komma, Leerzeichen usw.), sowie gewisse Steuerzeichen (zum Beispiel für den Wagenrücklauf und den Zeilenvorschub, damit ein Drucker angewiesen werden kann, den Schreibkopf wieder auf den Zeilenanfang zurückzuschieben und das Papier um eine Zeile weiter zu rücken, damit der neue Text in eine frische Zeile auf dem Papier gedruckt wird und die letzte Druckzeile nicht einfach überschreibt. Diese Steuerzeichen sind Überbleibsel aus der Zeit der Fernschreiber und haben tatsächlich die Funktion gehabt, diese automatischen Telegraphenempfänger zu steuern. Fernschreiber wurden in der frühen Rechnergeschichte oft als Drucker verwendet.

Die Gesamtanzahl der benötigten Zeichen ist auf jeden Fall mehr als 32, so dass 5 Bit für die Zeichenlänge nicht ausreichen. Wenn man auch Kleinbuchstaben getrennt von den Großbuchstaben kodieren will, braucht man insgesamt mehr als 64 Zeichencodes und somit eine Zeichenlänge von mindestens 7 Bits.

Für den amerikanischen Gebrauch sind 7 Bits (ausreichend für $2^7 = 128$ Zeichen) aber genug, und es wurde früh in der Rechnerentwicklung ein 7-Bit Zeichenkodierungsstandard mit dem Namen *American Standards Code for Information Interchange* (oder kurz ASCII) aufgestellt, der alle Zeichen auf amerikanischen Schreibmaschinen darstellen kann und der sehr weit verbreitet ist. Die ersten 32 ASCII Zeichen (mit Codes $0-31_{10} = 1F_{16}$) sind Steuerzeichen, dann kommt das Leerzeichen, das Ausrufungszeichen und weitere Satzzeichen bis Zeichen $2F_{16} = 47_{10}$, dann die Ziffern und weitere Satzzeichen, ab Zeichen $41_{16} = 65_{10}$ die Großbuchstaben (und ein paar Satzzeichen und Klammern), ab Zeichen $61_{16} = 97_{10}$ dann die Kleinbuchstaben, Satzzeichen und Klammern, und als Letztes das Steuerzeichen DEL=Löschen.

Sieben Bits ist eine krumme Zahl (die sich zum Beispiel nicht in handhabbare Gruppen unterteilen lässt für die Zahlendarstellung im Oktal- oder Hexadezimalsystem), und auch aus Gründen der Übertragungssicherheit fand man schnell einen Grund, ein achttes Bit hinzuzufügen, ein so genanntes **Paritätsbit**, das bei jedem Zeichen so gewählt wird, dass die Gesamtanzahl der Einsen im Zeichencode immer gerade ist (oder immer ungerade ist). Später wurde der ASCII Code erweitert, um die Sonderzeichen in den europäischen Sprachen zu erfassen.

Auf jeden Fall ergab sich dadurch ein Zeichendarstellungssystem, in dem jedes Zeichen 8 Bits beanspruchte. Diese Größe ist gleichzeitig geeignet für

die direkte Darstellung von Dezimalzahlen, falls man das wünscht. Da es mehr als 8 Dezimalziffern gibt, braucht man 4 Bits, um sie alle darzustellen, und 8 Bits reichen genau aus, um zwei Dezimalziffern wiederzugeben.

Auch die 8-Bit Zeichen reichen bei weitem nicht aus, um die Zeichen *aller* gängiger Schriftsprachen der Welt darzustellen (denken Sie etwa an Chinesisch oder Koreanisch). Aus diesem Grund wurde in den letzten Jahren ein 16-Bit Codierungssystem namens UNICODE eingeführt, das das ASCII System als seine ersten 256 Zeichen beinhaltet, aber auch exotische Alphabeten wie das kyrillische, das hebräische oder das arabische darstellen kann, sowie die gebräuchlichsten fernöstlichen Zeichen.

Wir sehen, dass für die sehr häufig bearbeiteten Textdaten eine Grundeinheit von 8 Bits sehr praktisch ist, und wegen der besonderen Wichtigkeit dieser Datenlänge gibt es dafür einen eigenen Namen: 8 Bits machen 1 **Byte** aus. Für die Darstellung von Zahlen in Dezimalschreibweise reichen halbe Bytes (als 4 Bits) aus; diese Einheit nennt sich ein **Nibble**. Diese Namen bilden ein englisches Wortspiel: Byte lehnt sich an Bit an, hat aber zur Unterscheidung ein *y* anstelle des *i*, wird aber so ausgesprochen, als wäre es ein *Bite*, zu deutsch „Bissen“. Das Verb *nibble* bedeutet „nagen“ und als Hauptwort ist *nibble* also ein „kleines Bissen“, sehr passend für die Hälfte eines Bytes.

Wegen der Nützlichkeit vom Byte als eine Speichergröße, die an die Darstellung von Schriftzeichen sehr gut angepasst ist, wurden schon sehr früh Rechner gebaut, die Daten bytewise ansprechen und verarbeiten konnten. Im Laufe der Zeit wurde es zur Regel, dass Rechner ihren Speicher in Bytes unterteilten und ein ganzes Vielfaches eines Bytes als ihre Wortlänge hatten. Noch spezieller wurde es außerdem üblich, eine Wortlänge zu haben, die eine Potenz von 2 war, also entweder 8 Bit (die allerersten PCs), 16 Bit, später 32 Bit und heute bis zu 64 Bit.

Das Fassungsvermögen von Datenspeicher aller Art (der interne Speicher der PCs, aber auch Disketten, Festplatten, CD-ROMs, DVDs, Speicherkarten für Digitalkameras, und USB Sticks) wird gewöhnlich in Byte oder genauer, weil 1 Byte nach heutigen Maßstäben eine winzige Dateneinheit ist, in Kilobyte (KB), Megabyte (MB), Gigabyte (GB) oder sogar Terabyte (TB) gemessen.

Wichtig! Diese Präfixe des metrischen Systems haben in der Computertechnik nicht die übliche Bedeutung, denn Datenspeichergößen sind meistens Zweierpotenzen oder nicht allzugroße Vielfache von Zweierpotenzen — das liegt daran, dass Datenspeicher vom Computer adressiert werden muss und für die Adressen eine bestimmte Bitzahl zur Verfügung steht, so dass der adressierbare Speicherraum automatisch eine Zweierpotenz ist.

Wegen dieser Besonderheit bezeichnen die Vorsätze „Kilo“, „Mega“ usw.

nicht wie sonst üblich Zehnerpotenzen, sondern ihnen nahe liegende *Zweier*potenzen.

Ein Kilobyte ist $2^{10} = 1024$ Byte, ein Megabyte ist $2^{20} = 1048576$ Byte, und ein Gigabyte beträgt $2^{30} = 1073741824$ Byte.

Anders verhält es sich allerdings, wenn man Übertragungsgeschwindigkeiten über Datenkanäle oder Datenleitungen angibt; hier ist die Grundeinheit Bit/Sekunde und es gibt keinen natürlichen Grund, warum nur Zweierpotenzvielfache davon sinnvoll sind. Deshalb haben die metrischen Präfixe hier die übliche Bedeutung; zum Beispiel, eine DSL Internetverbindung mit einer theoretischen Datentransferrate von 16 Mbit/Sekunde kann im Idealfall 16.000.000 Bit/Sekunde übertragen (in Wirklichkeit wird wegen Störungen und Rauschen diese maximale Geschwindigkeit aber nie erreicht).

In diesem Abschnitt ging es primär um die Motivierung und die Einführung der eben genannten Maßeinheiten, und wir werden später zu den hier kurz angerissenen Themen zurückkehren und weitere technische Einzelheiten erläutern.

1.2 Die Organisationsstruktur eines Rechners und die funktionellen Bestandteile der Computerarchitektur

Unser Ziel im ersten Teil der Vorlesung ist eine detaillierte Diskussion der verschiedenen Bestandteile eines funktionierenden Rechnersystems, wobei das Wort „Bestandteile“ nicht nur im physikalischen Sinn von elektronischen Bauteilen und Funktionseinheiten zu verstehen ist, die man mit dem Auge wahrnehmen kann, wenn man das Gehäuse eines Computers aufmacht und sich die darin befindlichen Platinen, Datenspeicherlaufwerke und dergleichen ansieht. Auch die Software, die den Rechner bedienbar macht, zählt dazu — jeder, der mit Computern Erfahrung hat, weiss, dass der Absturz des Betriebssystems genau so hartnäckig das Starten des Rechners verhindern kann, wie der Ausfall eines elektronischen Bauteils; nur die Reparatur ist vielleicht billiger.

Diese Bestandteile wollen wir also im Einzelnen untersuchen, aber bevor wir damit beginnen, macht es Sinn, in der Einführung einen Blick von außen auf das Zusammenwirken dieser Bestandteile zu werfen. In Analogie zu einem Auto wollen wir einen Überblick darüber geben, wie der Motor, das Getriebe, das Fahrwerk, die Karosserie und die Elektrik und Elektronik eines Autos zusammenwirken und zum fahrbereiten Fahrzeug am Straßenrand zusammengesetzt sind, bevor wir ins Detail gehen und diese einzelnen

Baugruppen in ihren Varianten näher beschreiben.

Das prägende Merkmal der Gesamtstruktur eines Rechners ist der enorme Unterschied zwischen den Grundfähigkeiten einer Maschine und der Erlebniswelt eines Menschen. Die Transistoren in einem Rechner sind nicht viel mehr als Schalter und können sich nur ein- und ausschalten, oder sich über Kupferleitungen nur auf ganz einfache Art gegenseitig beeinflussen und steuern — schon die Geometrie der sich kreuzenden Leitungsbahnen setzt der Kompliziertheit der Verbindungen eine Grenze. Trotzdem erwartet der Mensch, der vor seinem Laptop sitzt, dass jeder Mausklick intelligente und komplexe Wirkungen zeigt, und dass das Gerät sich fast auf seiner Stufe mit ihm „unterhält“.

Die große Lücke zwischen diesen beiden Welten lässt sich nur durch eine aufeinanderbauende Hierarchie von Zwischenebenen oder Schichten auffüllen; jede dieser Schichten (bis auf die unterste) interagiert mit der nächst tieferen und ergänzt sie durch einen kleinen und deshalb wirtschaftlich zu realisierenden Satz von zusätzlichen Fähigkeiten, bis schließlich die oberste Schicht eine mit jeder neuen Rechnergeneration bequemer werdende Oberfläche für die Bedienung des Rechners durch seine Benutzer (im Gegensatz zu seinen Entwicklern) bietet.

Die unterste Stufe dieser Hierarchie von Ebenen wird gebildet durch die Elektronik des Rechners mit seiner „Verdrahtung“, und diese Stufe bildet die eigentliche *Maschine*, die alles, was der Rechner kann, letztendlich ausführt. Die Grundfähigkeiten dieser Maschine sind aber sehr begrenzt; sie kann nur sehr einfache Operationen ausführen, die wir unten etwas näher beschreiben werden. Diese Grundoperationen und ihre Kombinationen bilden eine Art ***Sprache*** L0, deren Elemente gewisse Bitfolgen sind (die ***Maschinenbefehle***), die die Maschine mit ihrer Elektronik entziffern kann zu auszuführenden Operationen oder Befehle. Die Grundmaschine, die diese Sprache „versteht“, nennen wir M0.

Die Maschine M0 ist wirtschaftlich zu bauen, aber interessante Operationen (wie die Addition zweier Zahlen) sind in ihrer Maschinensprache L0 nicht direkt ausdrückbar, sondern nur durch komplizierte und lange Folgen von einfacheren Operationen. Menschen sind kaum in der Lage, fehlerfrei und mit vertretbarem Aufwand in dieser Sprache zu programmieren. Deshalb erfindet man eine bessere, leistungsfähigere Sprache L1, die interessantere und unmittelbar nützlichere Operationen ausführen kann, und versucht die Maschine M0 zu einer Maschine M1 zu erweitern, die L1 versteht und ausführen kann.

Die Erweiterung kann durch Hardware geschehen, d. h., durch zusätzliche Bausteine, die in der Lage sind, gewisse Folgen von L0 Befehlen auszuführen, wenn Sie mit einer Bitfolge gefüttert werden, die auf irgendeine Weise ko-

diert, dass diese bestimmte Folge von L0 Befehlen gewünscht wird. Die neue Maschine M1 ist stärker als M0 aber baut auf M0 auf.

Es kann aber auch sein, dass es nicht praktikabel ist, mit elektronischen Bauteilen M0 zu einer L1-fähigen Maschine M1 „aufzumöbeln“. In diesem Fall kann man aber trotzdem so tun, als gäbe es die Maschine M1, aber man ersetzt die nicht zu realisierende Elektronik für ihre Implementierung durch ein Computerprogramm, also durch eine in einem Datenspeicher abgelegte feste Folge von L0-Befehlen, die in der Lage ist, ein Programm aus L1-Befehlen zu lesen und seine gewünschte Wirkung in L0 zu bewerkstelligen.

Dafür gibt es zwei Methoden: man kann entweder jeden L1-Befehl *ersetzen* durch eine äquivalente Folge von L0-Befehlen, d. h., das L1-Programm in L0 **übersetzen** und dieses anschließend vielleicht sogar vereinfachen und optimieren, um ein reines L0-Programm zu erhalten, das genau das macht, was das L1 Programm hätte tun sollen (diese Methode nennt sich die **Übersetzungsmethode**); oder man kann ein L0 Programm schreiben, dass einzelne L1-Befehle lesen und „verstehen“ kann, in dem Sinne, dass es zu jedem einzelnen L1-Befehl die zu bearbeitenden Daten genauso verändert, wie der L1-Befehl es getan hätte — diese Methode nennt sich **Interpretierung** und das L0 Programm, das sie ausführt, ein **Interpreter**.

In beiden Fällen laufen dann in Wirklichkeit L0 Programme, wenn ein L1 Programm ausgeführt wird, aber bei Übersetzung wird für jedes L1 Programm zunächst *für das ganze Programm* ein spezielles L0 Programm erstellt mit der gleichen Wirkung, und später kommt nur diese übersetzte Version auf M0 zur Ausführung; während bei Interpretation bei jeder Ausführung das L1 Programm gelesen wird und Befehl um Befehl durch ein *auf den einzelnen Befehl zugemünztes* und ihn nachäffendes L0 Programm vorgetäuscht wird, dass der L1 Befehl zur Wirkung kommt. Die erste Methode ist schneller und in der endgültigen Ausführung effizienter, aber erfordert mehr Vorbereitung und mehr Aufwand; die zweite Methode lässt sich einfacher realisieren, aber die Interpretation der einzelnen L1 Befehle kostet viel Zeit, die für das Verstehen der Befehle aufgewendet werden muss, und eine große Zusatzlast neben der eigentlich gewünschten Wirkung der Befehle.

Letztendlich werden Praktikabilitäts- und Wirtschaftlichkeitsgründen dafür ausschlaggebend sein, nach welcher der genannten Methoden L1 implementiert wird. Für den *Anwender* von L1 ist es aber völlig belanglos, wie das gemacht wird, und ob die L1 Maschine M1 tatsächlich existiert oder nur von Software vorgetäuscht wird — Hauptsache für ihn ist, dass er die bequemeren L1 Befehle verwenden kann und dass er sich mit L0 nicht herumschlagen muss. Er ist dann eben nur ein Benutzer einer L1 Maschine M1, und muss noch nicht einmal wissen, ob es sich um eine physikalische Maschine handelt oder um eine durch Software implementierte scheinbare Maschine. Aus die-

sem Grund werden wir M1 eine *virtuelle Maschine* nennen — sie stellt sich dem Programmierer wie eine richtige Maschine dar, auch wenn sie es nicht ist.

Auch die virtuelle Maschine M1 wird in der Regel nicht sehr bequem sein, und deshalb wird man die Erweiterung wiederholen, um eine neue, bessere virtuelle Maschine M2 mit einer bequemerer Sprache L2 zu bauen; darauf kann man dann weitere Maschinen M3, ..., Mn mit Sprachen L3, ..., Ln aufbauen, bis schließlich die Letzte Mausklicks auf Ikonen oder Spracheingabe versteht und alle Wünsche für eine Luxusbedienung erfüllt.

Die Maschine Mn wird man sicher nicht in L0 konstruieren wollen! Deshalb sind einige Zwischenstufen erforderlich, damit die einzelnen Erweiterungen überblickbar, einfach und praktikabel bleiben, sowohl für die Konstruktion durch Elektronik, wenn das geht (meistens nur in den untersten Stufen), oder für die Realisierung durch Software, die in der Sprache der vorherigen Stufe noch handhabbar ist.

Ein wichtiges Prinzip ist aber, dass die Fähigkeiten der verschiedenen Stufen von Bedeutung sind, aber nicht die Art, wie diese Fähigkeiten letztendlich implementiert werden. Für die Gesamtwirkung des Rechners ist es völlig belanglos, welche Verbindungen zwischen Schichten oder welche Teile des Aufbaus durch Hardware und welche durch Software realisiert werden, und ob die Software als Übersetzer oder als Interpreter funktioniert.

Oft werden bei der Weiterentwicklung eines Rechnermodells die Realisierung von Teilen der Struktur, ohne ihren Beitrag zur Gesamtstruktur zu verändern, von Software in Hardware geändert oder umgekehrt. Kurz gesagt:

Hardware und Software sind äquivalent und in ihrer funktionellen Wirkung austauschbar.

Zum Beispiel konnten die ersten Intel Recheneinheiten zwar addieren und subtrahieren, aber nicht multiplizieren oder dividieren. Die mit diesen Zentraleinheiten gebauten Computer waren sehr wohl in der Lage, zu multiplizieren und zu dividieren (denn schon einfache Anwendungen verlangen das); aber jede Anwendung, die diese Fähigkeiten benötigte, musste eine kleine Programmschleife durchlaufen, um Produkte und Quotienten zu berechnen.

In der nächsten Generation wurden Multiplikations- und Divisionsbefehle in der Hardware der Recheneinheit realisiert, aber für Berechnungen mit *Gleitkommazahlen* (also mit reellen Zahlen verschiedener Größenordnungen mit einer festen Anzahl von signifikanten Stellen) musste wieder Software bemüht werden. Die heutigen Pentium Rechner können von sich aus Gleitkommaberechnungen ausführen.

Ähnlich unwichtig ist der Unterschied zwischen den zwei Arten von Softwareimplementation, wie wir gesehen haben. In der Regel werden Computersprachen mit einer starken und komplizierten auf bestimmte Datentypen angepassten inneren Struktur interpretiert, während allgemeinere Sprachen, die eine bequemere Benutzung der Grundfähigkeiten eines Rechners ermöglichen, leichter übersetzt werden können. Aber beide Methoden können kombiniert werden: dann wird ein Programm in einer höheren Sprache zunächst in eine einfachere *Zwischensprache* übersetzt, die dann interpretiert wird (wobei dies dann viel einfacher ist, als wenn man die ursprüngliche höhere Sprache interpretiert hätte).

Nach diesem noch sehr allgemeinen Überblick über den stufenweisen Aufbau eines Rechners wollen wir die einzelnen Stufen eines typischen modernen Computers ein bisschen genauer charakterisieren, denn sie haben schon ihre Eigenheiten und heben sich in wesentlichen Merkmalen voneinander ab. In der Automobilanalogie betrachten wir jetzt den Motor, die Bremsen, die Stränge elektrischer Leitungen, usw., aber es geht immer noch um den äußeren Eindruck — wir bauen sie nicht auseinander!

Die allerunterste Stufe der Hierarchie von Ebenen im Aufbau eines Computers wird gebildet durch einfache **logische Bausteine** — das sind Funktionseinheiten, die in der Lage sind, elementare logische Verknüpfungen zwischen ihren Eingangssignalen zu bilden und den Wert an ihrem Ausgang anzuzeigen; diese Signale haben zwei stabile Zustände und können deshalb durch ein Bit dargestellt werden. Mit **logischen Verknüpfungen** sind Operationen gemeint wie das UND der Eingangsbits (der Ausgang ist genau dann 1, wenn alle Eingänge 1 sind) oder das ODER der Eingangsbits (der Ausgang ist genau dann 1, wenn mindestens eines der Eingänge 1 ist), oder beliebige andere Zuordnungen, die auf eine festgelegte Weise jeder möglichen Kombination von Eingangsbits einen bestimmten Wert 0 oder 1 als Ausgang zuordnen.

Solche auf Bitwerte wirkende Operationen nennen wir **Boolesche Operationen** (nach *George Boole*, 1815–1864). Obwohl die wichtigsten Beispiele tatsächliche einfache Verknüpfungen aus der Logik ausdrücken (wie die Beispiele, die im letzten Absatz genannt wurden), ist *jede* bitwertige Funktion von n bitwertigen Variablen (n eine beliebige aber feste endliche Zahl) eine Boolesche Funktion (und kann durch einen logischen Baustein realisiert werden), auch wenn die Funktion nicht durch einen kurzen logischen Ausdruck beschreibbar ist.

Für die Mathematiker unter den Lesern: wenn wir B als den Bitraum

$$B := \{0, 1\}$$

definieren, dann ist eine n -stellige Boolesche Funktion eine beliebige mengen-

theoretische Funktion

$$f: B^n \longrightarrow B.$$

Die logischen Bausteine, auch **Gatter** genannt, berechnen meist einfache Boolesche Funktionen von einer oder zwei Variablen (d.h., n ist meistens klein), und sie können aus wenigen Transistoren aufgebaut werden. Diese Gatter bilden die allereinfachste Stufe der Kontroll- und Recheneinheiten eines Computers, die Stufe oder Ebene der **digitalen Logik**.

Durch Kombination von logischen Gattern kann man nun alle wesentlichen Bestandteile eines Computers aufbauen, darunter die Speicher (1-Bit Speicher, die dann zu größeren Speichern aufgebaut werden können) und die Bestandteile der Recheneinheit. Zu den letzteren gehören

Register: das sind schnelle Speicher, auf die die Recheneinheit direkt zugreifen kann, um Rechenoperationen auf ihren Inhalt auszuführen, die aber auch bei der Adressierung des Hauptspeichers benutzt werden können und somit von den Einheiten angesprochen werden, die Daten zwischen der Recheneinheit und dem Hauptspeicher bewegen;

Shifter: Funktionseinheiten, die Daten in den Registern um eine Bitstelle nach rechts oder nach links verschieben können;

Invertierer, die die Bits in einem Register umkehren können;

die ALU oder „Arithmetic Logic Unit“, die einige sehr einfache Grundrechenoperationen auf Register ausführen kann, und darunter das bitweise UND, das ODER, und die Summe von zwei Wörtern oder Wortteilen berechnen kann. Welche Operation ausgeführt wird, wird durch eine kleine Anzahl von Steuerbits in den Eingängen bestimmt.

Die aus diesen und weiteren benötigten Bestandteilen (Datenleitungen, Taktgeber, Zeiger auf Speicherstellen) aufgebaute zweite Ebene stellt die eigentliche *Maschine* dar, die physikalisch existiert und in ihrer Funktion die Rechenoperationen ausführt.

Das Problem ist, dass diese Maschine M1 nur sehr sehr einfache Befehle ausführen kann: sie kann verschiedene Register oder Recheneinheiten mit dem Datenbus verbinden, sie kann bestimmen, in welche Richtung Daten über diese Verbindung bewegt werden, und sie kann auswählen, welche Recheneinheit eine Operation ausführen darf, und um welche der oben genannten sehr einfachen Operationen es sich handelt. Zwar ist auf dieser Ebene alles da, was erforderlich ist für das Funktionieren des Rechners insgesamt, aber auf so elementarem Niveau, dass man für den menschlichen Anwender sinnvolle Operationen wie die Addition zweier Registerinhalte mit Abspeichern

des Ergebnisses, wie die Bewegung eines Speicherwortes an eine andere Stelle im Speicher, oder wie das Hochzählen eines Zählers nicht direkt ausführen kann, sondern nur die einfachsten Teilschritte einer solchen Operation. Deshalb nennt man diese Stufe die *Mikroarchitekturebene*.

Zusammengesetzte Operationen wie die gerade genannten können auf der nächsten Ebene M2 ausgeführt werden, die die *Befehlssatzarchitekturebene* oder *ISA Ebene* genannt wird („ISA“ ist die Abkürzung für „Instruction Set Architecture“).

Wer den Rechner auf dieser dritten Ebene betrachtet, sieht die oben genannten elementaren Bauteile kombiniert zu einem Gesamtrechenwerk, die *CPU* („central processing unit“), das kompliziertere Befehle und für Programmierer und Anwender sinnvolle Operationen ausführen kann, wie das Laden von Daten aus dem Hauptspeicher in Register, das Abspeichern von Registerinhalten im Hauptspeicher, und komplizierte und bequeme Rechenoperationen mit dem Inhalt der Register. Der Hersteller der CPU implementiert einen bestimmten, meist größeren Befehlssatz für seine CPU mit dem Hintergedanken, in Abwägung aller Faktoren wie Wirtschaftlichkeit und Rechengeschwindigkeit das Erstellen von Programmen für den Rechner möglichst einfach und bequem zu machen.

Die einzelnen Befehle im Befehlssatz werden, wie alles im Rechner, durch gewisse Bitfolgen kodiert, die im Speicher innerhalb von Wortgrenzen abgelegt sind und die die CPU der Reihe nach aus dem Speicher lesen und zur Ausführung der gewünschten Operation entziffern kann.

Neben Rechenbefehlen und Datenbewegungsbefehlen werden auch Steuerbefehle implementiert, die den normalerweise sequentiellen Programmfluss verändern können, und die gewisse Statusmerkmale von Rechenergebnissen (war das Ergebnis positiv? war es 0? hat es einen Übertrag gegeben?) abfragen können und in Abhängigkeit der Antwort den Programmfluss (Reihenfolge der Befehle) umdirigieren können. So können unter Programmkontrolle Entscheidungen getroffen werden und ein Programm kann flexibel auf Besonderheiten im Rechenablauf reagieren. Unter anderem können Programmabschnitte wiederholt werden, so dass oft benötigte komplizierte Operationen nur einmal programmiert werden müssen, und Operationen, die auf kleine Datenmengen agieren, können durch Wiederholung auf große Datenmengen angewendet werden, indem das Programm die Datenmenge in kleinen Stücken abarbeitet und selber entscheiden kann, wann alle Daten behandelt worden sind.

Die einzelnen Befehle, die die CPU ausführen kann, und die direkt durch ein Befehlscode angesprochen werden können, können auf dieser Stufe schon sehr kompliziert ausfallen — einzelne Befehle können große Textabschnitte im Speicher bewegen oder Zeichenfolgen in großen Textabschnitten suchen,

oder sie können umfangreiche Rechenoperationen wie Multiplikationen und Divisionen ausführen. Sogar aufwändige ***Gleitkommaoperationen***, die auf Zahlen in der „wissenschaftlichen Schreibweise“

$$\text{Mantisse} \times 2^{\text{Exponent}}$$

wirken und einem dadurch die Bürde abnehmen, bei Berechnungen auf die Lage des Kommas in einer Bruchzahl zu achten, sind im Befehlssatz mancher CPUs vertreten.

Wie wird die ISA Ebene realisiert? Es gibt zwei Möglichkeiten dazu. Schon 1951 wurde von Maurice Wilkes der Vorschlag gemacht, Computer mit drei Ebenen zu bauen, wobei die eigentliche Maschine, die mittlere Ebene, nur sehr einfache Befehle haben sollte, die leicht in Elektronik zu realisieren wären, und ein fest eingebautes und in der Maschinensprache dieser Ebene geschriebenes ***Mikroprogramm*** die komplizierteren Befehle der obersten Ebene interpretieren sollte, so dass der menschliche Programmierer eine virtuelle Maschine mit leistungsfähigeren Befehlen vorgesetzt bekäme.

Diese Idee wurde auf immer mehr Rechnern implementiert, bis sie schließlich um 1970 zum Standard für alle Rechner wurde. Die Entwickler konnten mit Hilfe der Mikroprogrammierung immer neue und immer bequemere und stärkere Befehle in den Befehlssatz einbauen, ohne dass die Elektronik zu kompliziert und teuer wurde, aber der Preis, der zu zahlen war, war dass diese Befehle immer langsamer wurden.

Die Rechner, die große, bequeme, aber dafür auch aufgeblasene Befehlssätze hatten, wurden ***CISC Rechner*** genannt („Complex Instruction Set Computer“), und ab 1980 gab es eine Gegenbewegung, die den Einsatz von ***RISC*** Technologie, d. h., „Reduced Instruction Set Computers“, propagierte. Diese Rechner hatten einen eingeschränkten und einfachen Befehlssatz, der direkt in Hardware realisiert werden konnte (oberhalb der Mikroarchitekturebene), und die Befehle waren deshalb sehr schnell und effizient, verglichen mit CISC Befehlen.

Die Kontroverse um diese Strategien ist inzwischen etwas abgeflaut, weil durch die rapide wachsende Transistordichte in integrierten Schaltkreisen die Hardwareimplementation immer kompliziertere Befehle möglich geworden ist. Heute ist es üblich, die wichtigsten und am häufigsten benutzten Befehle, die auch relativ einfach sind, als RISC Befehle, d.h. in Hardware implementiert, zu realisieren, aber auch kompliziertere Befehle in den Befehlssatz einzubauen, die dann teilweise durch ein Mikroprogramm interpretiert werden müssen.

Auch wenn die ISA Ebene leistungsfähige Befehle bietet, ist sie noch nicht angenehm zu benutzen, denn die Befehle eines Programms müssen dem

Rechner ja als Bitfolgen in seinem Speicher präsentiert werden, genau wie die Daten, die bearbeitet werden. Wer also in den Speicher hineinschaut, sieht nichts als Nullen und Einsen, und wer ein Programm erstellen will, muss die richtigen Folgen von Nullen und Einsen in den Speicher schreiben.

An dieser physikalischen Realisierung des Programms und seiner Daten deutet nichts auf die tatsächliche Bedeutung oder Wirkung einer bestimmten Bitfolge hin — alles ist willkürlich und alles sieht gleich aus.

Gewiss gibt es Handbücher, die beschreiben, welche Bits in einem Programmwort welche Bedeutung haben und welche Bitfolgen im Befehlscode-Teil des Wortes welchem Befehl entsprechen. Aber ein Programmierer müsste ständig diese Tabellen konsultieren oder Bitfolgentabellen auswendig lernen, um sie dann zu einem Programm zusammenzusetzen. Jede nachträgliche Änderung könnte weitreichende Anpassungen an anderer Stelle im Programm mit sich ziehen, weil auch die Adressen der Daten und der Befehle sich ändern könnten. Und Programme in dieser Form wären völlig unlesbar und unverständlich.

Es wäre kaum möglich, auf diese Weise sicher und fehlerfrei zu programmieren. Deshalb müssen der ISA Ebene noch weitere Ebenen aufgesetzt werden.

Bevor man Abhilfe für das eben genannte Problem schaffen kann, braucht man eine bessere Möglichkeit, die Gesamtbenutzung des Rechners zu steuern, um zu gewährleisten, dass mehrere Benutzer einen Rechner teilen können, und das Hilfsprogramme, wie sie auf den höheren Ebenen benötigt werden, für alle bereitgestellt werden können und nicht von jedem Benutzer selber beigesteuert werden müssen.

Zu diesem Zweck muss zunächst eine Art Koordinationsstelle eingerichtet werden. Diese Funktion hat die nächste Ebene M3, die **Betriebssystemebene**.

Die Betriebssystemebene stellt keine radikale Veränderung der ISA Ebene dar, sondern ihre Sprache beinhaltet die ISA Sprache unverändert und ergänzt sie nur durch ein paar zusätzliche Befehle, die in der ISA Sprache interpretiert werden. Diese zusätzlichen Befehle können aber Programme im Speicher durch andere Programme ersetzen, Programmausführungen unterbrechen, um andere Programme laufen zu lassen, und dann die ursprünglichen Programme wieder weiterlaufen lassen, sie können von einem Benutzer benötigte Hilfsprogramme aus einem zentralen Lager holen und in den Speicher laden, sie können Buch führen über benutzte Rechenzeit und über Betriebskosten, und sie können vieles andere mehr, das eine geordnete Nutzung des Rechners durch mehrere Personen ermöglicht und jedem Nutzer den Zugriff auf umfangreiche zentral bereitgestellte Ressourcen ermöglicht.

Diese speziell zum Betriebssystem gehörenden Befehle erleichtern die Ar-

beit des Benutzers erheblich. Er muss sein Programm nur in den Speicher bringen oder auf einem externen Speichermedium dem Computer zugänglich machen, aber er muss nicht selber direkt dafür sorgen, dass der Rechner andere Aufgaben fallen lässt und mit der Abarbeitung seines Programms beginnt. Er muss auch nicht selber mit anderen Benutzern konkurrieren, die auch Rechenzeit haben wollen (oder mit ständig laufenden Standardprogrammen wie etwa ein Virusprüfer, die wichtige Funktionen innehaben und deshalb gleichzeitig mit dem Benutzerprogramm weiter ausgeführt werden sollen).

Ein Computerprogramm muss nebenbei viele Aufgaben ausführen, die sehr kompliziert zu programmieren sind, wie etwa das Einlesen von Tastenbetätigungen auf der Tastatur, die Wahrnehmung von Mausclicks, die Ausgabe von Daten auf einem Drucker, die Reservierung von Speicherplatz für Zwischenberechnungen, und andere Aufgaben, über deren Notwendigkeit der Benutzer vielleicht noch nicht einmal Bescheid weiss.

All diese Dinge sind Standardaufgaben, für die es Standardprogramme gibt, entwickelt von den Herstellern der angeschlossenen Peripheriegeräte oder von den Entwicklern der Benutzeroberfläche, und das Betriebssystem hat die Aufgabe, dafür zu sorgen, dass diese Standardprogramme benutzt werden können, wann immer der Benutzer sie benötigt, und ohne dass er über die genaue Funktionsweise der benötigten Mittel, sagen wir über den inneren Befehlssatz des angeschlossenen Druckers, Bescheid wissen muss. Der Benutzer kann, um bei diesem Beispiel zu bleiben, einfach einen Ausdruck in einer bestimmten Qualität verlangen, und das Betriebssystem weiss, wie man den angeschlossenen Drucker dazu bringt, diesen Auftrag auszuführen, und es weiss es immer noch, wenn man den Drucker durch einen ganz anderen ersetzt.

Der Benutzer muss die erwähnten Hilfsprogramme nicht selber einbinden und er muss sie vor allem nicht selber entwickeln — dazu wäre er in der Regel gar nicht in der Lage. Diese Hilfsdienste werden alle vom Betriebssystem übernommen.

Das oben erwähnte Problem der kryptischen und für Menschen nicht direkt verständlichen Codes in Maschinenprogrammen wird auf der Betriebssystemebene noch nicht direkt behoben, aber die Betriebssystemebene erleichtert die Implementierung der Lösung, weil die Lösung durch ein Hilfsprogramm geboten wird, das auch sinnvollerweise zentral bereit gehalten werden sollte, weil es von vielen Benutzern gebraucht wird.

Dieses Hilfsprogramm nennt sich ein **Assembler** oder zu deutsch *Assemblierer* und es macht es möglich, Maschinenspracheprogramme zu schreiben, ohne die genaue Kodierung der Befehle zu kennen. In der Assemblersprache gibt es die gleichen Befehle, wie in der ISA Sprache, aber sie haben suggestive und leicht zu merkende Namen anstelle der Zahlencodes.

Zum Beispiel hat fast jeder Rechner einen speziellen Register, genannt der **Akkumulator**, in dem direkt gerechnet werden kann, und um Daten für eine Berechnung zugänglich zu machen muss man sie in den Akkumulator laden. Dafür gibt es immer einen Maschinenbefehl. Auf der IBM 7094 der 60er Jahre hieß dieser Befehl CLA (CLear and Add), auf der Intel 8080 hieß er LDA (LoaD A), auf der 8088 und auf dem Pentium heißt er MOV A,· (MOVe), aber alle Abkürzungen sind schnell erlernbar und leicht zu merken. Der Assembler liest die symbolischen Namen und ersetzt sie durch die richtigen Bitfolgen der Befehlscodes.

Auch mit der genauen Bitfolgenstruktur einzelner Befehlswörter muss der Programmierer sich nicht herumschlagen. Er muss zwar wissen, wieviele Operanden ein Befehl hat, aber in seinem Programm kann er sie in einer Standardschreibweise angeben (meistens nach dem Befehlsnamen und von diesem abgesetzt), ohne zu wissen, an welchen Stellen im Befehlswort der Operationscode steht und an welchen Stellen die Operanden stehen.

Ein weiterer und nicht zu unterschätzender Vorteil der Assemblersprache ist, dass sie **symbolische Adressierung** erlaubt. Für jede Speicherstelle, die im Programm angesprochen werden muss (das kann auch der Ort eines Programmbefehls sein!), darf der Programmierer einen Namen erfinden (insbesondere kann er den Namen so wählen, dass er an den Zweck oder die Bedeutung dieser Speicherstelle für sein Programm erinnert), und an anderer Stelle im Programm kann er diese Speicherstelle über ihren Namen ansprechen. Der Assembler führt Buch über die tatsächliche Lage der Speicherstellen und ersetzt den Namen später durch seinen Zahlenwert (Adresswert). Wenn der Programmierer sein Programm abändert, so dass Speicherzellen sich verlagern, muss er die Referenzen auf die Speicherzellen trotzdem nicht anpassen — das macht der Assembler automatisch.

Die symbolische Adressierung ermöglicht auch das Schreiben von Programmen, deren Lage im Speicher vorher nicht bekannt ist, so dass der Programmierer zahlenmäßig festgelegte Adressen gar nicht verwenden könnte.

(Eine Erweiterung dieser Fähigkeit, die von Betriebssystemen gewährleistet wird, ist die **Relozierbarkeit** von Programmen: die Assembler produzieren nicht die endgültige Version eines Maschinenprogramms, sondern eine „Musterversion“, in der die enthaltenen Speicheradressen sich auf den Programmbeginn beziehen und gekennzeichnet sind. Wenn das Programm dann vom Betriebssystem in den Speicher geladen wird, addiert es die Adresse des Programmbeginns zu diesen gekennzeichneten Stellen im Programm, so dass sie dann auf die richtige tatsächliche Lage zeigen. Das klappt dann auch genau so gut, wenn bei einem anderen Programmlauf das Programm an eine ganz andere Stelle im Speicher geladen wird.)

Die meisten Assembler bieten noch zusätzliche Annehmlichkeiten. Da-

ten, die schließlich binär kodiert im Speicher landen müssen, können in der Assemblersprache in ihrer für Menschen gewohnten Form als Text oder als Dezimalzahl eingegeben werden. Und Assembler bieten auch die Möglichkeit, so genannte **Makrobefehle** zu benutzen: wenn ein Programmierer feststellt, dass er für eine bestimmte kleine Berechnung immer wieder die gleiche Befehlsfolge in seinem Programm benutzt, kann er dieser Befehlsfolge einen Namen geben und die ganze Befehlsfolge über diesen Namen an jeder Stelle in seinem Programm einfügen.

Der Assembler ist ein **Übersetzer**, der die soeben beschriebene symbolische und für Menschen leicht verständliche Sprache lesen und entziffern kann, und in die richtigen Bitfolgen für die gewünschten Maschinenbefehle übersetzen kann.

Der Übergang zur Assemblerebene stellt eine wesentliche Zäsur in der Ebenenhierarchie dar.

Die Assemblerebene ist die erste, die *Computeranwendern* einen bequemen Zugang zu den Rechenfähigkeiten des Computers bietet. Programme auf den unteren Ebenen sind nichts anderes als Bitfolgen und somit sehr unhandlich für Menschen — die Programmierer, die auf diesen Ebenen Programme schreiben, sind nicht Endnutzer sondern **Systemprogrammierer**, deren Aufgabe es ist, die Maschinensprachenebene zu den höheren Ebenen auszuweiten, das Betriebssystem zu erstellen, und schließlich angenehmere Computersprachen wie zum Beispiel die Assembler zu schreiben. Und auch die Systemprogrammierer werden für die Weiterentwicklung bestehender Systeme von den symbolischen Möglichkeiten ab dem Assemblerniveau Gebrauch machen.

Ein anderer Unterschied ist, dass die Sprachen der unteren Ebenen fast immer interpretiert werden (in der Sprache einer tiefer liegenden Ebene), während die Sprachen der Ebenen oberhalb der Betriebssystemebene meistens übersetzt werden in eine tiefere Sprache. Aber zu dieser Regel gibt es auch viele Ausnahmen.

Die Assemblerebene ist noch immer nicht die letzte. Sie hat den Nachteil, so bequem sie ist, dass sie immer noch auf den Befehlssatz der Maschine ausgerichtet ist und die Maschinensprache nur in etwas abgemilderter und erträglicherer Form dem Programmierer zugänglich macht. Immer noch muss er sich aber nach den Fähigkeiten der Maschine richten und die abstrakte Aufgabe, die er eigentlich programmieren will, so in detailliert festgelegte Einzelschritte zerlegen, dass der Befehlssatz des Rechners diese Schritte ausführen kann.

Will man ein Programm auf einen neueren Rechner oder auf einen Rechner eines ganz anderen Typs übertragen, so muss man unter Umständen das ganze Programm vom Grunde auf neu schreiben.

Die meisten Anwender wollen gar nicht wissen, wie der Rechner, den sie benutzen, im Einzelnen funktioniert — sie haben wissenschaftliche Berechnungen durchzuführen, müssen für eine Firma Lagerbestände verwalten und mit den daraus resultierenden Zahlen spezielle Berechnungen durchführen, oder sie wollen vielleicht ein Nutzprogramm, wie zum Beispiel ein Textverarbeitungsprogramm schreiben.

Um diese Bedürfnisse zu befriedigen gibt es fast seit dem Beginn der Computernutzung Programmiersprachen, die direkt auf die angepeilte Anwendung zugeschnitten sind und sich überhaupt nicht (oder nur indirekt) auf die Struktur des Rechners beziehen, auf dem sie ausgeführt werden. Diese problemorientierten Sprachen heißen *höhere Programmiersprachen* oder *Hochsprachen*. In ihnen kann man algebraische Berechnungen in der gewohnten algebraischen Notation angeben, man kann Programmschleifen mit Abbruchbedingungen auf sehr natürliche Weise formulieren, und man kann auch mit komplizierten Daten wie Tabellen oder mathematischen Matrizen oder komplexen Zahlen direkt umgehen.

Andere Hochsprachen sind auf Listenverarbeitung ausgerichtet oder auf sprachliche Aufgaben (wie die Interpretation von Aussagen in einer formalen Sprache und dergleichen). Andere Sprachen sind für Anwendungen im Betriebswesen konzipiert, andere für Präsentationen im Internet, wieder andere für Steuerungsaufgaben.

Wichtige (und historisch wichtige) Beispiele von Hochsprachen sind FORTRAN, LISP (aus den frühen 60er Jahren aber immer noch aktuell), SNOBOL (Textmanipulation), ALGOL, BASIC (eine interpretierte Hochsprache), PASCAL, C (aus den frühen 70er Jahren aber immer noch aktuell), C++, und Java. Im Laufe der Zeit haben sich auch Programmierstile weiterentwickelt, grob gesagt dahingehend, dass in einem Programm nicht die Rechenmethode sofort ins Auge springen soll, sondern eher der Rechenzweck.

So enthielten frühe Programme viele Sprünge auf eine bestimmte Stelle im Programm, um Schleifen zu durchlaufen, und später wurden solche Techniken, deren Zweck nicht sofort ersichtlich ist, durch bedingte Schleifen (**while** Bedingung **do**; Ausführungsteil; **end do**;) ersetzt, bei denen die Intention klarer ist.

Die letzte wichtige Entwicklung in diese Richtung sind Objekt-orientierte Sprachen, bei denen Daten nicht als Eingaben zu einem Rechenalgorithmus aufgefasst werden, und dann vom richtigen Datentyp für den Algorithmus sein müssen, sondern als Objekte aufgefasst werden, die von sich aus gewisse Operationen („Methoden“) verstehen und selber wissen, wie mit ihnen gerechnet werden muss, wenn eine Methode auf sie angewandt wird.

Zum Beispiel kann man Zahlen addieren, aber es gibt auch eine „Addition“ + für Texte, die sie einfach aneinanderreihet. In einer nicht objektorientierten

Sprache muss man zwischen diesen Operationen streng unterscheiden und die Typen der Operanden kontrollieren, bevor man die $+$ Operation ausführen kann. In einer objektorientierten Sprache wissen Daten selber, welche Variante der $+$ -Methode für sie richtig ist, und der Programmierer muss sich nicht darum kümmern.

Nach diesem Überblick über die Ebenenhierarchie und die allgemeine Rechnerarchitektur sollte noch ein kurzer Abriss über die Computergeschichte folgen, aber aus Zeitgründen haben wir dieses Thema in die Übungsstunden verbannt und wir verzichten deshalb auch darauf, dieses Thema im Skriptum zu behandeln.

Kapitel 2

Digitale Logik

Wir haben in der Einleitung gesehen, dass die Architektur eines Computers aus einer Hierarchie von Ebenen besteht, die in „kleinen“ Schritten die einfachen Fähigkeiten einer aus elektronischen Bausteinen gebauten Maschine zu einem bequemen und leistungsfähigen modernen Anwendungssystem erweitert.

In diesem Kapitel wollen wir nun die unterste, einfachste Ebene detailliert kennenlernen. Sie wird gebildet von der eigentlichen Elektronik, die sich in den integrierten Schaltkreisen des Rechners befindet, und diese ist aus relativ einfachen Bausteinen zusammengesetzt, die aber in großer Zahl vorhanden sind, um die Grundfunktionen des Rechners auszuführen.

Diese elementaren Bausteine, *digitale Logikgatter* oder schlicht *Gatter* genannt, werden aus wenigen miteinander verbundenen *Transistoren* gebaut, und wir wollen deshalb mit einer kurzen Beschreibung der Funktionsweise eines Transistors beginnen.

Es gibt mehrere Arten von Transistoren, die verschiedene Vor- und Nachteile haben, wobei man für den Rechnerbau solche Varianten bevorzugt, die weniger Energie verbrauchen und die sich in einem integrierten Schaltkreis dichter verpacken lassen. Ihre prinzipielle Eignung als Grundsteine für den Rechnerbau hängt aber von solchen Besonderheiten nicht ab, sondern von der allgemeinen Fähigkeit von Transistoren (oder in der Frühzeit der Rechnerentwicklung von Vakuumröhren), als elektrisch gesteuerte elektrische *Schalter* zu fungieren. Transistoren und Vakuumröhren können auch zur Signalverstärkung eingesetzt werden, aber diese Fähigkeit spielt für den Rechnerbau keine Rolle.

Weil es zum Verständnis des Prinzips des Rechnerbaus aus Transistoren nicht auf die eingesetzte Transistorart ankommt, beschreiben wir stellvertretend für alle Typen nur ein typisches Beispiel eines Transistors, die *Feldeffekttransistoren*, und nur ihre Funktion als *Schalter*.

In einem elektronischen Rechner werden Bits durch unterscheidbare elektrische Zustände dargestellt, und in modernen Rechnern sind diese Zustände elektrische *Spannungen*. Teile der Rechner sind geerdet, d. h., sie liegen auf Masse und ihre Spannung bildet das Grundniveau 0 V für die Spannungsskala. Der Rechner wird mit einer Betriebsspannung U_H von etwa +5 V versorgt (etwas genauer: die meisten modernen CPUs und Speicher arbeiten mit einer Betriebsspannung von +3,3 V). Diverse Stellen in den Rechnerschaltkreisen können mit der Betriebsspannung oder mit Masse elektrisch verbunden werden, so dass zwei Spannungszustände möglich sind.

Für die Darstellung von Bits werden zwei Spannungsbereiche unterschieden, wobei der Bereich 0–1 V als niedriges Signal zum Beispiel einen 0 Bit darstellen kann, und der Bereich 2–5 V dann als hohes Signal gedeutet wird und einen 1 Bit darstellt.

Diese Signale können durch Transistoren verändert werden. Abbildung 2.1 ist die Skizze eines **Feldeffekttransistors**, der aus einer Siliziumschicht besteht, die mit fremden Atomen „verunreinigt“ wurde, so dass ein Teil (die p-Schicht) ein Mangel an Elektronen hat, und zwei in die p-Schicht eingelassene Bereiche (die n-Schicht) ein Überfluss an Elektronen haben. Der Transistor

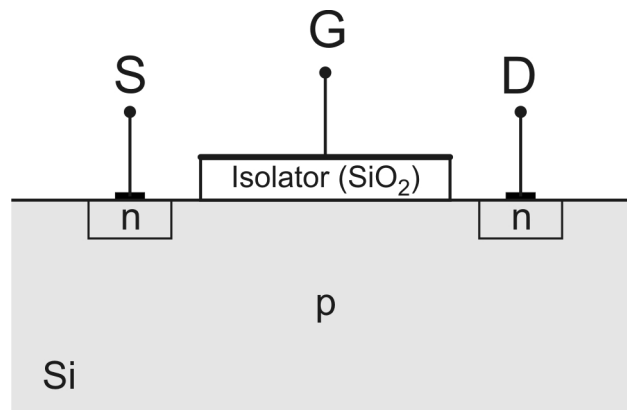


Abbildung 2.1: Ein Feldeffekttransistor

hat drei Verbindungen zur Außenwelt, also drei Kontakte, genannt **Source** (S), **Gate** (G) und **Drain** (D). Die Verbindungen S und D werden mit den beiden n-Gebieten verbunden, und G wird mit einer leitenden Schicht verbunden, die auf einem Isolator aufliegt (meistens aus Silizium Dioxid), der zwischen den n-Regionen auf der p-Schicht liegt.

Solange an G keine Spannung anliegt, ist die p-Schicht isolierend und zwischen S und D kann kein Strom fließen. Wenn aber an G eine genügend hohe Spannung (etwa 5 V) angelegt wird, werden durch das elektrische Feld Elektronen in die Grenzschicht vom p-Gebiet zum Isolator hineingezogen und sie

wird leitend, d. h., Strom kann jetzt zwischen S und D fließen. (Wohlbemerkt: zwischen G und S oder D fließt *kein* Strom; nur der Feldeffekt wirkt.)

Die Wirkung des Transistors ist also die eines Schalters, der durch Anlegen einer Spannung an G geschlossen werden kann.

Heutzutage können sehr winzige Feldeffekttransistoren auf einem Siliziumchip integriert werden. Typisch sind Gate-Breiten von etwa 100 nm und im Labor sind Breiten von 50 nm und Isolatordicken am Gate von 1,2 nm (etwa 5 Atomdurchmesser) erzielt worden.

Neben den Feldeffekttransistoren gibt es auch **bipolare Transistoren**, die die gleiche Wirkung aufweisen, obwohl sie anders aufgebaut sind. Auch sie haben drei Kontakte, diesmal genannt **Base**, **Emitter** und **Collector**, und die Verbindung zwischen Emitter und Collector ist isolierend (wie ein unendlich hoher Widerstand), wenn die Spannung an der Basis niedrig ist, und ist leitend, wenn die Spannung an der Basis hoch ist. Bei diesen Transistoren fließt aber tatsächlich Strom durch die Basis.

In der Wirkung als Schalter sind beide Arten ähnlich, und darauf kommt es bei der Konstruktion von digitalen Logikelementen an.

Hier einige gebräuchliche Symbole für Transistoren, die in Schaltkreisdigrammen verwendet werden; die Symbole geben die interne Struktur in etwa wieder, so dass es noch andere Symbole für andere Typen gibt (und außerdem gibt es europäische und amerikanische Varianten), aber die hier präsentierten sollen ja nur als Beispiel dienen.

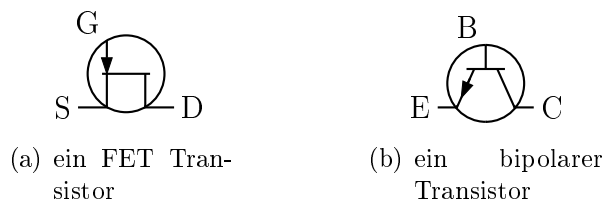


Abbildung 2.2: Schaltkreissymbole für Transistoren

Wir wollen jetzt sehen, wie man mit Transistoren digitale Daten elektronisch verarbeiten kann. Wir beginnen mit einem ganz einfachen Schaltkreis (Abbildung 2.3 auf der nächsten Seite) mit einem FET Transistor, dessen Source mit Masse verbunden, also geerdet ist, und dessen Drain über ein Widerstand (zur Vermeidung eines Kurzschlusses) mit der Betriebsspannung $U_H \approx 5\text{ V}$ verbunden ist. Zwischen dem Drain und dem Widerstand greifen wir die dort gegebene Spannung als Ausgangssignal U_a ab.

Wenn die Eingangsspannung U_e am Gate niedrig ist, also eine 0 darstellt, dann ist der Transistor gesperrt und deshalb isolierend, und am Ausgang liegt die Betriebsspannung $U_a = U_H$ an, die eine 1 darstellt.

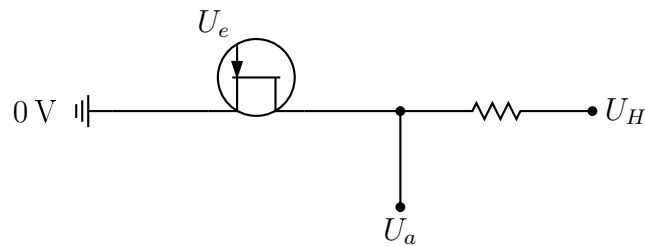


Abbildung 2.3: Ein Transistor als Inverter

Hingegen, wenn die Eingangsspannung U_e am Gate hoch ist, also eine 1 darstellt, dann ist der Transistor als Schalter geschlossen, also leitend, und der Ausgang ist mit Masse verbunden und es liegt dort die Spannung $U_a = 0\text{ V}$ an, die eine 0 darstellt.

In anderen Worten, diese einfache Schaltung ist ein **Inverter**; sie *invertiert* den Eingang und liefert am Ausgang genau den *anderen* der beiden Zustände 0 und 1, der am Eingang anliegt.

Mit zwei Transistoren können wir nach dem gleichen Schema eine Schaltung bauen, die zwei Eingänge an den Gates der Transistoren hat und an der gleichen Stelle wie in Abbildung 2.3 ein Ausgangssignal liefert, der eine logische Funktion (**boolesche Funktion**) der beiden Eingänge ist. Eine solche Schaltung nennen wir ein **Gatter**.

Wir können die beiden Transistoren in Serie verbinden zwischen Masse und der Betriebsspannung, wie in Abbildung 2.4 (wieder verhindert der Widerstand einen Kurzschluss).

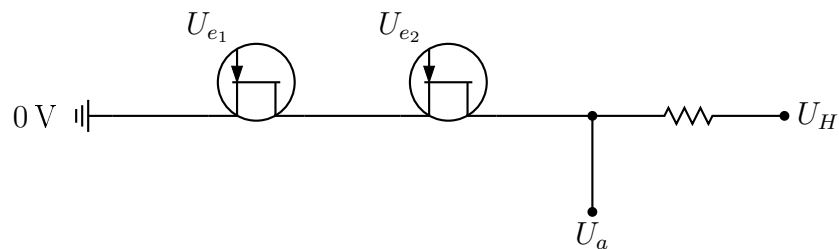


Abbildung 2.4: Ein NAND Gatter aus zwei Transistoren

Am Ausgang liegt die Betriebsspannung an, es sei denn, *beide* Transistoren sind geschlossen, d. h., an beiden Eingängen liegt ein hohes Signal an. Nur in diesem Fall ist der Ausgang mit Masse verbunden und wird auf 0 V heruntergezogen.

In der Sprache der Bits: der Ausgang ist 1, außer wenn $e_1 = e_2 = 1$; dann (und nur dann) ist der Ausgang 0.

Die von diesem Gatter „berechnete“ boolesche Funktion können wir auch durch eine Wertetabelle angeben. Sie sieht so aus:

e_1	e_2	a
0	0	1
0	1	1
1	0	1
1	1	0

Tabelle 2.1: Wertetabelle der NAND Funktion

Wenn 1 (wie nach Konvention üblich) ein logisches WAHR und 0 ein logisches FALSCH darstellt, dann ist der Ausgang dieses Gatters genau dann FALSCH, wenn beide Eingänge, also Eingang 1 *und* Eingang 2, WAHR sind. Das ist gerade die Umkehrung oder Invertierung der UND Funktion (englisch AND), und deshalb heißt diese boolesche Funktion die „NOT AND“ oder kurz und bündig die **NAND Funktion**.

Das Gatter, das wir gebaut haben und das diese Funktion berechnet, heißt deshalb ein **NAND Gatter**.

Wir sollten erwähnen, dass moderne NAND (und andere von uns hier vorgestellte) Gatter einen etwas komplizierteren Aufbau als den hier beschriebenen haben (mit der doppelten Anzahl von Transistoren, aber den Vorteil, dass sie schneller schalten und weniger Leistung verbrauchen, als der von uns vorgeschlagene Schaltkreis¹).

Statt der Schaltung in Abbildung 2.4 können wir die beiden Transistoren auch *parallel* zwischen Masse und der durch einen Widerstand abgesicherten Betriebsspannung legen, wie in Abbildung 2.5 auf der nächsten Seite.

Wenn mindestens *einer* der beiden Transistoren geschlossen ist, dann ist der Ausgang mit Masse verbunden und wird auf 0 V heruntergezogen. Nur wenn beide Transistoren gesperrt sind (weil an ihren Eingängen ein niedriges Signal anliegt), hat der Ausgang die hohe Spannung.

In der Sprache der Bits: der Ausgang ist 0, außer wenn $e_1 = e_2 = 0$; dann (und nur dann) ist der Ausgang 1.

¹Unsere Schaltkreise beschreiben nMOS Gatter, wo MOS die Abkürzung von „metal oxide semiconductor“ ist und das „n“ so genannte n-Kanal Transistoren kennzeichnet, mit einem Aufbau wie in Abbildung 2.1. Es gibt auch Transistoren, in denen die p und n Bereiche vertauscht sind, und die deshalb genau umgekehrt auf eine am Gate angelegte Spannung reagieren. Die heute eingesetzten CMOS=„complementary metal oxide semiconductor“ Bausteine benutzen immer Transistorpaare von den beiden (komplementären) Typen mit einer symmetrischen Anordnung, die den Ausgang entweder mit der Betriebsspannung oder mit Masse verbindet. Dieser symmetrische Aufbau, obwohl komplizierter, bewirkt die genannten Vorteile.

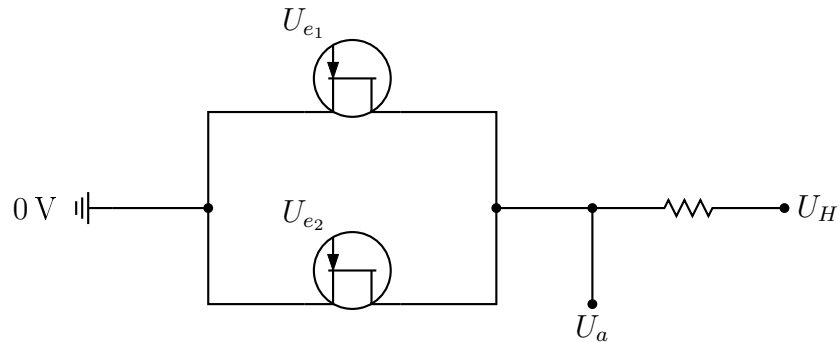


Abbildung 2.5: Ein NOR Gatter aus zwei Transistoren

Die von diesem Gatter berechnete boolesche Funktion können wir wieder durch eine Wertetabelle angeben. Sie sieht so aus:

e_1	e_2	a
0	0	1
0	1	0
1	0	0
1	1	0

Tabelle 2.2: Wertetabelle der NOR Funktion

Der Ausgang dieses Gatters ist also genau dann FALSCH, wenn mindestens einer der beiden Eingänge, also Eingang 1 *oder* Eingang 2, WAHR ist. Das ist gerade die Umkehrung oder Invertierung der ODER Funktion (englisch OR), und deshalb heißt diese boolesche Funktion die „NOT OR“ oder abgekürzt die **NOR Funktion**.

Das Gatter, das wir mit einem parallel geschalteten Transistorenpaar gebaut haben und das diese Funktion berechnet, heißt deshalb ein **NOR Gatter**.

Übrigens, für den aus einem Transistor aufgebauten Invertierer (Abbildung 2.3), der unser erster Versuch bei der Konstruktion von Gattern war, können wir auch eine Wertetabelle aufstellen: Wir sehen, der Ausgang hat

e	a
0	1
1	0

Tabelle 2.3: Wertetabelle der NOT Funktion

den Wert 1 = WAHR genau dann, wenn der Eingang 0 = FALSCH ist, al-

so NICHT WAHR ist, und deshalb wird diese Funktion die NOT Funktion genannt und jedes Gatter, das sie berechnet, ein NOT Gatter.

Wir haben bisher immer so getan, als würden Transistoren und die aus ihnen gebauten Gatter *sofort* schalten. Natürlich stimmt das nicht — es dauert immer eine gewisse, allerdings sehr kurze Zeit, bis elektrische Signale von einem Ort zum anderen fließen, und es dauert etwas Zeit, bis Spannungen sich verändern und bei ihrem neuen Wert stabil sind.

Aber die schnellsten NAND Bausteine (aus bipolaren Transistoren) können in etwa $10\text{psec} = 10^{-12}\text{sec}$ umschalten. Unter der Annahme, dass elektrische Signale sich auf einem Chip mit Lichtgeschwindigkeit $= 0,3\text{mm/psec}$ ausbreiten, braucht ein Signal 100 psec oder 10 NAND-Zeiten, um eine typische Strecke von 3 cm auf einer Platine zurückzulegen.

Deshalb ist die Anordnung der Bausteine auf einer Platine von Bedeutung, obwohl diese Anordnung inzwischen wegen der Fertigungsverfahren weitgehend genormt ist.

Wir wollen jetzt weitere Gatter und Logikbausteine untersuchen, aber nicht mehr versuchen, sie aus einzelnen Transistoren aufzubauen. Denn wir haben schon einen nützlichen und billigen (da aus wenigen Transistoren bestehenden) Grundbaustein im NAND Gatter gefunden (oder im NOR Gatter). Durch Verbindung der Ausgänge solcher Gatter mit den Eingängen anderer Exemplare kann man weitere Gatter leicht realisieren.

Das NAND Gatter und das NOR Gatter sind wegen ihrem einfachen Aufbau die Grundelemente im Rechnerbau, und es gibt inzwischen eine etablierte, gut beherrschte und wirtschaftliche Technologie für die Fertigung integrierter Schaltungen mit sehr vielen solchen Gattern, sinnvoll miteinander verbunden, auf einem einzigen Siliziumchip.

NAND Gatter auf Chips haben heute eine Größe von unter 10^{-6}m , und sind auf vielfältiger Weise realisierbar.

Wir werden deshalb als nächstes schauen, welche zusätzlichen einfachen Gatter wir aus Kombinationen von NAND Gattern erhalten können. Da man oft auch boolesche Funktionen von mehr als zwei Variablen realisieren muss, werden wir auch theoretische Überlegungen über die Darstellbarkeit von booleschen Funktionen mit Gattern anstellen, wozu die Entwicklung einer bequemen Notation und bequeme Rechenregeln für boolesche Funktionen gehört.

Ab jetzt ist die innere Struktur eines Gatters für uns nicht mehr wichtig, und wir werden jetzt Schaltkreise zeichnen, in denen keine Transistoren mehr erscheinen, sondern gleich ganze Gatter als Grundelement.

Es gibt für Gatter mehrere Notationen, aber die amerikanischen Symbole sind sehr einprägsam und leicht zu lesen, und wir werden deshalb diese Notation und nicht die DIN Symbole benutzen.

Hier sind die Symbole für die bisher konstruierten Gatter:

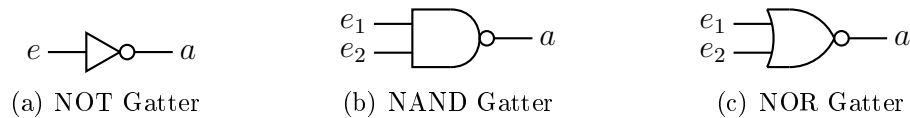


Abbildung 2.6: Schaltkreissymbole für Gatter

Obwohl wir ein NOT Gatter schon aus Transistoren konstruiert haben, kann man es auch leicht aus einem einzigen NAND Gatter konstruieren, indem man das (einzige) Eingangssignal auf beide Eingänge des NAND Gatters legt:

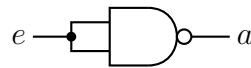


Abbildung 2.7: Ein NOT Gatter durch Zusammenlegung der Eingänge eines NAND Gatters

Wir können leicht nachprüfen, dass diese Konstruktion tatsächlich ein NOT Gatter liefert, denn weil das gleiche Signal e an beiden Eingängen des NAND Gatters anliegt, erhalten wir die Wertetabelle für die von diesem Schaltkreis berechnete boolesche Funktion aus den beiden Zeilen der Wertetabelle 2.1 für die NAND Funktion, in denen $e_1 = e_2 =: e$, und offensichtlich ergeben diese Werte die Werte der NOT Funktion (Umkehrung von e).

Als Symbol für das NOT Gatter benutzt man natürlich nicht den abgebildeten Schaltkreis, sondern ein NAND Symbol mit nur einem Eingang

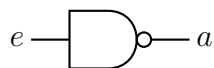


Abbildung 2.8: Ein NOT Gatter als NAND mit einem Eingang

oder das Symbol aus Abbildung 2.6(a).

Weil aber Signalinvertierer eine sehr nützliche und häufig verlangte Funktion ausüben, gibt es eine noch bessere und knappere Notation dafür. Haben Sie sich schon gewundert über die kleinen Kreise am Ausgang der bisher vorgestellten Gatter? Sie sind das Zeichen für eine Signalinversion, und wenn man sie weglässt, dann erhält man die Symbole für Gatter, die das Eingangssignal durchleiten und unverändert weitergeben, oder die das AND (also „UND“) von zwei Signalen berechnen, oder die das OR (also „ODER“) von zwei Signalen berechnen, ohne ein invertiertes Ergebnis zu liefern.

Solche Gatter haben wir noch nicht konstruiert, aber sie lassen sich leicht erhalten, wenn man NAND und NOT Gatter (die ja eine Art von NAND Gatter sind) kombiniert. Zum Beispiel, die Hintereinanderschaltung von zwei NOT Gattern liefert ein nichtinvertierendes Gatter, einen so genannten **Puffer**:

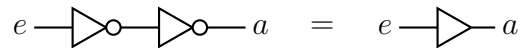


Abbildung 2.9: Ein Puffer Gatter aus zwei NOT Gattern

Sie fragen sich sicher, wozu man ein Gatter braucht, das sein Eingangssignal unverändert weiterreicht; stattdessen könnte man ja einfach eine ununterbrochene Leitung durchlegen.

Das stimmt aber nicht ganz, denn wir haben bisher nicht berücksichtigt, dass alle Gatter in einem Rechner einen zusätzlichen, bisher nicht erwähnten Eingang haben, der sie *freischaltet*. Auf diese Weise ist es zum Beispiel möglich, mit einem Uhrensinal für eine geregelte Verarbeitung von Datensignalen zu sorgen, die nur dann weiterverarbeitet werden sollen, wenn sicher ist, dass lange genug gewartet wurde, bis diese Signale (die ja in der Regel Spannungen sind) ihren endgültigen Wert erreicht haben.

Nicht freigeschaltete Bausteine sind gesperrt und geben keine Signale weiter. Ein Puffer erfüllt also den Zweck, ein Signal aufzuhalten und sozusagen „aufzubewahren“, bis die Empfangsstelle bereit ist, es anzunehmen.

Ein NAND Gatter, wie wir wissen, liefert einen Wert 1 an seinem Ausgang genau dann, wenn es *nicht* der Fall ist, dass beide Eingänge 1 sind, d. h., es ist, wie der Name sagt, ein AND Gatter mit invertiertem Ausgang. Zwar können wir die Inversion am Ausgang nicht direkt entfernen, aber wir können sie aufheben, wenn wir das Signal *noch einmal* invertieren.

In anderen Worten, die Hintereinanderschaltung eines NAND Gatters mit einem NOT Gatter an seinem *Ausgang* liefert ein Gatter, dass die AND Funktion berechnet:

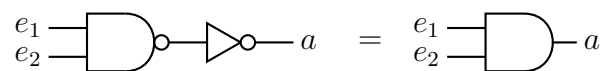


Abbildung 2.10: Ein AND Gatter aus einem NAND Gatter gefolgt durch ein NOT Gatter

Es ist leicht zu verifizieren, dass diese Schaltung tatsächlich ein AND berechnet; dazu schreibe man einfach die NAND Wertetabelle 2.1 ab und invertiere die Werte in der *a*-Spalte, da der Ausgang des NANDs ja durch ein NOT Gatter geschickt wird. Das Ergebnis ist die Wertetabelle der AND

e_1	e_2	a
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 2.4: Wertetabelle der AND Funktion

Funktion; der Ausgang ist genau dann 1, wenn beide Eingänge 1 sind.

Umgekehrt kann man aus einem AND Gatter wieder ein NAND Gatter erhalten, wenn man den Ausgang des AND Gatters durch einen Invertierer schickt.

Die Konstruktion eines OR Gatters ist nicht so offensichtlich, aber trotzdem nicht schwer. Nämlich, das NAND Gatter liefert genau dann eine 1, wenn mindestens einer seiner Eingänge, das heißt e_1 oder e_2 , eine 0 ist. Wir suchen ein Gatter, das eine 1 liefert, wenn mindestens einer seiner Eingänge eine 1 ist, und genau dieses Verhalten können wir erzeugen, wenn wir die Eingänge invertieren und dann in ein NAND Gatter füttern.

In anderen Worten, wenn wir anstelle des Ausgangs jeden der *Eingänge* eines NAND Gatters vorher durch ein NOT Gatter schicken, wird daraus ein OR Gatter.

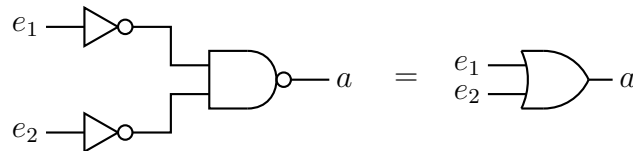


Abbildung 2.11: Ein OR Gatter aus einem NAND Gatter mit NOT Gattern vor den Eingängen

Diese Behauptung kann man wieder mit der Wertetabelle leicht nachprüfen. Die Wertetabelle der abgebildeten Schaltung erhält man aus der NAND Wertetabelle diesmal indem man die Werte in den *Eingangsspalten* invertiert. Das Ergebnis ist

e_1	e_2	a		e_1	e_2	a
1	1	1	oder umsortiert	0	0	0
1	0	1		0	1	1
0	1	1		1	0	1
0	0	0		1	1	1

Tabelle 2.5: Wertetabelle der OR Funktion

und der Ausgang ist genau dann 1, wenn mindestens einer der Eingänge 1 ist, d. h., wir haben hier die Wertetabelle der OR Funktion.

Nun stellt sich die Frage, welche weiteren booleschen Funktionen kann man mit Schaltkreisen aus bekannten Gattern „berechnen“? Wie wir sehen werden, haben wir mit dem NOT, dem AND und dem OR Gatter einen sehr geeigneten Grundsatz an Bausteinen für die Konstruktion solcher Schaltkreise.

Erstens sind diese Bausteine noch relativ einfach (sie lassen sich aus einem, bzw. aus zwei, bzw. aus drei NAND Gatter zusammensetzen), und sie lassen sich *systematisch* einsetzen zur Realisierung beliebiger boolescher Funktionen, mit Schaltkreisen begrenzter Komplexität und deshalb mit schnellen und begrenzten Schaltzeiten.

Der *systematische* Einsatz dieser Grundbausteine bedeutet, dass wir zur Realisierung weiterer boolescher Funktionen keine *ad hoc* Überlegungen mehr anstellen müssen, wie wir es gemacht haben, um das NOT, das AND und das OR Gatter aus NANDs zusammenzusetzen.

Vielmehr werden wir ein Kalkül entwickeln, mit dem wir jede boolesche Funktion direkt in eine Schaltung übersetzen können, und wir werden darüber hinaus Verfahren präsentieren, um die gewonnenen Schaltungen weitgehend zu vereinfachen und zu optimalen (d. h., billigsten) Schaltungen mit dem jeweils gewünschten Verhalten zu reduzieren.

Die erste Überlegung, die wir anstellen, ist dass man durch eine Verkettung von AND oder ODER Gattern mit zwei Eingängen auch AND und ODER Gatter mit beliebig vielen Eingängen konstruieren kann, nach folgendem Schema (hier am Beispiel von ANDs illustriert):

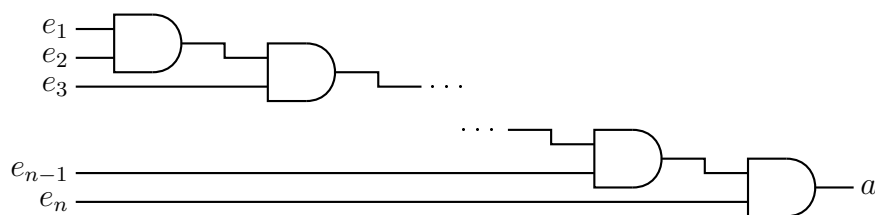


Abbildung 2.12: Ein AND mit mehreren Eingängen

Die ersten beiden Eingangssignale werden auf das erste AND geleitet, dessen Ausgang und das nächste Eingangssignal werden auf die Eingänge des nächsten ANDs geleitet, und so weiter bis schließlich der Ausgang des vorletzten ANDs und das letzte Eingangssignal auf die beiden Eingänge des letzten ANDs geleitet werden. Am Ausgang des letzten ANDs erscheint der

Wert der Funktion

$$\text{AND}\left(\text{AND}\left(\dots \text{AND}\left(\text{AND}(e_1, e_2), e_3\right), \dots e_{n-1}\right), e_n\right),$$

die offensichtlich genau dann 1 ist, wenn *alle* Eingänge 1 waren.

Den Namen dieser Funktion kürzen wir ab als

$$\text{AND}(e_1, e_2, \dots, e_n).$$

Als Symbol dafür verwenden wir ein AND Gatter mit mehr als zwei Eingängen:

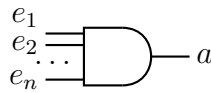


Abbildung 2.13: Ein AND mit mehreren Eingängen als einzelnes Gatter

Eine entsprechende Konstruktion und eine entsprechende Kurznotation benutzen wir auch für OR Gatter und die OR Funktion anstelle von AND.

Nach diesen kleinen Vorbereitungen sind wir nun in der Lage, ein bequemes Verfahren anzugeben, um für jede beliebige boolesche Funktion $f(e_1, e_2, \dots, e_n)$ einen Schaltkreis zu zeichnen, der n Eingänge und einen Ausgang hat, und der an seinem Ausgang die gewünschte Funktion f der Eingänge ausgibt.

Die Idee dazu besteht darin, die boolesche Funktion, die normalerweise durch ihre Wertetabelle vorgegeben ist, durch einen **booleschen Ausdruck** darzustellen, d. h., durch einen logischen Ausdruck, in dem die Eingänge e_i , als Wahrheitswerte aufgefasst (1 = WAHR und 0 = FALSCH) mit den logischen Operatoren NOT, AND und OR verbunden werden. Die Form eines solchen Ausdrucks werden wir gleich genauer beschreiben, aber durch eine Umdeutung des booleschen Ausdrucks, wobei man die Namen NOT, AND und OR jetzt als Gatternamen auffasst, erhält man unmittelbar eine Vorschrift für den Aufbau eines Schaltkreises mit dem gewünschten Verhalten.

Hier nun die genaue Definition eines booleschen Ausdrucks:

Definition 2.1 Wir verwenden für boolesche Ausdrücke eine Sprache mit:

Konstantensymbolen: 0 und 1;

Variablensymbolen: e_1, e_2, \dots , oder auch andere Kleinbuchstaben a, b, c usw., eventuell mit einem ganzzahligen Index versehen (z. B., a_2, c_1);

Operatornamen: NOT, AND und OR;

Satzsymbolen: $(,)$ und $,.$

Ein boolescher Ausdruck ist entweder

- a) ein einzelnes Konstantensymbol oder Variablensymbol, oder
- b) ein Ausdruck der Gestalt $\text{NOT}(A)$, wo A ein boolescher Ausdruck ist, oder
- c) ein Ausdruck der Gestalt $\text{AND}(A_1, A_2, \dots, A_n)$, wo jedes A_i ein boolescher Ausdruck ist, oder
- d) ein Ausdruck der Gestalt $\text{OR}(A_1, A_2, \dots, A_n)$, wo jedes A_i ein boolescher Ausdruck ist.
- e) Nichts anderes ist ein boolescher Ausdruck.

Es ist immer eindeutig feststellbar, ob ein gegebener Ausdruck ein boolescher Ausdruck ist oder nicht, denn boolesche Ausdrücke kürzer als vier Zeichen können nur Konstante oder Variablennamen sein, und bei Ausdrücken der Form b)–d) muss man die Definition zwar noch einmal auf Teilausdrücke A_i anwenden, aber diese sind kürzer als der ursprüngliche Ausdruck, so dass die verschachtelte Anwendung der Definition auf Teilausdrücke nur endlich oft wiederholt werden muss, bis nur noch Fall a) auftreten kann.

Beispiel 2.2 Ein Beispiel eines booleschen Ausdrucks ist

$$\text{NOT}\left(\text{AND}\left(\text{OR}(1, e_1), \text{NOT}(\text{AND}(1, a, c)), \text{OR}(a, \text{NOT}(b), c)\right)\right) \quad (2.1)$$

Definition 2.3 Eine *Belegung* einer booleschen Variablen ist die Zuordnung eines der Werte 0 oder 1 zu dieser Variablen (unabhängig von einer eventuellen Belegung anderer Variablen).

Für jede Belegung der in einem booleschen Ausdruck vorkommenden verschiedenen Variablen erhält der gesamte Ausdruck einen *logischen Wert* oder *booleschen Wert* 0 oder 1 nach folgendem Schema:

- a) 0 hat den Wert 0 und 1 hat den Wert 1;
- b) eine boolesche Variable hat ihre Belegung 0 oder 1 als Wert;
- c) $\text{NOT}(A)$ hat den Wert 1, wenn A den Wert 0 hat, und den Wert 0, wenn A den Wert 1 hat;
- d) $\text{AND}(A_1, A_2, \dots, A_n)$ hat den Wert 1, wenn *alle* A_i den Wert 1 haben, und sonst den Wert 0.

- e) $\text{OR}(A_1, A_2, \dots, A_n)$ hat den Wert 1, wenn *mindestens ein* A_i den Wert 1 hat, und sonst den Wert 0.

Beispiel 2.4 Der boolesche Ausdruck aus Beispiel 2.2 enthält die Variablen e_1 , a , b und c , die wir mit den Werten

$$e_1 = a = c = 0 \quad \text{und} \quad b = 1$$

belegen wollen.

Den booleschen Wert des gesamten Ausdrucks müssen wir *von innen nach außen* bestimmen, indem wir die kleinsten, innersten Teilausdrücke zuerst bewerten, damit wir anschließend die sie umschließenden Ausdrücke bewerten können.

Zunächst setzen wir für die Variablen ihre Werte aus der gewählten Belegung ein, und der Gesamtausdruck verwandelt sich in

$$\text{NOT}\left(\text{AND}\left(\text{OR}(1, 0), \text{NOT}(\text{AND}(1, 0, 0)), \text{OR}(0, \text{NOT}(1), 0)\right)\right)$$

Als nächstes bewerten wir die Teilausdrücke, in denen nur ein Operator auf ein Tupel von Konstanten angewendet wird. Diese Bewertung nach den oben genannten Vorschriften ergibt

$$\text{OR}(1, 0) = 1 \quad \text{und} \quad \text{AND}(1, 0, 0) = 0 \quad \text{und} \quad \text{NOT}(1) = 0,$$

und wenn wir diese Teilausdrücke durch ihre Werte ersetzen verwandelt sich der Gesamtausdruck in

$$\text{NOT}\left(\text{AND}(1, \text{NOT}(0), \text{OR}(0, 0, 0))\right)$$

Wieder bewerten wir die neu entstandenen Teilausdrücke, in denen nur ein Operator auf ein Tupel von Konstanten angewendet wird, und finden

$$\text{NOT}(0) = 1 \quad \text{und} \quad \text{OR}(0, 0, 0) = 0.$$

Einsetzen in den Gesamtausdruck ergibt

$$\text{NOT}(\text{AND}(1, 1, 0)).$$

Dieses Verfahren wiederholen wir noch zweimal, und weil

$$\text{AND}(1, 1, 0) = 0$$

erhalten wir für den Gesamtausdruck den Wert

$$\text{NOT}(0) = 1.$$

Hier ist eine wichtige Bemerkung angebracht. Natürlich hat der boolesche Ausdruck (2.1) eine *logische Bedeutung*, aber die Bewertung dieses Ausdrucks mit der gegebenen Belegung haben wir *rein mechanisch* durchgeführt, indem wir stur die Regeln aus Definition 2.3 angewendet haben. Zu keiner Zeit mussten wir über die logische Bedeutung des Ausdrucks wirklich nachdenken.

Das hat zur Folge, dass der Ausdruck *genau so mechanisch* von einem geeigneten Schaltkreis ausgewertet werden kann, und der boolesche Ausdruck in der obigen Schreibweise liefert sogar unmittelbar eine Vorschrift für die Konstruktion eines solchen Schaltkreises: wir müssen nur die logischen Operatoren AND, OR und NOT im booleschen Ausdruck jetzt als Namen von Gattern interpretieren und die Gatter nach folgender Regel miteinander verbinden:

Bemerkung 2.5 Sei A ein boolescher Ausdruck; man kann wie folgt einen Schaltkreis konstruieren, der den Ausdruck A auswertet.

Dieser Schaltkreis hat *einen* Ausgang und für jede verschiedene boolesche Variable, die in A erscheint, einen Eingang.

Ähnlich wie bei der Bewertung des Ausdrucks, verarbeiten wir ihn von innen nach außen, um ihn in ein Schaltkreis zu übersetzen, wobei Bestandteile des endgültigen Schaltkreises nach und nach entstehen.

- a) Die Schaltkreisübersetzung einer Konstanten 0 ist eine Leitung, die mit Masse verbunden ist.
- b) Die Schaltkreisübersetzung einer Konstanten 1 ist eine Leitung, die mit der Betriebsspannung U_H verbunden ist.
- c) Die Übersetzung einer booleschen Variablen ist eine Leitung, die mit dem der Variablen zugeordneten Eingangssignal verbunden ist.
- d) Die Übersetzung eines Teilausdrucks $\text{NOT}(A)$ ist ein NOT Gatter, dessen Eingang mit dem Ausgang der Schaltkreisübersetzung des Ausdrucks A verbunden ist.
- e) Die Übersetzung eines Teilausdrucks $\text{AND}(A_1, A_2, \dots, A_n)$ ist ein AND Gatter mit n Eingängen, wobei für jedes i der i -te Eingang mit dem Ausgang der Schaltkreisübersetzung des Ausdrucks A_i verbunden ist. Wenn $n > 2$ ist, kann man ein solches AND Gatter aus Gattern mit jeweils zwei Eingängen nach dem Schema von Abbildung 2.12 konstruieren.
- f) Die Übersetzung eines Teilausdrucks $\text{OR}(A_1, A_2, \dots, A_n)$ ist ein OR Gatter mit n Eingängen, wobei für jedes i der i -te Eingang mit dem

Ausgang der Schaltkreisübersetzung des Ausdrucks A_i verbunden ist. Wenn $n > 2$ ist, kann man ein solches OR Gatter aus OR Gattern mit jeweils zwei Eingängen nach dem Schema von Abbildung 2.12 konstruieren.

- g) Als Ausgang des ganzen Schaltkreises dient der Ausgang des Gatters (oder die Leitung), die dem äußersten Operator oder Element des Ausdrucks A zugeordnet ist.

Aus der Konstruktionsvorschrift, der Funktionsweise der Gatter und der Bewertungsvorschrift ist klar, dass dieser Schaltkreis tatsächlich an seinem Ausgang den Wert liefert, den der Ausdruck A annimmt, wenn die Variablen mit den Werten der Eingänge des Schaltkreises belegt werden. Also: der Schaltkreis „berechnet“ den booleschen Ausdruck A .

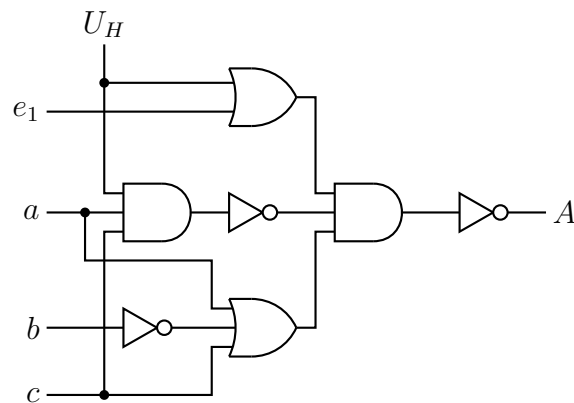


Abbildung 2.14: Die Schaltung zu Ausdruck (2.1)

In Abbildung 2.14 haben wir als Beispiel nach diesem Verfahren die Schaltung angegeben, die den Ausdruck aus Beispiel 2.2 berechnet. Im Schaltdiagramm heißt ein Eingang a , und deshalb haben wir den Ausgang diesmal mit A gekennzeichnet. Diese Schaltung wurde wieder *rein mechanisch* nach der Anleitung in Bemerkung 2.5 erstellt.

Es sollte aber auffallen, dass einige Vereinfachungen möglich sind. Zum Beispiel ist der Ausgang des oberen OR Gatters immer 1, weil ein Eingang mit der Betriebsspannung verbunden ist und immer 1 ist, und dieser Ausgang des OR Gatters wird auf ein AND Gatter weitergeleitet, dessen Ausgangswert es überhaupt nicht verändert.

Der mit U_H verbundene Eingang des linken AND Gatters beeinflusst auch dessen Wert nicht.

Deshalb könnte man alle Verbindungen zu U_H und den oberen OR Gatter und seine Verbindungen ganz aus dem Schaltkreis entfernen.

Solche Vereinfachungen führt man am besten aber nicht am Schaltdiagramm aus, sondern schon vorher, an dem booleschen Ausdruck, den das Schaltdiagramm berechnet. Wir werden auch später sehen, wie man schon auf diesem Niveau boolesche Ausdrücke weitgehend optimieren kann, ohne ihren Wert zu verändern.

Trotz dieser kleinen Probleme haben wir aber schon einen wesentlichen Schritt in Richtung auf eine Lösung unserer Aufgabe, boolesche Funktionen durch Schaltungen zu berechnen, geleistet. So lange wir die boolesche Funktion durch einen booleschen Ausdruck darstellen können, dessen Werte bei jeder Belegung die Werte der booleschen Funktion bei diesen Variablenwerten genau wiedergeben, können wir aus dem booleschen Ausdruck direkt eine funktionierende Schaltung gewinnen.

Wir müssen aber noch überlegen, ob man wirklich *jede* booleschen Funktion durch einen entsprechenden booleschen Ausdruck darstellen kann.

In der Tat kann man das, aber bevor wir sagen, wie, lohnt es sich, die klobige Notation, die wir bisher für boolesche Ausdrücke verwendet haben, etwas lesbarer und „handhabbarer“ zu machen.

Dazu nehmen wir folgende Veränderungen vor.

- Die Anwendung des NOT-Operators in einem booleschen Ausdruck kennzeichnen wir durch einen Querstrich quer über den ganzen Teilausdruck, auf den das NOT angewendet wird, d. h., anstelle von $\text{NOT}(A)$ schreiben wir \overline{A} .
- Die AND und OR Operatoren ersetzen wir durch eine Infix-Notation, d. h., durch binäre Operationen, die wir *zwischen* den Operanden schreiben, und zwar:

- statt $\text{AND}(A_1, A_2, \dots, A_n)$ schreiben wir $A_1 \wedge A_2 \wedge \dots \wedge A_n$;
- statt $\text{OR}(A_1, A_2, \dots, A_n)$ schreiben wir $A_1 \vee A_2 \vee \dots \vee A_n$.

Das Zeichen \wedge ist das übliche mathematische Zeichen für *und*, und Mathematiker können es sich leicht merken, weil es an das Zeichen \cap für einen *Mengendurchschnitt* erinnert. Das Zeichen \vee ist das übliche mathematische Zeichen für *oder*, und Mathematiker können es sich leicht merken, weil es an das Zeichen \cup für eine *Mengenvereinigung* erinnert.

Wenn die Operatoren AND und OR im booleschen Ausdruck verschachtelt sind, d. h., wenn Argumente eines solchen Operators wieder Anwendungen dieser Operatoren sind, dann muss man Klammern um die Umnotierung

der Argumente setzen, um deutlich zu machen, in welcher Reihenfolge die Operationen \wedge und \vee anzuwenden sind.

Andererseits gibt es aber eine Prioritätsregel, die besagt, dass \wedge Verknüpfungen vor \vee Verknüpfungen auszuführen sind (ähnlich wie die Regel „Punktrechnung geht vor Strichrechnung“), und diese Regel macht es manchmal möglich, Klammern wegzulassen. So ist der Ausdruck

$$A \wedge B \vee C$$

eindeutig als

$$(A \wedge B) \vee C$$

zu lesen (und nicht anders geklammert), so dass wir die Klammern hier nicht schreiben müssen (trotzdem dürfen wir sie natürlich schreiben, und das wäre oft zu empfehlen, um die Struktur des Ausdrucks deutlicher hervortreten zu lassen).

In dieser Notation schreibt sich der boolesche Ausdruck aus Beispiel 2.2 als

$$\overline{(1 \vee e_1) \wedge \overline{(1 \wedge a \wedge c)} \wedge (a \vee \bar{b} \vee c)} \quad (2.2)$$

Man sieht, dass diese Notation viel klarer gegliedert und lesbarer ist, als die Operatornotation.

Es gibt aber eine noch bessere und effizientere Notation, gerade wenn man, wie wir, vorhat, viel mit booleschen Ausdrücken zu rechnen, und das ist die algebraische Notation der **booleschen Algebra** oder **Schaltalgebra**, erfunden vom englischen Logiker George Boole (1815–1864).

Diese Notation unterscheidet sich von der gerade vorgeführten logischen Notation nur durch eine Umbenennung der Operationen \wedge und \vee : in der booleschen Algebra schreibt man \cdot statt \wedge und $+$ statt \vee , so dass logische Ausdrücke (fast) so aussehen, wie algebraische Ausdrücke (der NOT Operator wird aber weiterhin mit einem Querstrich notiert).

Obwohl dies formal keine wesentliche Änderung ist, macht sie es trotzdem leichter und bequemer, mit logischen Ausdrücken zu rechnen. Dazu trägt sicher bei, dass viele (aber nicht alle!) Regeln der normalen Algebra auch in der booleschen Algebra gelten.

Zwei Konventionen werden aus der normalen Algebra übernommen. Die eine ist die Praxis, den Produktoperator \cdot meistens nicht explizit zu schreiben, sondern Produkte einfach durch lückenloses Nebeneinandersetzen von Variablen zu notieren. Die andere ist die schon aus der Schule bekannte Regel, jetzt wieder wörtlich zu verstehen, dass „Punktrechnung vor Strichrechnung geht,“ so dass man bei Summen von Produkten keine Klammern um die Produkte schreiben muss. Diese Regel entspricht genau der schon genannten

Prioritätsregel zwischen \wedge und \vee , aber die algebraische Notation macht es viel leichter, die Regel richtig herum zu behalten.

In der algebraischen Notation schreibt sich der boolesche Ausdruck aus Beispiel 2.2 als

$$\overline{(1 + e_1)(1ac)(a + \bar{b} + c)} \quad (2.3)$$

Dieser Ausdruck schreit nach einigen Vereinfachungen (z.B., bezüglich des Faktors 1 in einem Produkt), aber diese können wir erst vornehmen, wenn wir die Rechenregeln der booleschen Algebra erläutert haben.

Die „Grundrechenarten“ der booleschen Algebra können wir aber jetzt schon durch Tabellen beschreiben (der Beweis für die Richtigkeit dieser Tabellen liegt einfach in der logischen Bedeutung der booleschen Zahlen 0 und 1 und der booleschen Operationen \neg , $+$ und \cdot).

a	\bar{a}	$+$	0	1	\cdot	0	1
0	1	0	0	1	0	0	0
1	0	1	1	1	1	0	1

Tabelle 2.6: Boolesche Rechentabellen

Man beachte, dass $a + b$ der „numerisch größere“, also das „Maximum“ der beiden Summanden ist, und dass ab der „numerisch kleinere“, also das „Minimum“ der beiden Faktoren ist.

Mit dieser Notation können wir nun bequem beschreiben, wie man zu jeder beliebigen booleschen Funktion $f(e_1, \dots, e_n)$, gegeben durch ihre Wertetabelle, einen booleschen Ausdruck finden kann, der diese Funktion beschreibt.

Angenommen, in der Wertetabelle von f gibt es genau *eine* Zeile mit einer 1 in der letzten Spalte (wo die Werte von f stehen). Dann gibt es nur *eine* Konstellation aus 0- und 1-Werten für die Variablen e_i , bei der f den Wert 1 annimmt.

Für jedes i ($1 \leq i \leq n$) sei $E_i = \bar{e}_i$, wenn e_i in der oben genannten Konstellation von Werten den Wert 0 hat, und sei $E_i = e_i$, wenn e_i in der oben genannten Konstellation von Werten den Wert 1 hat. Dann hat jedes E_i bei dieser Konstellation von Werten den Wert 1, aber das gilt *nur* bei dieser speziellen Konstellation von Werten der e_i .

Also hat das boolesche Produkt $E_1 E_2 \cdots E_n$ bei dieser Konstellation und *nur* bei dieser Konstellation von Werten der e_i den Wert 1, genau wie die boolesche Funktion f .

Das bedeutet, dass der boolesche Ausdruck $E_1 E_2 \cdots E_n$ für alle Belegungen der e_i die gleichen Werte hat, wie die boolesche Funktion f , d.h., er

beschreibt diese boolesche Funktion genau.

Im allgemeinen enthält die Wertespalte einer booleschen Funktion f aber mehr als eine einzige 1. Das ist aber kein Problem: in diesem Fall bilde man für *jede* Zeile der Wertetabelle, in der in der letzten Spalte eine 1 steht, den Produktterm wie oben beschrieben, und dann bilde man die (boolesche) *Summe* dieser Produktterme.

Da die Summe dem OR der Produktterme entspricht, ist die Summe genau dann 1, wenn mindestens einer der Summanden 1 ist, also wenn mindestens eine der Wertekonstellationen für die e_i gegeben ist, für die f den Wert 1 annimmt. Tritt aber eine andere als eine dieser Wertekonstellationen auf, dann hat f auch nicht den Wert 1.

Das heißt, die beschriebene Summe von Produkttermen hat genau dann den Wert 1, wenn auch f den Wert 1 hat, und wieder haben wir einen booleschen Ausdruck gefunden, dessen Wert für jede Belegung der Variablen mit dem entsprechenden Wert von f übereinstimmt, und der in diesem Sinne f genau beschreibt.

Übrigens, wenn in *keiner* Zeile der Wertetabelle eine 1 steht, dann hat f immer den Wert 0 und wird von der booleschen Konstanten 0 dargestellt!

Wir führen diese Methode an einem Beispiel vor:

Beispiel 2.6 Sei f die boolesche Funktion mit der Wertetabelle

e_1	e_2	e_3	a
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

In dieser Tabelle gibt es drei Zeilen mit einer 1 in der letzten Spalte, entsprechend der Wertekonstellationen 010, 100 und 101 für e_1 , e_2 und e_3 .

Für jede dieser Wertekonstellationen bilden wir das Produkt der e_i , wobei wir e_i mit einem Querstrich versehen, wenn sein Wert in der Wertekonstellation 0 ist.

Die erste Wertekonstellation 010 liefert den Produktterm $\overline{e_1}e_2\overline{e_3}$.

Die zweite Wertekonstellation 100 liefert den Produktterm $e_1\overline{e_2}\overline{e_3}$.

Die dritte Wertekonstellation 101 liefert den Produktterm $e_1\overline{e_2}e_3$.

Die Summe dieser Produkte ist ein boolescher Ausdruck

$$\overline{e_1}e_2\overline{e_3} + e_1\overline{e_2}\overline{e_3} + e_1\overline{e_2}e_3, \quad (2.4)$$

der die boolesche Funktion f genau beschreibt.

Sollte diese Funktion durch einen Schaltkreis berechnet werden, so könnte man den booleschen Ausdruck (2.4) sofort in einen Ausdruck in Operator-schreibweise übersetzen:

$$\begin{aligned} \text{OR} \Big(& \text{AND}(\text{NOT}(e_1), e_2, \text{NOT}(e_3)), \\ & \text{AND}(e_1, \text{NOT}(e_2), \text{NOT}(e_3)), \\ & \text{AND}(e_1, \text{NOT}(e_2), e_3) \Big), \quad (2.5) \end{aligned}$$

und aus diesem wie in Bemerkung 2.5 den gewünschten Schaltkreis gewinnen.

Allerdings wäre dies nicht der effizienteste Schaltkreis zur Berechnung dieser Funktion! Dieses Problem werden wir gleich diskutieren.

Festzuhalten sind folgende Punkte:

- Wir haben jetzt ein Rechenverfahren oder Algorithmus, mit dem wir jeder boolesche Funktion einen äquivalenten booleschen Ausdruck in der Schreibweise der booleschen Algebra zuordnen können.
- Die boolesche Algebra ist nur eine von mehreren äquivalenten Schreibweisen für logische Ausdrücke, und den oben erhaltenen Ausdruck kann man sofort und ohne Mühe in einen äquivalenten Ausdruck in der Operatorschreibweise mit NOT, AND und OR übersetzen.
- Diesen Ausdruck in der Operatorschreibweise kann man direkt umdeuten als eine Bauanleitung für einen Schaltkreis mit NOT, AND und OR Gattern, der die gleiche boolesche Funktion berechnet.
- In anderen Worten, wir haben eine effektive und einfache Methode, um beliebige boolesche Funktionen durch Schaltkreise zu realisieren.

Allerdings ist diese Methode noch nicht sehr effizient, d.h., die gewonnenen Schaltkreise sind oft viel größer, als sie sein müssten, um die gewünschte Funktion zu berechnen.

Zum Beispiel kann man den oben berechneten booleschen Ausdruck (2.4) verkürzen zu

$$\overline{e_1}e_2\overline{e_3} + e_1\overline{e_2},$$

ohne seinen Wert zu verändern (rechnen Sie es nach!), und das bedeutet, dass man die boolesche Funktion aus Beispiel 2.6 genau so gut mit einem kleineren

Schaltkreis aus wenigeren und einfacheren Gattern (ANDs und ORs mit zwei statt drei Eingängen) hätte realisieren können.

Aus diesem Grund wollen wir untersuchen, wie man boolesche Ausdrücke am besten optimieren kann, zu einer möglichst kurzen äquivalenten Form, aus der man dann sparsame Schaltkreise gewinnen kann.

Wir beginnen, indem wir uns ein bisschen vertrauter machen mit den Rechenregeln für boolesche Ausdrücke.

Unten steht eine Grundliste wichtiger Regeln. Das Gleichheitszeichen in diesen Gleichungen bedeutet **logische Äquivalenz**, d.h., dass die booleschen Ausdrücke auf beiden Seiten der Gleichung bei *jeder* Belegung der Variablen die gleichen Werte annehmen, also die gleiche boolesche Funktion beschreiben.

Die algebraische Notation lässt viele dieser Regeln so aussehen, wie bekannte Regeln der Algebra, was das Behalten der Regeln sehr erleichtern kann aber auch zu falschen Schlüssen verführen kann. Achten Sie deshalb bitte besonders auf die Fälle, wo die Regeln *anders* sind, als man in der Algebra gewohnt ist, oder wo der NOT-Operator eingeht, der in der normalen Algebra kein Gegenstück hat (zwar wird die komplexe Konjugation genau so notiert, aber sie verhält sich ganz anders!).

Rechenregeln 2.7 Für alle booleschen Größen a , b und c (die die booleschen Werte 0 und 1 annehmen können) gilt:

$\overline{\overline{a}} = a$	(Negation der Negation)
$ab = ba$	(Kommutativgesetz)
$a(bc) = (ab)c$	(Assoziativgesetz)
$a(b + c) = ab + ac$	(Distributivgesetz)
$aa = a$	(Idempotenz)
$a + \overline{a} = 1$	(Komplementgesetz)
$0a = 0$	(Nullgesetz)
$1a = a$	(Einsgesetz)
$ab + a\overline{b} = a$	(Absorptionsgesetz)
$\overline{ab} = \overline{a} + \overline{b}$	(De Morgan'sches Gesetz)

Zusätzlich gilt ein **Dualitätsprinzip** — um es zu formulieren, definieren wir die Operationen $+$ und \cdot als **dual** zueinander, und wir nennen die booleschen Werte 0 und 1 **dual** zueinander. Dann gilt:

Jede gültige Gleichung bleibt gültig, wenn man in ihr alle Vorkommnisse von $+$, \cdot , 0 und 1 durch ihre Dualen ersetzt.

Beweis. Alle Regeln in der Liste kann man leicht beweisen, indem man die Wertetabellen beider Seiten der Gleichungen vergleicht und feststellt, dass sie übereinstimmen. Aber die meisten Regeln lassen sich auch einfacher nachprüfen, wie wir jetzt kurz erläutern.

Die erste Regel ist klar, weil NOT die Werte 0 und 1 vertauscht; macht man es zweimal, bleiben die Werte, wie sie ursprünglich waren.

Weil die boolesche Multiplikation die AND Operation ist, ist ein Produkt genau dann 1, wenn alle Faktoren 1 sind. Das gilt auch, wie man leicht überlegt, wenn das Produkt durch Klammern gegliedert ist. Daraus sind das Kommutativgesetz und das Assoziativgesetz sofort klar.

Das Nullgesetz und das Einsgesetz folgen unmittelbar aus der Multiplikationstabelle 2.6; ebenso das Idempotenzgesetz.

Jetzt lässt sich das Distributivgesetz schnell nachprüfen: wenn $a = 0$, dann sind die linke Seite dieses Gesetzes und beide Summanden auf der rechten Seite 0; laut Additionstabelle 2.6 ist auch die Summe auf der rechten Seite 0 und die Gleichung gilt.

Wenn $a = 1$, dann folgt aus dem Einsgesetz, dass beide Seiten des Distributivgesetzes den Wert $b + c$ haben, und deshalb gleich sind.

Das Komplementgesetz folgt sofort aus der Additionstabelle, denn die linke Seite hat den Wert $0 + 1$ oder den Wert $1 + 0$, und beides ergibt 1.

Das Absorptionsgesetz folgt sofort aus dem Distributivgesetz, dem Komplementgesetz, dem Kommutativgesetz und dem Einsgesetz, denn es gilt

$$ab + a\bar{b} = a(b + \bar{b}) = a1 = 1a = a.$$

Das de Morgan'sche Gesetz kann man mit der Wertetabelle nachprüfen, oder nach der folgenden einfachen Überlegung: die linke Seite \overline{ab} ist genau dann 0, wenn $ab = 1$, und das gilt laut Multiplikationstabelle genau dann, wenn a und b Eins sind. Die rechte Seite ist genau dann 0 laut Additionstabelle, wenn beide Summanden \bar{a} und \bar{b} Null sind, und das ist wieder genau dann der Fall, wenn a und b Eins sind.

Beide Seiten sind also unter den gleichen Bedingungen 0, und deshalb auch unter den gleichen Bedingungen 1. Somit sind sie logisch äquivalent.

Die Dualität ist eine sofortige Konsequenz der de Morgan Regel, wie wir jetzt darlegen werden.

Wir beginnen mit der Beobachtung, dass jede richtige boolesche Gleichung auch richtig bleibt, wenn man in ihr alle Variablen negiert (also komplementiert), denn die negierten Variablen haben *insgesamt* das gleiche Wertespektrum, als die unnegierten. Sie haben zwar in jeder *Instanz* andere Werte, aber das macht nichts, denn eine Gleichung ist genau dann richtig, wenn sie für *alle* Belegungen der Variablen stimmt. Jede Belegung der negierten

Variablen kommt auch bei den unnegierten vor, nur zu einem anderen „Zeitpunkt“; trotzdem gilt die Gleichung für diese Belegung, weil sie ja „immer“ gilt.

Ferner bleibt natürlich jede richtige Gleichung richtig, wenn man beide Seiten der Gleichung als Ganzes negiert, denn das ändert nichts daran, dass sie zu jeder Belegung gleiche boolesche Werte haben.

Mit diesen beiden Beobachtungen erhalten wir zunächst die duale Version der de Morgan Regel, denn aus der Version, die oben angegeben ist, folgt nach den vorgeschlagenen Modifizierungen

$$\overline{\overline{ab}} = \overline{\overline{a} + \overline{b}}$$

und wenn wir die doppelten Negationen entfernen (nach unseren bisherigen Regeln erlaubt) erhalten wir ein neues de Morgan'sches Gesetz

$$\overline{ab} = \overline{a + b},$$

das Duale des anderen (wenn man die Seiten vertauscht).

Unter Anwendung beider Versionen des de Morgan'schen Gesetzes können wir nun zeigen, dass aus jeder gültigen Gleichung ihre duale Version folgt.

Dazu negieren wir zuerst beide Seiten der Gleichung, wobei die Gleichung ihre Gültigkeit bewahrt.

Durch wiederholte Anwendung der de Morgan Regeln können wir die gerade vorgenommene Negation Stufe um Stufe in den Ausdruck hineinziehen, bis nur die Variablen negiert sind (natürlich abgesehen von Negationen, die schon in der ursprünglichen Gleichung vorhanden waren — diese werden beim „hereinziehen“ nicht verändert und bleiben bestehen!). Dabei wird durch die de Morgan'schen Gesetze jedes Plus in ein Mal verwandelt und umgekehrt, und jede Null wird in eine Eins verwandelt und umgekehrt. Die Gleichung bleibt bis zum Schluss gültig, weil wir bei jedem Schritt einen Teilausdruck durch einen anderen aber gleichwertigen Ausdruck ersetzt haben.

Am Ende steht die dualisierte Gleichung da, allerdings mit negierten Variablen. Aber wir haben schon gesehen, dass wir die Negation der Variablen entfernen können, ohne die Gültigkeit der Gleichung zu beeinträchtigen. ■

Der Vollständigkeit halber führen wir noch die dualen Regeln auf; bewiesen haben wir sie schon. Die erste Regel können wir weglassen, weil sie „selbstdual“ ist und durch Dualisierung nicht verändert wird.

Einige dieser neuen Regeln (zum Beispiel das duale Distributivgesetz und das duale Absorptionsgesetz) sind etwas überraschend, wenn nicht gar kontraintuitiv!

Rechenregeln 2.8 Für alle booleschen Größen a , b und c (die die booleschen Werte 0 und 1 annehmen können) gilt:

$$\begin{array}{ll}
 a + b = b + a & \text{(Kommutativgesetz)} \\
 a + (b + c) = (a + b) + c & \text{(Assoziativgesetz)} \\
 a + bc = (a + b)(a + c) & \text{(Distributivgesetz)} \\
 a + a = a & \text{(Idempotenz)} \\
 a\bar{a} = 0 & \text{(Komplementgesetz)} \\
 1 + a = 1 & \text{(Einsgesetz)} \\
 0 + a = a & \text{(Nullgesetz)} \\
 (a + b)(a + \bar{b}) = a & \text{(Absorptionsgesetz)} \\
 \overline{a + b} = \bar{a}\bar{b} & \text{(De Morgan'sches Gesetz)}
 \end{array}$$

Die jetzt gewonnenen Kenntnisse über das boolesche Rechnen werden nützlich sein bei der Optimierung von digitalen Schaltkreisen.

Wir erinnern uns, dass wir jetzt schon in der Lage sind, jede boolesche Funktion durch eine Schaltung zu realisieren, deren „Bauanleitung“ ein boolescher Ausdruck ist, der die boolesche Funktion beschreibt. Das Verfahren zur Gewinnung dieses booleschen Ausdrucks steht auf Seite 59 und Seite 60.

Dieses Verfahren geht von der Wertetabelle der booleschen Funktion aus und liefert einen booleschen Ausdruck, der einerseits eine schöne und für die Anwendung im Rechnerdesign günstige Gestalt hat (welche das ist, und warum sie günstig ist, werden wir gleich sagen), aber der in der Regel unnötig kompliziert ist (und deshalb doch nicht so günstig ist — zu komplizierte boolesche Ausdrücke führen zu Schaltkreisen mit unnötig vielen Gattern, und solche Schaltkreise schalten langsamer und sind teurer als effizientere).

Wir wollen uns deshalb nicht zufrieden geben mit den booleschen Ausdrücken, die unser bisheriges Verfahren für eine gegebene boolesche Funktion liefert, sondern wir haben einige Ansprüche, die nach Möglichkeit erfüllt werden sollen. Wir suchen boolesche Ausdrücke, die eine für den Rechnerbau geeignete Gesamtstruktur haben sollen (ein wesentliches Kriterium ist hier die Länge des Signalwegs durch die Schaltung, die direkt einen Einfluss auf die Schaltgeschwindigkeit hat) und gleichzeitig möglichst kurz sein sollen, weil dann die entsprechende Schaltung billig, sparsam im Betrieb, zuverlässig und schnell ist.

Wir beschreiben jetzt Methoden, um durch Umwandlung von booleschen Ausdrücken diese Ziele zu erreichen.

Dazu führen wir etwas Terminologie ein:

Definition 2.9 Ein *Konjunktionsterm* (der Name bedeutet: *und*-Term) ist ein boolescher Ausdruck der Gestalt

$$X_1 X_2 \dots X_n,$$

wo jedes X_i eine unnegierte oder negierte boolesche Variable ist und keine zwei X_i die gleiche Variable enthalten. Hier darf die Anzahl n der Faktoren auch 0 sein; in diesem Fall ist der Term nach Konvention die Konstante 1.

In der Notation der booleschen Algebra sieht ein Konjunktionsterm also aus wie ein Produktterm.

Es gibt dazu einen dualen Begriff: Ein *Disjunktionsterm* (der Name bedeutet: *oder*-Term) ist ein boolescher Ausdruck der Gestalt

$$X_1 + X_2 + \dots + X_n,$$

wo jedes X_i eine unnegierte oder negierte boolesche Variable ist und keine zwei X_i die gleiche Variable enthalten. Hier darf die Anzahl n der Faktoren auch 0 sein; in diesem Fall ist der Term nach Konvention die Konstante 0.

In der Notation der booleschen Algebra sieht ein Disjunktionsterm also aus wie eine Summe von Variablen oder negierten Variablen.

Disjunktionsterme dieser Gestalt werden uns nicht so sehr interessieren, sondern vielmehr eine Disjunktion von *Konjunktionstermen* (anstelle von Variablen) der folgenden Art:

Eine *disjunktive Normalform* (kurz **DN**) ist ein boolescher Ausdruck der Gestalt

$$K_1 + K_2 + \dots + K_m,$$

wo die K_i paarweise nichtäquivalente Konjunktionsterme sind. Auch 0 (die Summe von keinen Summanden) ist eine disjunktive Normalform.

Disjunktive Normalformen sind also Summen von Produkten in der Schreibweise der booleschen Algebra, und alle Summanden müssen verschieden sein.

Beispiel 2.10 Beispiele von disjunktiven Normalformen sind

$$\overline{a}bc + ac + a\overline{b} \quad \text{oder} \quad x_1\overline{x_2} + x_2\overline{x_3}.$$

Man beachte, dass nicht jede Variable in jedem Summanden vorkommen muss. Dazu mehr gleich.

Disjunktive Normalformen haben große Vorteile für den Rechnerbau. Nämlich, wenn man sie in der Operatorform schreibt, dann ist der äußere Operator ein OR (der eventuell sogar fehlen kann, wenn es keine oder nur einen Summanden im algebraischen Ausdruck gibt).

Die Argumente des ORs sind Anwendungen des AND Operators, und deren Argumente sind entweder Variablen oder schlimmstenfalls eine Anwendung eines NOT Operators auf eine Variable.

Der gesamte Aufbau ist also höchstens dreistufig, und das gleiche gilt natürlich für die digitale Schaltung, die man aus einem solchen booleschen Ausdruck bauen würde. Als Beispiel zeigen wir in Abbildung 2.15 die Schaltung für den DN-Ausdruck $\bar{a}bc + \bar{a}c$.

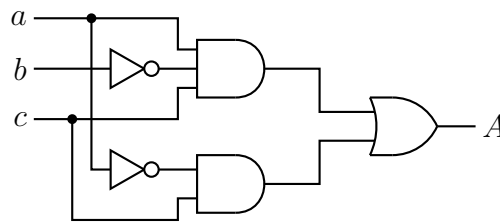


Abbildung 2.15: Eine DN Schaltung, für den Ausdruck $\bar{a}bc + \bar{a}c$.

Die innerste Stufe des booleschen Ausdrucks in der Operatorschreibweise ist ein NOT, die nächste Stufe ein AND und die äußerste ein OR. Für die digitale Schaltung bedeutet das, dass jedes Eingangssignal im schlimmsten Fall nur ein NOT Gatter, ein AND Gatter und ein OR Gatter durchlaufen muss, bis es am Ausgang der Schaltung ankommt.

Das ist der längste Weg, den das Signal durchlaufen muss, und deshalb ist auch die Zeit, die dazu erforderlich ist, begrenzt und kurz. Boolesche Ausdrücke in digitaler Normalform liefern also *schnelle Schaltungen*.

Betrachten wir einen modernen Prozessor mit einer Taktfrequenz von etwa 4 GHz; ein Takt dauert dann 250 psec. Die Schaltdauer eines FET liegt bei etwa 50 psec; bei einer Schaltung mit so wenigen Stufen wie gerade beschrieben ist es also durchaus gerade noch realisierbar, ein Signal in einem einzigen Takt die Schaltung durchlaufen zu lassen.

Wir werden es uns deshalb zum Prinzip machen, boolesche Funktionen und boolesche Ausdrücke bevorzugt in disjunktiver Normalform zu schreiben.

Ist das denn immer möglich? Natürlich ist es das! denn wir haben ja schon ein Verfahren angegeben, um jede boolesche Funktion durch einen booleschen Ausdruck zu beschreiben, und dieses Verfahren liefert immer eine disjunktive Normalform.

Nur: diese kann unnötig kompliziert sein, und deshalb eine unnötig große Anzahl von Gattern zur Realisierung als Schaltung verlangen. Außerdem sind viele „Schaltprobleme“ von vornherein in der Gestalt eines booleschen Ausdrucks vorgegeben (oder als die alltagssprachliche Umschreibung eines booleschen Ausdrucks) und nicht als die vollständige Wertetabelle einer booleschen Funktion. Man möchte deshalb gerne eine direktere Methode haben,

um boolesche Ausdrücke in DN Form zu bringen, als zuerst die Wertetabelle der zugehörigen booleschen Funktion erstellen zu müssen.

Eine solche direkte Methode lässt sich mit Hilfe der booleschen Rechenregeln leicht ausarbeiten, aber bevor wir das tun wollen wir „voraussehend“ warnen, dass das erste Ergebnis, obwohl es kurze Formeln liefert, sich doch nicht als ideal herausstellen wird, weil gerade die Kürze verhindert, dass man weitere mögliche Optimierungen erkennt!

Es wird deshalb eine zweite Methode folgen, die paradoxerweise zunächst ganz lange „vollständige“ Formen produziert, die anschließend perfekter optimiert werden können.

Rechenverfahren 2.11 Einen beliebigen booleschen Ausdruck A kann man immer mit folgenden Schritten in disjunktive Normalform bringen (hierbei ist jeder Schritt so oft durchzuführen, bis er nicht mehr anwendbar ist):

Schritt 1.

Man wende die de Morgan Regeln wiederholt an, um alle Negationen auf zusammengesetzte Teilausdrücke in die Ausdrücke hineinzuziehen und auf die Argumente zu verteilen (d. h., man ersetze \overline{BC} durch $\overline{B} + \overline{C}$ und $\overline{B + C}$ durch $\overline{B} \overline{C}$), bis schließlich nur noch Variablen und Konstanten negiert sind.

Schritt 2.

Man entferne alle Doppelnegationen.

Schritt 3.

Man ersetze alle negierten Konstanten durch ihren Wert, also $\overline{0}$ durch 1 und $\overline{1}$ durch 0.

Schritt 4.

In geklammerten Teilausdrücken, wo die Klammern gleichartige binäre Operationen (also nur Multiplikationen oder nur Additionen) trennen, in anderen Worten, in Teilausdrücken der Form $B(CD)$ oder $(BC)D$ oder $B + (C + D)$ oder $(B + C) + D$, entferne man die Klammern. Sie sind auf Grund der Assoziativgesetze überflüssig.

Schritt 5.

Produkte von Summen multipliziere man aus mittels des Distributivgesetzes, d. h., man ersetze $B(C + D)$ durch $BC + BD$ und $(B + C)D$ durch $BD + CD$. Nach diesem Schritt hat der Ausdruck die Gestalt einer Summe von Produkten.

Schritt 6.

Klammern um ein Produkt kann man wegen der Präzedenzregel entfernen.

Schritt 7.

Man wähle eine beliebige aber feste „alphabetische Reihenfolge“ der Variablennamen und der Konstantennamen 0 und 1 und in jedem Produkt sortiere man die Faktoren alphabetisch. Das darf man machen wegen des Kommutativgesetzes.

Schritt 8.

Jedes Vorkommen von $a\bar{a}$ oder von $\bar{a}a$, wo a ein Variablenname ist, ersetze man durch 0 (nach dem Komplementgesetz).

Schritt 9.

Wenn ein Produktterm einen Faktor 0 hat, entferne man den ganzen Produktterm aus der Summe von Produkten, denn nach dem Nullgesetz ist sein Wert 0 und er verändert die Summe der anderen Summanden nicht. Wenn nach diesem Verfahren keine Summanden mehr vorhanden sind, hat A die DN 0.

Schritt 10.

Wenn ein Produktterm einen Faktor 1 hat, lasse man diesen Faktor weg (es sei denn, er ist der einzige Faktor!), denn nach dem Einsgesetz verändert er das Produkt der anderen Faktoren nicht.

Schritt 11.

Wenn in einem Produktterm eine Variable a mehrmals vorkommt, ersetze diese Vorkommnisse durch eine einzige, d.h., ersetze aa durch a (nach dem Idempotenzgesetz). Nach diesem Schritt ist jeder Produktterm ein Konjunktionsterm.

Schritt 12.

Wenn in der Summe von Produkten ein Summand 1 ist, hat A die DN 1 (nach dem Einsgesetz).

Schritt 13.

Wenn in der Summe von Produkten zwei Summanden gleich sind, entferne ein Exemplar. Nach diesem Schritt steht A in disjunktiver Normalform da.

Beispiel 2.12 Wir wenden dieses Verfahren auf den booleschen Ausdruck aus Beispiel 2.2 an, der in der algebraischen Schreibweise die Gestalt

$$\overline{(1 + e_1)(1ac)(a + \bar{b} + c)}$$

aus Formel (2.3) hat.

Die erste Anwendung von Schritt 1 verwandelt dies in

$$\overline{(1 + e_1)} + \overline{(1ac)} + \overline{(a + \bar{b} + c)},$$

eine zweite Anwendung in

$$(\bar{1} \bar{e}_1) + \overline{(\bar{1} + \bar{a} + \bar{c})} + (\bar{a} \bar{\bar{b}} \bar{\bar{c}}),$$

eine letzte Anwendung schließlich in

$$(\bar{1} \bar{e}_1) + (\bar{\bar{1}} \bar{\bar{a}} \bar{\bar{c}}) + (\bar{a} \bar{\bar{b}} \bar{\bar{c}}).$$

Nach Schritt 2 haben wir

$$(\bar{1} \bar{e}_1) + (1ac) + (\bar{a}b\bar{c})$$

und nach Schritt 3

$$(0\bar{e}_1) + (1ac) + (\bar{a}b\bar{c})$$

Schritt 5 muss nicht angewendet werden, und nach Schritt 6 haben wir

$$0\bar{e}_1 + 1ac + \bar{a}b\bar{c}$$

Schritte 7 und 8 müssen nicht angewendet werden, und nach Schritt 9 haben wir

$$1ac + \bar{a}b\bar{c},$$

nach Schritt 10 dann

$$ac + \bar{a}b\bar{c}.$$

Die restlichen Schritte entfallen und wir haben den ursprünglichen Ausdruck in disjunktive Normalform gebracht und dabei wesentlich vereinfacht.

Beispiel 2.13 Hier ein weiteres Beispiel; wir bringen den Ausdruck

$$a(\overline{ab}) \tag{2.6}$$

in disjunktive Normalform.

Schritt 1 verwandelt dies in

$$a(\bar{a} + \bar{b}).$$

Als nächstes ist Schritt 5 anzuwenden (die Schritte 2–4 müssen nicht und können nicht ausgeführt werden). Wir multiplizieren das Produkt aus und erhalten

$$a\bar{a} + a\bar{b}.$$

Mit Schritt 8 wird hieraus

$$0 + a\bar{b}$$

und Schritt 9 entfernt den ersten Summanden.

Die restlichen Schritte entfallen und wir erhalten die disjunktive Normalform

$$a\bar{b}.$$

Diese Beispiele lassen hoffen, dass unser Verfahren, um boolesche Ausdrücke in disjunktive Normalform zu bringen, dazu geeignet ist, in der Regel komplizierte vorgegebene Ausdrücke wesentlich zu straffen und effiziente kurze Normalformen zu liefern.

Aber in Wirklichkeit ist das Verfahren gar nicht auf Optimalität ausgelegt und kann diese Hoffnung nicht erfüllen. Hier ein paar einfache Beispiele.

Die disjunktive Normalform

$$\bar{a}bc + ac + a\bar{b}$$

aus Beispiel 2.10 ist äquivalent zum wesentlich kürzeren Ausdruck

$$a\bar{b} + bc,$$

obwohl das nicht offensichtlich ist. Es lässt sich aber leicht mit den Wertetabellen bestätigen.

Auch die disjunktive Normalform

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

deren Schaltdiagramm in Abbildung 2.15 abgebildet ist, lässt sich ein bisschen kürzen zu

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

was im Schaltdiagramm zumindest ein Eingang eines AND Gates einspart (und dadurch die Leistungsaufnahme vermindert).

Das Verfahren, um Ausdrücke in disjunktive Normalform zu bringen, würde diese Ausdrücke überhaupt nicht verändern, da sie schon in disjunktiver Normalform sind. Das Verfahren ist ja nur darauf ausgelegt, Verletzungen der disjunktiven Normalform zu entfernen und auszuglätten.

Die oben möglichen Verbesserungen springen auch nicht ins Auge, was vielleicht darauf hindeutet, dass das Optimierungsproblem nicht völlig trivial ist. Trotzdem gibt es Umwandlungsverfahren, mit denen man solche Optimierungen quasi automatisch finden kann (mit einem bisschen Feinschliff am Ende).

Der Trick dabei besteht eigentlich in einem Schritt in die falsche Richtung. Man bläst den zu optimierenden Ausdruck zunächst auf, macht ihn größer und umfangreicher, aber dafür in seiner Struktur auch vollständiger. Die vollständigere Form bietet mehr Angriffsflächen für Kürzungsmöglichkeiten und macht sonst versteckte Möglichkeiten leichter erkennbar — am Ende erzielt man dadurch einen Nettogewinn.

Sowohl für das Aufblasen wie auch für das spätere wieder Herunterblasen wird das gleiche Mittel benutzt: das Absorptionsgesetz der booleschen Algebra.

Wir beginnen mit der Erläuterung der Methode, wie üblich durch die Einführung einiger technischer Bezeichnungen.

Definition 2.14 Eine disjunkte Normalform A für eine boolesche Funktion von n Variablen heißt **kanonisch** (Abkürzung: KDN), wenn in jedem Konjunktionsterm (also Produktterm) von A *alle* n Variablen vorkommen, entweder in unnegierter oder in negierter Form.

Die Konjunktionsterme einer kanonischen disjunktiven Normalform werden auch **Minterme** genannt.

Beispiel 2.15 Die oben betrachteten Normalformen

$$\bar{a}bc + ac + a\bar{b} \quad \text{und} \quad a\bar{b}c + \bar{a}c$$

sind nicht kanonisch.

Die Normalformen

$$\bar{a}bc + abc + \bar{a}\bar{b}c + a\bar{b}\bar{c} \quad \text{und} \quad a\bar{b}c + \bar{a}bc + \bar{a}\bar{b}c$$

sind kanonisch (nebenbei sind sie äquivalent zu den zuerst angegebenen Formen).

Bemerkung 2.16 Sei f eine boolesche Funktion in n Variablen. Der boolesche Ausdruck für f , den wir nach der auf Seiten 59 und 60 beschriebenen Methode aus der Wertetabelle von f erhalten, ist immer kanonisch.

Rechenverfahren 2.17 Mit einem einfachen Rechenverfahren kann man jeden booleschen Ausdruck in disjunktiver Normalform kanonisch machen (dabei wird der Ausdruck aber länger).

Schritt 1.

Wenn in einem Konjunktionsterm T eine Variable a nicht vorhanden ist, ersetze T mittels des Absorptionsgesetzes durch

$$aT + \bar{a}T.$$

Weil a vorher in T nicht vertreten war, sind beide Summanden, die hier entstehen, wieder Konjunktionsterme.

Man wiederhole diesen Schritt so oft er noch möglich ist, d. h., bis alle Konjunktionsterme alle Variablen enthalten.

Schritt 2.

Bei Schritt 1 kann mehrmals der gleiche neue Konjunktionsterm entstehen. Man fasse gleiche Summanden zu einem Exemplar zusammen (das darf man machen wegen der Regel $a + a = a$).

Beispiel 2.18 Wir betrachten wieder die disjunktive Normalformen

$$\bar{a}bc + ac + a\bar{b} \quad \text{und} \quad a\bar{b}c + \bar{a}c.$$

a) Aus dem Term ac in

$$\bar{a}bc + ac + a\bar{b}$$

wird nach Schritt 1 die Summe $abc + \bar{a}bc$ und aus dem Term $a\bar{b}$ wird $a\bar{b}c + a\bar{b}\bar{c}$. Der Term $\bar{a}bc$ ist schon kanonisch.

Insgesamt erhalten wir nach Schritt 1 den Ausdruck

$$\bar{a}bc + abc + a\bar{b}c + a\bar{b}\bar{c} + a\bar{b}\bar{c}$$

Schritt 2 vereinfacht dies zu

$$\bar{a}bc + abc + a\bar{b}c + a\bar{b}\bar{c},$$

die vorhin angegebene kanonische Variante von dieser Normalform.

b) Aus dem Term $\bar{a}c$ in

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

wird nach Schritt 1 die Summe $\bar{a}bc + \bar{a}\bar{b}c$ und insgesamt erhalten wir nach diesem Schritt

$$a\bar{b}c + \bar{a}bc + \bar{a}\bar{b}c. \quad (2.7)$$

Schritt 2 muss nicht ausgeführt werden.

An der endgültigen Form (2.7) in Beispiel 2.18 b) ist nicht mehr zu erkennen, welcher Term genau aufgeblasen wurde, um diese Gestalt zu erreichen. Eine zweite Möglichkeit, die kanonische Form wieder „abzublasen“, sollte sofort auffallen.

Der Clou in unserem Optimierungsverfahren ist, dass wir nicht nur die tatsächlich ausgeführten Aufblasungsschritte wieder rückgängig machen können, sondern *gleichzeitig* auch die neuen Kürzungsmöglichkeiten wahrnehmen dürfen, die erst durch das Aufblasen entstanden sind.

Der Grund ist, dass ein (durch die Umkehr des Aufblasens) weggekürzter Term dadurch nicht verbraucht wird, denn wieder nach der Regel $a + a = a$ dürfen wir jeden Term so oft vermehren, dass wir genügend viele Kopien haben für alle Kürzungsschritte, die wir durchführen wollen!

Für die effizienteste Beschreibung des jetzt anschließenden Kürzungsverfahrens führen wir eine neue Notation ein, die uns dabei helfen wird, keine Kürzungsmöglichkeiten zu übersehen.

Definition 2.19 Sei A ein boolescher Ausdruck in kanonischer disjunktiver Normalform. Wir wählen für die Variablennamen in A eine feste alphabetische Reihenfolge und sortieren in jedem Minterm die Variablen nach dieser Reihenfolge.

Mit dieser einmal gewählten festen Form können wir jeden Minterm eindeutig beschreiben durch das in ihm stehende Muster von unnegierten und negierten Variablen.

Dieses Muster können wir durch ein eindeutige *Bitfolge* beschreiben, wenn wir für jede Variable in der Reihenfolge ihres Erscheinens in den Mintermen ein 1-Bit schreiben, wenn die Variable unnegiert erscheint, und ein 0 Bit, wenn sie negiert ist.

Diese Bitfolge gibt gleichzeitig das eindeutige Belegungsmuster der Variablen mit Werten an, bei dem dieser Minterm den Wert 1 annimmt!

Wir nennen diese Bitmuster die ***Binäräquivalenten*** der Minterme, und benutzen sie als informative Bezeichnungen für die Minterme (informativ, weil man aus dem Namen sofort den Minterm ablesen kann!). Manchmal benennt man Minterme auch in der Form m_k , wo k der *dezimale* Wert des Binäräquivalenten ist.

Zum Beispiel, bezüglich der normalen alphabetischen Reihenfolge der Buchstaben hat der Minterm $a\bar{b}c$ den Binäräquivalenten 101 und kann auch als m_5 bezeichnet werden. Hingegen wäre m_6 der Minterm $ab\bar{c}$ (mit Binäräquivalenten 110) und m_7 wäre der Minterm abc , dessen Binäräquivalent 111 ist.

Die Bitmuster der Binäräquivalenten sind sehr hilfreich bei der Organisation eines Verfahrens, das die Minterme einer kanonischen disjunktiven Minimalform zu so genannten ***Primtermen*** kürzt. Primterme sind Konjunktionsterme, aus denen *überflüssige* Variablen entfernt worden sind (Variablen, deren Belegung in Kombination mit der im Konjunktionsterm gegebenen Belegung der anderen Variablen die gegebene booleschen Funktion nicht beeinflusst), aber die in diesem Sinne nicht *weiter* kürzbar sind — daher der Name *prim*.

Rechenverfahren 2.20 (Quine-McCluskey, erster Teil) Sei A ein boolescher Ausdruck in n Variablen in kanonischer disjunktiver Normalform.

Folgendes Rechenverfahren benutzt das Absorptionsgesetz, um Terme aus A unter Kürzung einer Variablen zu kombinieren und vereinfachen, und findet eine maximale Anzahl solcher Kürzungen (aber noch nicht eine *minimale* disjunktive Normalform für die gegebene boolesche Funktion; zu diesem Zweck wird ein zweiter Schritt anschließen).

Schritt 1.

Schreibe eine Spalte mit den Binäräquivalenten aller Minterme von A ; jedes Binäräquivalent ist ein n -stelliges Bitmuster.

Schritt 2.

Sortiere die Spalte in aufsteigender Reihenfolge der Anzahl der Einsbits in den Binäräquivalenten, und gruppieren die Binäräquivalenten nach diesem Kriterium.

Schritt 3.

Beginnend mit der Gruppe mit der niedrigsten Anzahl von Einsen, vergleiche jedes Bitmuster mit den Bitmustern der nächsthöheren Gruppe auf der Suche nach Paaren von Bitmustern, die sich nur in einer Ziffer unterscheiden, wobei das Bitmuster aus der niederen Gruppe an dieser Stelle eine 0 hat, und das gepaarte Bitmuster aus der höheren Gruppe an der gleichen Stelle eine 1.

Wann immer ein solches Paar gefunden wird, markiere man die beiden Binäräquivalenten, die gepaart wurden, mit einem Haken, und man kopiere ihr Bitmuster in die nächste Spalte, aber mit einem Strich statt einer Ziffer an der nichtübereinstimmenden Bitstelle.

Die abgehakten Binäräquivalenten dürfen trotzdem an weiteren Vergleichen beteiligt werden!

Schritt 4.

Wenn alle Gruppen verarbeitet wurden und alle Vergleiche durchgeführt wurden, wiederhole man dieses Verfahren ab Schritt 2 mit der nächsten Spalte.

Wenn aber die nächste Spalte leer ist, dann ist man fertig. In diesem Fall übersetze man die *nicht* abgehakten Bitmuster aus allen Spalten in Konjunktionsterme nach der Regel: für ein 1 Bit schreibe man die Variable an dieser Stelle unnegiert, für ein 0 Bit schreibe man sie negiert, und für einen Strich schreibe man sie gar nicht.

Jeder so erhaltene Konjunktionsterm ist ein Primterm, und die Summe dieser Primterme ist ein boolescher Ausdruck B , der zu A äquivalent ist.

Was passiert hier wirklich, hinter den Kulissen? Jedes Mal, das wir ein Paar von Bitmustern abhaken und dafür ein neues Bitmuster schreiben, haben wir, bezogen auf das Endergebnis, aus A einen Ausdruck der Gestalt

$$Ka + K\bar{a} \quad (\text{wo } K \text{ ein Konjunktionsterm ist})$$

durch den kürzeren Ausdruck K ersetzt; aus dem ursprünglichen Ausdruck wird dadurch etwas äquivalentes, denn dies ist eine Anwendung der Absorptionsregel:

$$Ka + K\bar{a} = K(a + \bar{a}) = K1 = K.$$

Falls einer der abgehakten Bitmuster auch an weiteren Paaren beteiligt ist, denken wir uns, dass wir ihn *vor* dem obigen Kürzen durch eine Anwendung der Idempotenzregel $a + a = a$ verdoppelt haben, damit er *nach* dem Kürzen für die Beteiligung an einem weiteren Kürzvorgang noch vorhanden ist.

Da hier nur gültige Äquivalenzregeln der booleschen Algebra angewendet wurden, ist das Endergebnis B tatsächlich äquivalent zum Ausgangsausdruck A .

Wir illustrieren die oben beschriebene Methode anhand einiger Beispiele.

Beispiele 2.21 a) Wir betrachten die kanonische disjunktive Normalform

$$\bar{a}\bar{b}c + \bar{a}bc + a\bar{b}c + abc + a\bar{b}\bar{c}. \quad (2.8)$$

Die Binäräquivalenten der Minterme sind 001, 011, 101, 111 und 100. Wir sortieren Sie nach der Anzahl der Einsen wie folgt:

Gruppe	Binäräquivalent	verbraucht?
1	001	
	100	
2	011	
	101	
3	111	

Wir vergleichen zunächst die Muster mit einem Einsbit mit den Mustern in der nächsten Gruppe. Wenn ein Paar sich nur an einer Stelle unterscheidet, muss das Paarelement aus der niederen Gruppe an dieser Stelle eine 0 und das Element aus der höheren Gruppe an dieser Stelle eine 1 haben, da sonst die Anzahl der Einsen beim zweiten Element nicht *größer* wäre.

Das heißt, zu 001 in Gruppe 1 suchen wir 101 oder 011 in Gruppe 2. Beide sind vorhanden und wir haken deshalb 001, 101 und 011 ab und schreiben für das Paar 001 mit 101 die Kürzung -01 in die zweite Spalte, für das Paar 001 mit 011 schreiben wir 0-1 in die zweite Spalte, beide in Gruppe 1:

Gruppe	BÄ	?	BÄ	?
1	001	✓	-01	
	100		0-1	
2	011	✓		
	101	✓		
3	111			

Wir suchen in der zweiten Gruppe auch einen Partner für 100 aus Gruppe 1. Mögliche Partner wären 110 und 101, aber nur 101 ist vorhanden (und schon abgehakt). Wir können deshalb 100 auch abhaken und für das gefundene Paar 10- in die zweite Spalte schreiben.

Gruppe	BÄ	?	BÄ	?
1	001	✓	-01	
	100	✓	0-1	
			10-	
2	011	✓		
	101	✓		
3	111			

Jetzt vergleichen wir die Elemente von Gruppe 2 mit den Elementen der dritten Gruppe. Für beide Muster aus Gruppe 2 ist 111 der einzig mögliche Partner mit drei Einsen, und dieses Muster ist tatsächlich vorhanden. Deshalb können wir 111 abhaken und für die Kombination mit 011 können wir -11 in die zweite Spalte schreiben, für die Kombination mit 101 können wir 1-1 schreiben.

Gruppe	BÄ	?	BÄ	?
1	001	✓	-01	
	100	✓	0-1	
			10-	
2	011	✓	-11	
	101	✓	1-1	
3	111	✓		

Jetzt sind keine Vergleiche in Spalte 1 mehr möglich, und wir verarbeiten Spalte 2. Das Element -01 aus Gruppe 1 kann nur mit -11 gepaart werden, und dieses Element existiert tatsächlich in Gruppe 2. Also haken wir beide ab und schreiben --1 in Spalte 3.

Gruppe	BÄ	?	BÄ	?	BÄ	?
1	001	✓	-01	✓	--1	
	100	✓	0-1			
			10-			
2	011	✓	-11	✓		
	101	✓	1-1			
3	111	✓				

Das Element 0-1 aus Gruppe 1 kann nur mit 1-1 gepaart werden, und auch dieses Element existiert in Gruppe 2. Wieder haken wir beide Elemente des Paares ab und müssten --1 in Spalte 3 schreiben. Dieses Element ist aber schon vertreten und deshalb fügen wir doch nichts der dritten Spalte zu.

Gruppe	BÄ	?	BÄ	?	BÄ	?
1	001	✓	-01	✓	--1	
	100	✓	0-1	✓		
			10-			
2	011	✓	-11	✓		
	101	✓	1-1	✓		
3	111	✓				

Das letzte Element aus Gruppe 1 in Spalte 2 ist 10- und könnte nur mit 11- in der nächsten Gruppe gepaart werden. Dieses Element ist aber *nicht* vorhanden, so dass wir 10- nicht abhaken dürfen und auch nichts in die dritte Spalte schreiben dürfen.

Weitere Vergleiche aus Spalte 2 sind nicht möglich, und in Spalte 3 sind auch keine Vergleiche möglich, weil die Spalte nur ein Element enthält.

Wir sind also mit dem Kürzen fertig. In der gesamten Tabelle tragen nur die Elemente 10- in Spalte 2 und --1 in Spalte 3 kein Häkchen. Sie entsprechen den Konjunktionstermen $a\bar{b}$ und c , so dass wir den ursprünglichen Ausdruck jetzt zu der disjunktiven Normalform

$$a\bar{b} + c \quad (2.9)$$

gekürzt haben.

Weitere Kürzungen, auch mit anderen Verfahren, sind nicht möglich, weil die Variablen in den beiden Konjunktionstermen unabhängig voneinander sind und deshalb wesentlich zu den Werten des Ausdrucks beitragen.

b) Wir betrachten die kanonische disjunktive Normalform

$$\bar{a}bc + abc + \bar{a}\bar{b}c + a\bar{b}\bar{c} \quad (2.10)$$

aus Beispiel 2.18 a).

Die Binäräquivalenten der Minterme sind 011, 111, 101 und 100. Wir sortieren Sie nach der Anzahl der Einsen wie folgt:

Gruppe	Binäräquivalent	verbraucht?
1	100	
2	101	
	011	
3	111	

Wir vergleichen zunächst das einzige Muster mit nur einem Einsbit, nämlich 100, mit den Mustern in der nächsten Gruppe. Mögliche Partner in Gruppe 2 wären 110 und 101. Nur 101 ist in Gruppe 2 vorhanden und wir haken deshalb 100 und 101 ab und schreiben 10- in die zweite Spalte, natürlich in Gruppe 1:

Gruppe	BÄ	?	BÄ	?
1	100	✓	10-	
2	101	✓		
	011			
3	111			

Beide Elemente von Gruppe 2 haben 111 als Partner in der dritten Gruppe. Deshalb können wir jetzt doch 011 und natürlich 111 abhaken und für die Kombinationen mit den Elementen von Gruppe 2 können wir 1-1 und -11 in die zweite Spalte schreiben, beide in Gruppe 2.

Gruppe	BÄ	?	BÄ	?
1	100	✓	10-	
2	101	✓	1-1	
	011	✓	-11	
3	111	✓		

Jetzt sind keine Vergleiche in Spalte 1 mehr möglich, und wir betrachten Spalte 2. Das Element 10- aus Gruppe 1 könnte nur mit 11- gepaart werden, aber Gruppe 2 enthält überhaupt kein Element, das mit einem

Strich endet. Deshalb ist nichts abzuhaken und nichts in Spalte 3 zu schreiben.

Weitere Vergleiche aus Spalte 2 sind auch nicht möglich. Wir sind also mit dem Kürzen fertig. Alle Elemente aus der ersten Spalte sind abgehakt, aber keines der Elemente aus Spalte 2.

Wir erhalten aus den unabgehakten Elementen diesmal die disjunktive Normalform

$$a\bar{b} + ac + bc. \quad (2.11)$$

Leider ist dies aber *nicht* eine optimale Vereinfachung von (2.10), denn (2.11) ist äquivalent zu

$$a\bar{b} + bc \quad (2.12)$$

und der Summand ac ist überflüssig.

Das kann man schnell einsehen, denn mit der Absorptionsregel kann man den ersten plus den zweiten Summanden von (2.10) zu bc kürzen und den dritten plus den vierten zu $a\bar{b}$. Die Summe ergibt die optimale Form (2.12) (und viel schneller als das Verfahren von Quine-McCluskey).

Das letzte Beispiel entwertet das Verfahren von Quine und McCluskey nicht wirklich, denn hier lag eine besonders ungünstige KDN vor. Die Summanden ließen sich in Paaren direkt mit dem Absorptionsgesetz wegekürzen, während das Verfahren von Quine-McCluskey noch andere Kürzungsmöglichkeiten fand, die aber leider zu weiteren nicht mit den anderen vergleichbaren Kurzformen führten. Am Ende blieben überflüssige Terme über.

Beispiel 2.21 a) zeigt aber, wie leistungsfähig dieses Kürzungsverfahren sein kann.

Trotzdem können wir das Ergebnis von Beispiel 2.21 b) nicht einfach so stehen lassen. Wir müssen also jede Berechnung mit dem Verfahren von Quine-McCluskey im gleichen Sinne hinterfragen und folgende weitere Bereinigung anschließen lassen:

Rechenverfahren 2.22 (Quine-McCluskey, zweiter Teil) Sei B eine disjunktive Normalform, erhalten als das Ergebnis einer Berechnung mit dem ersten Teil des Quine-McCluskey Verfahrens.

Um eine optimale DN für die entsprechende boolesche Funktion zu erhalten, müssen wir untersuchen, ob einige der Primterme in B überflüssig sind.

Ein Primterm P , wie jeder boolesche Ausdruck, hat eine eindeutige Darstellung als eine Summe von Mintermen. Wir sagen, dass P die Minterme

in dieser Summe **überdeckt**. Logisch gesehen ist P das Oder der von ihm überdeckten Minterme.

Jeder Minterm K wird bei genau einer Belegung der Variablen 1, und die von P überdeckten Minterme sind genau diejenigen, bei deren Belegungen $P = 1$ wird. Da jeder Konjunktionsterm ein Produkt (also ein Und) von Bedingungen für Variablen ist, und da bei einem Minterm jede Variable vorkommen muss, überdeckt ein Primterm P einen Minterm K genau dann, wenn K alle Bedingungen aus P stellt *und eventuell noch zusätzliche*. In anderen Worten, P überdeckt K genau dann, wenn P in der Produktschreibweise so aussieht, wie ein „Teiler“ von K , also wenn P aus K hervorgeht durch Streichung einiger Faktoren.

Diesen Umstand kann man mit den Binäräquivalenten sehr schön beschreiben. Das Binäräquivalent eines Minterms ist eine Bitfolge; für einen Primterm können wir auch ein Binäräquivalent schreiben, wenn wir die vorhandenen Variablen auf die übliche Weise durch eine 0 oder eine 1 darstellen und die fehlenden Variablen durch einen Strich kennzeichnen. Diese Folge von Nullen, Einsen und Strichen kann man als ein Bitfolgenmuster verstehen, wobei an den Stellen der Striche beliebige Werte 0 oder 1 erlaubt sind. Ein Primterm P überdeckt einen Minterm K genau dann, wenn die Binäräquivalentbitfolge von K zum Bitfolgenmuster des Binäräquivalenten von P passt.

Sei B ein boolescher Ausdruck, der eine Summe von Primtermen ist. Die kanonische disjunktive Normalform von B ist die Summe der Minterme, die von diesen Primtermen überdeckt werden (da jeder Primterm die Summe der Minterme ist, die er selber überdeckt).

Wenn nun ein Primterm P nur Minterme überdeckt, die auch von anderen Primtermen von B überdeckt werden, dann kann man P aus B entfernen, ohne dass die Menge der überdeckten Minterme sich ändert, also ohne dass die vom Ausdruck beschriebene boolesche Funktion sich ändert.

Solche möglichen Kürzungen entdeckt man mit einer Tabelle mit einer Spalte für jeden Minterm (oder sein Binäräquivalent) und einer Zeile für jeden Primterm in B (oder sein Binäräquivalent). Für jeden Primterm P aus B und jedem Minterm K , den er überdeckt, markieren wir die Zelle am Schnittpunkt der Zeile von P mit der Spalte von K mit einem Kreuz.

In der Sprache der Binäräquivalenten sieht das so aus: die Zeilen sind Primtermbitmustern zugeordnet und die Spalten sind Mintermbitfolgen zugeordnet. Wir setzen Kreuze in allen Zellen, deren Spaltenbitfolge dem Zeilenbitmuster entspricht.

Die kanonische disjunktive Normalform des Ausdrucks B ist die Summe der Minterme, deren Spalten in dieser Tabelle mindestens ein Kreuz enthalten.

Wenn jedes Kreuz in der Zeile eines Primterms in einer Spalte steht, in der es auch andere Kreuze gibt, dann kann man diesen Primterm ganz aus B entfernen, ohne dass die KDN (oder die von ihr beschriebene boolesche Funktion) sich ändert. Solche Primterme sind in B überflüssig und können schadlos aus B gekürzt werden.

Hingegen die Primterme, die zu Kreuzen gehören, die alleine in einer Spalte stehen, sind wesentlich und dürfen nicht aus B gekürzt werden (sie zu entfernen würde die KDN und somit die boolesche Funktion des Ausdrucks verändern).

Mit den Tabellen findet man sehr schnell alle überflüssigen Primterme in B , wobei man nach jeder Kürzung die Tabellenreihe des gekürzten Primterms streichen muss und die Tabelle neu evaluieren muss. Jede Kürzung kann aus diesem Grund andere Kürzungen blockieren, die am Anfang noch möglich gewesen wären.

Man muss deshalb alle möglichen Reihenfolgen von solchen Kürzungen untersuchen, um diejenige zu finden, bei denen die wenigsten Blockaden auftreten und bei denen die Kürzungen am längsten fortgesetzt werden können. Nur eine solche Untersuchung erlaubt es, die maximal mögliche Anzahl von Kürzungen zu finden und die kürzeste Primtermssumme für eine boolesche Funktion zu finden.

Wir illustrieren dieses Verfahren an einem Beispiel, und zwar anhand der Ergebnisse aus Beispiel 2.21.

Beispiele 2.23 a) In Beispiel 2.21 a) erhielten wir als Ergebnis des ersten Teils des Quine-McCluskey Verfahrens die disjunktive Normalform

$$a\bar{b} + c.$$

Wir haben drei Variablen und betrachten folgende Tabelle von Primtermen (mit ihren Binäräquivalenten) und Binäräquivalenten von Mintermen.

	000	001	010	011	100	101	110	111
$a\bar{b}$ (BÄ 10-)					×	×		
c (BÄ --1)		×		×		×		×

Wir sehen sofort, dass jede Reihe Spaltenmarkierungen hat, die nur in dieser Reihe vorkommen. Deshalb ist keiner der beiden Summanden überflüssig und keine Kürzung möglich.

- b) In Beispiel 2.21 b) erhielten wir als Ergebnis des ersten Teils des Quine-McCluskey Verfahrens die disjunktive Normalform

$$a\bar{b} + ac + bc.$$

Wir haben drei Variablen und betrachten jetzt folgende Tabelle von Primtermen (mit ihren Binäräquivalenten) und Binäräquivalenten von Mintermen.

	000	001	010	011	100	101	110	111
$a\bar{b}$ (BÄ 10-)					×	×		
ac (BÄ 1-1)						×		×
bc (BÄ -11)				×				×

Die Reihen $a\bar{b}$ und bc haben Spaltenmarkierungen, die nur in der betreffenden Reihe vorkommen. Deshalb können diese Summanden nicht weggekürzt werden.

Aber beide Markierung in der Reihe für den Primterm ac stehen in Spalten, in denen es auch Markierungen von anderen Primtermen gibt. Das bedeutet, dass die Minterme, die ac überdeckt, auch nach Entfernung dieses Summanden von anderen Primtermen überdeckt sein werden, d. h., ac kann gestrichen werden, ohne die boolesche Funktion zu verändern.

Schon vor der Entfernung von ac konnten wir sehen, dass die anderen Primterme nicht aus dem Ausdruck entfernt werden dürfen; für den kürzeren Ausdruck gilt das erst recht. Weitere und andere Kürzungen sind also nicht möglich.

Deshalb ist

$$a\bar{b} + bc$$

nun eine *optimale* disjunkte Normalform für (2.10), wie wir in der Diskussion auf Seite 71 schon behauptet haben.

Unsere errungenen Kenntnisse über den Entwurf von digitalen Schaltungen, um bestimmte boolesche Funktionen zu berechnen, wollen wir jetzt einsetzen, um einige der unverzichtbaren Bausteine zu entwerfen, die in jedem modernen Computer ihren Dienst tun und ohne die er nicht funktionieren könnte.

Man unterscheidet verschiedene Arten von Bausteinen. **Kombinatorische Schaltkreise** haben Eingänge und Ausgänge und berechnen intern gewisse logische Funktionen der Eingangssignale und machen die Ergebnisse an ihren Ausgängen verfügbar. Die Arithmetic and Logic Unit, die ALU, die die eigentliche Recheneinheit des Computers ist, erfüllt zum Beispiel solche Aufgaben.

Aber es reicht nicht, nur die gegenwärtig anliegenden Zustände zu verarbeiten. Manchmal müssen Ergebnisse über längere Zeit aufbewahrt werden und später in Berechnungen wieder einfließen. Kein Rechner kann funktionieren ohne eine Möglichkeit, Daten zu Speichern, sei es nur, um sie für eine Berechnung parat zu halten oder weil sie über längere Zeit die Arbeitsweise des Rechners steuern sollen.

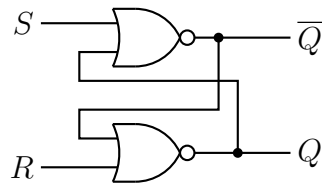
Deshalb braucht jeder Computer neben kombinatorischen Schaltkreisen auch Schaltkreise, die in einen bestimmten Zustand gebracht werden können und diesen Zustand beibehalten, bis er mit Absicht und kontrolliert verändert wird. Solche Bausteine heißen **Speicher**.

Obwohl in einem Rechner viele Daten gespeichert werden müssen, reicht es für den Anfang, einen **1-Bit Speicher** zu konstruieren, denn diese kann man später miteinander verkoppeln, um größere, sogar sehr große Speicher zusammenzusetzen.

Wie kann man also einen Bit speichern? Wir brauchen einen Baustein, der zwei stabile Zustände annehmen kann (wir nennen sie 0 und 1, ohne uns zunächst darum zu kümmern, wie diese Zustände wirklich realisiert werden) und der von außen gesteuert werden kann, um eines von drei Dingen zu tun: den Zustand auf 0 zu setzen, den Zustand auf 1 zu setzen oder als dritte Möglichkeit, weder das eine noch das andere zu tun, sondern den Zustand so zu belassen, wie er als letztes eingestellt wurde.

Folgende Schaltung (Abbildung 2.16 auf der nächsten Seite), eine so genannte **SR Flipflop-Schaltung**, kann dieses leisten. Man beachte, dass diese Schaltung sich von den bisher von uns betrachteten unterscheidet, weil sie eine **Rückkopplung** der Ausgänge mit den Eingängen beinhaltet. Schaltungen dieser Art berechnen keine booleschen Funktionen, weil ihr Verhalten nicht nur von den gegenwärtigen Eingängen abhängt, sondern auch von der Vorgeschichte der Schaltung (das ist ja auch gerade der Sinn eines Speichers!). Deshalb kann man sie auch nicht mit den oben ausgearbeiteten Verfahren optimieren.

Wie verhält sich dieses Flipflop? Beachten Sie, dass diese Schaltung zwei Eingänge und auch zwei Ausgänge hat. Die Eingänge heißen S und R, was auf ihre Funktion hinweisen soll aber auch der Schaltung ihren Namen gibt. Eingang S wird auf 1 gebracht, um das gespeicherte Bit zu **Setzen** (englisch **Set**), und Eingang R wird auf 1 gebracht, um das gespeicherte Bit **Rückzusetzen**

Abbildung 2.16: Ein *SR*-Flipflop

(englisch **R**eset), also auf den Zustand 0 zu bringen. Die beiden Ausgänge Q und \overline{Q} sind nicht unabhängig, sondern bei normalem Funktionieren immer entgegengesetzt zueinander, was natürlich nicht aus dieser Namensgebung folgt, aber sie rechtfertigt. Die genaue Funktionsweise untersuchen wir jetzt.

Wir erinnern zunächst kurz an die Funktionsweise der NOR-Gatter in der Schaltung. Hier noch einmal die Wertetabelle der NOR Funktion: Wenn ein

e_1	e_2	a
0	0	1
0	1	0
1	0	0
1	1	0

Tabelle 2.7: Wertetabelle der NOR Funktion

Eingang auf 0 steht, wird der andere Eingang am Ausgang invertiert. Wenn ein Eingang auf 1 steht, ist der Ausgang immer 0.

In der SR-Schaltung wird der Ausgang jedes NOR Gatters auf einen Eingang des anderen Gatters gelegt; der andere Eingang ist die S oder die R Leitung. Daraus und aus der obigen NOR Tabelle folgt folgendes Verhalten der SR-Ausgänge als Funktion von den Eingängen und ihrem bisherigen Zustand \overline{Q}_0 und Q_0 :

S	R	\overline{Q}	Q
0	0	\overline{Q}_0	Q_0
0	1	1	0
1	0	0	1
1	1	0	0

Tabelle 2.8: Wertetabelle des SR-Flipflops

Das müssen wir ein bisschen erklären. Wenn beide Eingänge S und R Null sind, dann invertieren beide Gatter ihren zweiten Eingang, der gleich dem Ausgang des anderen Gatters ist. Das heißt, \overline{Q} nimmt den negierten Wert

des bisherigen Zustandes von Q an, also den Wert $\overline{Q_0}$, und Q nimmt den negierten Wert des bisherigen Zustandes von \overline{Q} an, also den Wert $\overline{\overline{Q_0}}$.

So lange die bisherigen Werte von Q und \overline{Q} komplementär zueinander waren, werden diese Werte beibehalten! Das heißt, einmal gesetzte komplementäre Werte am Ausgang sind stabil und werden durch die Rückkopplung bewahrt, solange beide Eingänge 0 sind.

Wenn $S = 1$ und $R = 0$, wird unabhängig vom bisherigen Wert der Ausgänge \overline{Q} zwangsweise auf 0 gesetzt. Weil $R = 0$, zeigt der Ausgang Q für kurze Zeit die Negation des bisherigen Wertes von \overline{Q} , aber nachdem $\overline{Q} = 0$ geworden ist dann die Negation *dieses* Wertes, also 1. Die Signalkombination $S = 1$ und $R = 0$ setzt also nach kurzer Zeit $Q = 1$ und $\overline{Q} = 0$, und das sind komplementäre Werte.

Wenn $S = 0$ und $R = 1$, wird unabhängig vom bisherigen Wert der Ausgänge Q zwangsweise auf 0 gesetzt. Weil $S = 0$, zeigt der Ausgang \overline{Q} für kurze Zeit die Negation des bisherigen Wertes von Q , aber nachdem $Q = 0$ geworden ist dann die Negation *dieses* Wertes, also 1. Die Signalkombination $S = 0$ und $R = 1$ setzt also nach kurzer Zeit $Q = 0$ und $\overline{Q} = 1$, und das sind wieder komplementäre Werte.

Wenn wir Q als den „eigentlichen“ Ausgang der Schaltung auffassen, so hat die Eingangskombination $S = 1$, $R = 0$ tatsächlich die Wirkung, Q auf 1 zu setzen, und $S = 0$, $R = 1$ hat tatsächlich die Wirkung, Q auf 0 zurückzusetzen. Der andere Ausgang \overline{Q} ist nur ein „Anhängsel“ von Q und gibt immer seinen komplementären Wert wieder. Und die Eingangskombination $S = R = 0$ bewahrt tatsächlich das komplementäre Paar Q , \overline{Q} wie es vorher war, d. h., wie es zuletzt gesetzt oder rückgesetzt wurde.

Nur die letzte Zeile in der Tabelle haben wir noch nicht besprochen. Wenn S und R beide 1 sind, dann werden beide Ausgänge auf 0 gezwungen. Das ist zunächst nicht weiter schlimm, abgesehen davon, dass Q und \overline{Q} nicht mehr komplementär sind. Aber es löscht die gespeicherten Daten und jede Erinnerung an sie, und macht den Speicher funktionsunfähig, bis neue Daten 0 oder 1 explizit hineingeschrieben werden.

Im Normalzustand hat dieses Flipflop komplementäre Ausgänge und die meiste Zeit, bis auf Schreibvorgänge, werden beide Eingänge effektiv auf 0 gehalten, um die gespeicherten Daten nicht zu verändern. Sobald aber beide Eingänge auf 1 gesetzt wurden, kann der gespeicherte Zustand nicht mehr stabil gemacht werden durch Setzen beider Eingänge auf 0. Dass bei $S = R = 0$ überhaupt ein bestimmter Zustand angenommen wurde hing von der Annahme ab, dass Q und \overline{Q} vorher komplementäre Werte hatten. Waren diese Werte aber *beide* vorher 0, wie das bei $S = R = 1$ eingerichtet wird, dann versuchen bei einem späteren Zustand von $S = R = 0$ zunächst beide

Ausgänge auf 1 zu gehen. Sobald ein Ausgang aber tatsächlich 1 wird, zwingt er durch die Rückkopplung den anderen Ausgang auf 0 (weil ein NOR mit einem Eingang 1 immer den Ausgang 0 hat).

Das heißt, wenn beide Eingänge auf 0 gehen, dann werden wieder komplementäre Ausgänge eingerichtet, aber nicht auf eindeutig festgelegte Weise. Es ist zufällig und auf jeden Fall nicht vorhersehbar, ob nun Q oder \overline{Q} auf 1 geht, und das Ergebnis kann nur dann wirklich bestimmt werden, wenn S und R nicht *gleichzeitig* auf 0 gehen. Ist die Rücksetzung eines der Eingänge verzögert, liegt dann für kurze Zeit ein Setz- oder Rücksetzzustand vor und er bestimmt, wie die Ausgänge belegt werden. Aber diese Belegung kann nicht als sinnvoll und verwertbar angesehen werden.

Aus dem genannten Grund ist die Eingangskombination $R = S = 1$ verboten. Man kann die Schaltung auch so verfeinern, dass diese Kombination nie auftreten kann.

In der Praxis wird auch eine andere Art von Verfeinerung benötigt. Daten zu einem bestimmten Zweck liegen nie sofort bereit in einem Computer, weil sie in der Regel aus einem vorherigen Verarbeitungsschritt oder einem Speicherzugriff stammen, der eine gewisse Zeitspanne beansprucht, bis die Daten „durchgelaufen“ sind und stabil auf den Leitungen vorliegen. Deshalb lässt man Daten auch nicht unkontrolliert in eine Schaltung einfließen, sondern regelt den zeitlichen Ablauf des Datenflusses mit einer Uhr, damit die richtigen Daten zusammentreffen und in der richtigen Reihenfolge auf die vorgesehene Weise verarbeitet werden. Zu diesem Zweck hat jeder Computer eine Uhr, die als Taktgeber verwendet wird.

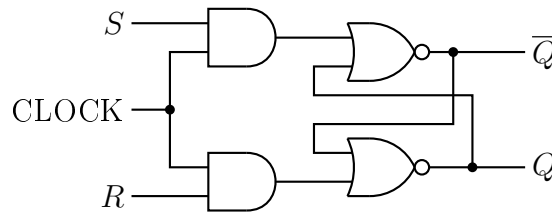
Die zeitliche Regelung bedeutet auch, dass Daten nur zu gewissen Zeiten im Taktzyklus in ein Speicher weitergeleitet werden. Wir wissen ja schon, wie verhindert werden kann, dass Daten im Speicher gesetzt werden. Dazu muss man nur beide Eingänge des Flipflops auf 0 setzen, und das kann man sehr einfach mit Hilfe eines Uhrensignals regeln.

Folgende Schaltung (Abbildung 2.17 auf der nächsten Seite) funktioniert wie das Flipflop aus Abbildung 2.16, aber es kann das Flipflop nur setzen oder rücksetzen, wenn das Signal CLOCK den Wert 1 hat; in diesem Fall werden die Signale S und R durch die AND Gatter an die NOR Gatter der ursprünglichen SR -Flipflopschaltung weitergeleitet.

Wenn aber $CLOCK = 0$, dann sind die Ausgänge beider ANDs Null und deshalb werden beide Eingänge des ursprünglichen Flipflops Null; er bewahrt seinen bisherigen Zustand.

Nichts an der Schaltung verlangt, dass das Signal CLOCK wirklich von der Uhr stammt, d. h., man kann jedes andere Steuersignal zur Freigabe des Flipflops verwenden, wenn man es auf die CLOCK Leitung legt.

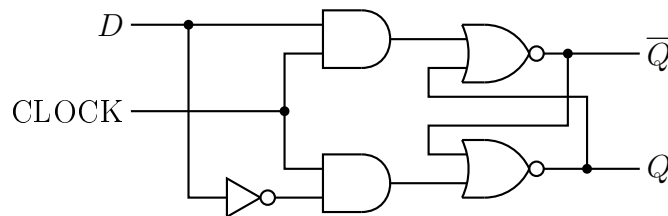
In der ursprünglichen Schaltung mussten S und R auf 0 gesetzt werden,

Abbildung 2.17: Ein gesteuertes SR -Flipflop

damit die Schaltung ihren Zustand aufbewahrt; in der erweiterten Schaltung kann CLOCK diese Funktion übernehmen, so dass die Kombination $S = R = 0$ zu diesem Zweck nicht mehr benötigt wird.

Da die Signalkombination $S = R = 1$ ohnehin verboten ist, brauchen wir (neben CLOCK) nur zwei verschiedene Zustände der anderen Eingangsleitungen, um die Schaltung alle ihre Funktionen ausführen zu lassen, und wenn wir bereit sind, auf die Kombination $S = R = 0$, weil sie überflüssig ist, bewusst zu verzichten, können wir die verbotene Kombination $S = R = 1$ von vornherein ausschließen.

Genau das macht folgende Modifizierung, das **gesteuerte D -Flipflop**, das nur einen Dateneingang hat und die ganze Steuerung mit dem Uhrensinal regelt:

Abbildung 2.18: Ein gesteuertes D -Flipflop

Das einzige Datensignal wird aufgeteilt und auf einem Zweig invertiert, so dass dem SR -Flipflop *immer* ein komplementäres Paar von Datensignalen zugeführt wird. Der verbotene Zustand *kann* so nicht zustande kommen. Trotzdem ist es möglich, den Speicher „ruhig“ (und stabil) zu stellen, weil die Datensignale das eigentliche Flipflop nur erreichen, wenn $CLOCK = 1$; zu allen anderen Zeiten werden von den AND Gattern Nullen weitergereicht und die vorhandenen Daten bleiben gespeichert.

Wir haben diese Flipflops entworfen, um einen 1-Bit Speicher zu realisieren, and natürlich kann man viele von ihnen zusammenbinden (mit einem gemeinsamen CLOCK Signal) um größere Speicher zu realisieren. Speicher

dieser Art werden **statische Speicher** genannt, weil sie wirklich ihren Inhalt verlustlos aufbewahren, so lange der Rechner unter Strom steht (damit die Betriebsspannung vorhanden ist) und das Steuersignal auf 0 steht. Sie haben auch den Vorteil, sehr schnell zu reagieren. Aber sie haben den Nachteil, relativ teuer zu sein und wegen des komplizierten Aufbaus viel Platz zu beanspruchen.

Deshalb werden statische Speicher hauptsächlich dort eingesetzt, wo es auf Geschwindigkeit ankommt: in Registern und im Cachespeicher der CPU, ein schnell adressierbarer Speicher in oder direkt angeschlossen an die CPU, in dem Daten und Befehle zwischengespeichert werden können, damit sie schnell verfügbar sind. Nur wenn die gesuchten Daten sich nicht im Cache befinden muss ein langsamer Zugriff auf den Hauptspeicher erfolgen, um die gewünschten Daten (und ihr Nachbarbereich, der vielleicht als Nächstes angesprochen wird) in den schnellen Speicher zu holen.

Es gibt aber neben den Flipflops eine zweite Art von Speicher, der pro Bit viel weniger Platz braucht und viel billiger ist, da für jedes Bit nur ein Kondensator zur Speicherung des Bits und ein Transistor, um das Bit zu setzen oder zurückzusetzen, benötigt werden. Diese Konstruktion ist viel langsamer als statische Speicher, aber ihr Hauptnachteil ist, dass die Kondensatoren langsam ihre Ladung verlieren und die Speicherung deshalb nicht dauerhaft ist. Um diesem Problem entgegenzutreten müssen die Speicherzellen in regelmäßigen Abständen von einigen Millisekunden gelesen und mit dem gleichen Wert neu geschrieben werden; man nennt dies **Auffrischen** (*refresh*) der Zellen. Wegen der Notwendigkeit eines ständigen Auffrischens heißt diese Art Speicher **dynamischer Speicher**. Der inzwischen meistens sehr große Hauptspeicher moderner Rechner besteht fast immer aus dynamischen Speicherbausteinen.

Natürlich sind die Hauptplatinen für die Aufnahme solchen Speichers vorbereitet und es gibt im Computeraufbau Bausteine, die nichts anderes tun, als den dynamischen Speicher turnusmäßig zu lesen und aufzufrischen. Darum muss sich der Rechnerentwickler kümmern; die CPU und der Programmierer müssen es aber nicht und sind von dieser Aufgabe entlastet.

Der Benutzer eines Rechners muss also in Prinzip nicht wissen oder darauf eingehen, welche Art von Speicher im Rechner wo eingebaut ist.

In einem modernen Rechner gibt es natürlich nicht nur *einen* 1-Bit Speicher, und auch nicht nur ein paar oder ein paar Tausende oder ein paar Millionen, sondern typischerweise schon ein paar Milliarden.

Durch diesen Umstand stellt sich uns ein neues Problem: wenn wir die Daten aus einer *bestimmten* Speicherstelle benötigen oder Daten in eine *bestimmte* Speicherstelle hinschreiben möchten, wie kann dafür gesorgt werden, dass die Datenleitungen mit genau diesem Speicherbit (oder eher mit

genau diesem Speicherwort) verbunden werden, und nicht mit einem oder vielleicht sogar allen der vielen anderen, die es gibt?

Zu diesem Zweck wird jedes Speicherwort mit einer n -Bit binären Zahl, genannt seine **Adresse**, gekennzeichnet, und der Rechner enthält Bausteine, die in der Lage sind, Adressen zu entziffern und die Datenleitung zu genau der beschriebenen Speicherstelle freizuschalten und alle anderen Datenleitungen zu sperren.

Eigentlich sind das zwei Aufgaben; wenn Speicherstellen gelesen werden, braucht man ein Baustein, der Eingaben von vielen Datenleitungen annehmen kann, aber gesteuert durch die Adresse nur die gewünschte Eingabe an seinen Ausgang weiterleitet. Wenn Speicherstellen beschrieben werden sollen (d. h., wenn Daten hineingeschrieben werden sollen), dann braucht man einen Baustein, der einen Eingang hat aber mehrere Ausgangsleitungen, und der gesteuert durch eine Adresse den Eingang auf genau den gewünschten Ausgang leitet (aber auf keinen anderen).

Bausteine, die die erste Aufgabe erfüllen, heißen **Multiplexer** (MUX). Bausteine, die die zweite Aufgabe erfüllen können, heißen **Demultiplexer** (DeMUX).

Hier ist zunächst eine schematische Darstellung eines Multiplexers (ohne die Schaltungen, die ihn realisieren).

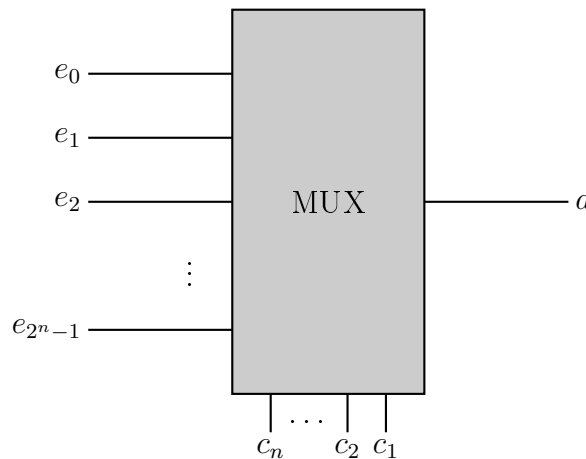


Abbildung 2.19: Ein Multiplexer (schematisch)

Der Multiplexer hat einen Ausgang, eine gewisse Anzahl n von *Steuereingängen* oder *Selektiereingängen* c_1, c_2, \dots, c_n , und 2^n *Dateneingänge* nummeriert von 0 bis $2^n - 1$. Die Steuereingänge werden zusammengefasst und als eine n -Bit binäre Zahl i gelesen, und bewirken, dass der i -te Eingang e_i zum Ausgang a weitergeleitet wird.

Hierbei könnte zum Beispiel i eine Speicheradresse sein und der Baustein würde also ein Bit aus dieser Adresse „auslesen“ und weiterreichen. In Kombination mit weiteren solchen Bausteinen könnten dann alle Bits eines Speicherworts weitergeleitet werden.

Es ist nicht schwer, einen Multiplexer zu bauen. Dazu braucht man neben den Steuerbits auch ihre Negativen, d. h., ihre Komplemente, und dann wird jede Eingangsleitung e_k mit den Signalen c_j oder \bar{c}_j (für alle j) geundet, wobei man für ein bestimmtes j die unnegierte Version c_j nimmt, wenn das j -te Bit (von rechts) der Zahl k in Binärschreibweise eine 1 ist, und die negierte Version \bar{c}_j nimmt, wenn das j -te Bit von rechts der Zahl k in Binärschreibweise eine 0 ist.

Das AND der so modifizierten Steuersignale ist genau dann 1, wenn die Werte der Steuersignale genau dem Bitmuster der Zahl k entsprechen; also ist das AND der Steuersignale mit e_k gleich dem Wert von e_k , wenn die Steuersignale dem Bitmuster von k entsprechen, und ist 0 in allen anderen Fällen, also für alle anderen k .

Wenn man das OR der Werte aller 2^n ANDs bildet, werden $2^n - 1$ Nullen mit einem einzigen Eingangssignal e_i geodert, und zwar mit dem Eingangssignal, dessen Index i der Wert der von den Steuersignalen gebildeten Binärzahl $i = c_n c_{n-1} \dots c_1$ ist.

Wir illustrieren dieses Prinzip anhand des Schaltdiagramms eines 2-Bit Multiplexers (der $2^2 = 4$ Dateneingänge hat).

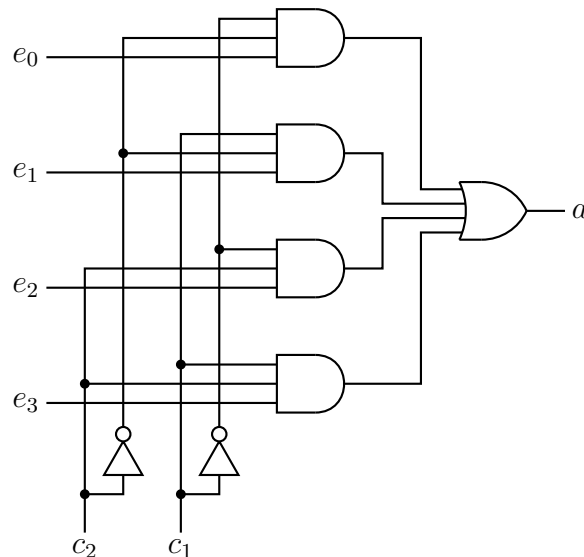


Abbildung 2.20: Ein Multiplexer mit 4 Eingängen

Natürlich kann man nach dem gleichen Prinzip auch Multiplexer mit mehr

Eingängen bauen, etwa mit 3 Steuerbits und 8 Dateneingängen, mit 4 Steuerbits und 16 Dateneingängen usw.

Multiplexer haben wir eingeführt wegen ihrer Verwendung zur Speicheradressierung. Moderne Rechner haben aber einen Adressraum von 4 GB oder mehr und enthalten auch etwa so viele Speicherzellen. Man wird kaum einen Multiplexer bauen können, der eine Milliarde Eingangsleitungen hat! In Wirklichkeit wird mit Multiplexern der Speicherbereich stufenweise selektiert, d. h., mit einem Multiplexer wird zunächst ein größerer Speicherblock selektiert und sein Ausgang weitergereicht, während weitere Multiplexer nur auf diesen Block wirken und darin einen (jetzt nicht mehr so großen) Teilbereich für die Weitergabe auswählen, und dies fortgesetzt bis schließlich doch einzelne Zellen selektiert werden können.

Multiplexer werden für Lesevorgänge aus einem großen Adressbereich benötigt. Für Schreibvorgänge braucht man einen **Demultiplexer**, der das zu schreibende Bit auf einen von mehreren Ausgängen leitet (nämlich auf denjenigen Ausgang, dessen Nummer in den Selektiereingängen steht) und alle anderen Ausgänge auf Null setzt.

Im schematischen Bild Abbildung 2.19 müssen nur der Ausgang und die Eingänge die Plätze tauschen, um die schematische Zeichnung eines Demultiplexers zu erhalten (oder man lässt die Daten im *unveränderten* Bild von rechts nach links laufen).

Im Schaltdiagramm 2.20 muss man nur die Eingänge e_i der AND Gatter statt ihrer dortigen Verbindung an den Ausgang des Gatters gesetzt werden, so dass Daten auf diesen Leitungen die Schaltung verlassen, statt dort einzugehen. An die freigewordenen Eingangsstellen schaltet man ein einziges Eingangssignal e , dass durch die Selektierung durch die Steuersignale auf den selektierten Ausgang, d. h., auf den Ausgang des selektierten Gatters geschrieben wird. Das OR Gatter kann entfallen, da man kein Gatter braucht, um ein einzelnes Signal auf mehrere Leitungen gleichzeitig zu legen (das ist ja wesentlich einfacher, als mehrere eventuell verschiedene Eingänge zu einem Signal zu kombinieren).

Noch geeigneter für die Adressenselection als ein Demultiplexer ist ein **Dekodierer** oder **Decoder**, der auf die gleiche Weise wie beim Demultiplexer unter Kontrolle von Steuerbits eine Ausgangsleitung auswählt, aber auf diese Leitung keine variablen Daten schreibt (die eingegeben werden müssen), sondern die selektierte Leitung auf 1 setzt und alle anderen Ausgänge auf 0 setzt. Diese Ausgänge kann man dann zur selektiven Freigabe einer einzigen Speicherzelle (und Sperrung aller anderen) verwenden, wobei die zu speichernden Daten an alle beteiligten Speicherzellen geschickt werden. Nur in der selektierten Zelle werden sie tatsächlich gespeichert; der Inhalt aller anderen Speicherzellen bleibt erhalten, wie er war.

Demultiplexer und Decoder haben einen sehr ähnlichen Aufbau und jedes kann mit wenig zusätzlichen Bauteilen dazu gebracht werden, die Funktion des anderen auszuführen.

Es wurde schon hingewiesen auf die Notwendigkeit, im Rechner einen Taktgeber zu haben, damit Daten, die zusammen verarbeitet werden, oder Vorgänge, die miteinander koordiniert sind, synchronisiert werden können. Wir wollen kurz erläutern, wie solche Uhren beschaffen sind und wie sie eingesetzt werden zur Steuerung der Abläufe im Computer.

Die Grunduhr in einem Rechner ist meistens eine Quarzuhr, gesteuert durch die Schwingungen eines Kristalls. Sie produziert ein *symmetrisches* Signal der Gestalt



das genau so lange den Zustand 1 einnimmt, wie den Zustand 0. Die Gesamtdauer einer Schwingung (Zustand 1 gefolgt von Zustand 0) nennt sich ein **Zyklus** der Uhr. Mit dem Uhrensinal kann man Schaltungen steuern, indem man das Uhrensinal durch ein AND Gatter mit den Eingängen der Schaltung schickt (wie bei den gesteuerten *SR*-Flipflops und *D*-Flipflops), oder indem man mit der Uhr elektronische Schalter steuert oder das Uhrensinal auf Freigabeeingänge von Bausteinen legt, damit diese ihre Funktion nur ausführen können, wenn das Uhrensinal den Wert 1 hat.

Diese Art von Steuerung reicht allerdings nicht ganz aus, weil die Zeiteinteilung durch ein symmetrisches Uhrensinal zu grob ist (die Uhr steht zu lange auf 1). Wenn Abläufe wirklich gleichzeitig ausgeführt werden müssen, will man sie nicht mit dem Pegel der Uhr steuern, sondern mit einem kurzen Impuls, etwa in dem Moment, in dem die Uhr von 0 auf 1 geht (man nennt diesen Zeitpunkt die **steigende Flanke** der Uhr; natürlich hat das Uhrensinal auch eine **fallende Flanke**, wenn es von 1 auf 0 zurückgeht).

Es gibt verschiedene Hilfsmittel, um den Taktzyklus der Uhr in kleinere Zeitintervalle zu unterteilen, oder um Impulse zu erzeugen, die die steigende oder fallende Flanke der Uhr markieren.

Betrachten wir zum Beispiel folgende Schaltung:

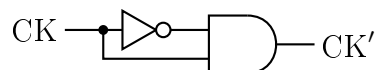


Abbildung 2.21: Ein Impulsgeber für die steigende Flanke der Uhr

Zunächst sieht es so aus, als ob das Ausgangssinal CK' immer auf 0 stehen müsste, denn die Eingänge zum AND Gatter sind immer komplementär,

da sie von einem einzigen Signal stammen, das vor genau einem der Eingänge zum AND invertiert wird.

Das stimmt auch fast, aber nicht hundertprozentig. Es stimmt auf jeden Fall, wenn der Zustand des Signals CK seit einiger Zeit besteht, weil dann das NOT Gatter wirklich dafür sorgt, dass der zweite Eingang des ANDs genau den anderen Bitwert hat, als der erste.

Aber wenn der Zustand von CK sich ändert, dann spürt der untere Eingang des ANDs die Änderung *sofort*, der obere Eingang aber erst mit Verzögerung, weil das geänderte Signal zuerst das NOT Gatter durchlaufen muss. Das kostet ein kleines bisschen Zeit, ein winziges Zeitintervall, während dessen der untere Eingang schon den neuen Wert hat, aber der obere Eingang noch den alten Wert hat.

Für diese kurze Zeit sind beide Eingänge des ANDs gleich und ihr Wert wird am Ausgang angezeigt.

Wenn der neue Uhrenpegel Null ist, ist der Ausgang des ANDs auch 0, und weil dies auch der Normalwert ist, der ausgegeben wird, wenn beide Zweige am Eingang des ANDs ihren stabilen Zustand erreicht haben, bleibt das Ausgangssignal so, wie es auch kurz vorher war und kurz nachher wieder sein wird. Das heißt, die fallende Flanke des Uhrenssignals macht sich am Ausgang CK' nicht bemerkbar.

Aber wenn die Uhr von 0 auf 1 geht, dann sind für kurze Zeit beide Eingänge des ANDs Eins (der untere, weil dies der neue Zustand von CK ist, der obere, weil dies der alte Zustand von \overline{CK} ist), und der Ausgang CK' nimmt für ganz kurze Zeit den Zustand 1 an.

In anderen Worten, diese Schaltung produziert einen kurzen Impuls mit Wert 1 bei jeder *steigenden* Flanke der Uhr, aber sie reagiert nicht auf die fallende Flanke.

Mit diesem Impuls kann man Abläufe zeitgenauer steuern, als nur mit dem Pegel der Uhr.

Mit einem ähnlichen Trick kann man phasenverschobene Kopien des ursprünglichen Uhrenssignals erzeugen. Dazu zweigt man das Uhrensignal ab und schickt es durch eine Verzögerungsschaltung, die das Signal unverändert wieder ausspuckt, aber erst nach einer kurzen bekannten Zeitspanne. Als Verzögerungselement würde sich sogar schon ein nichtinvertierender Puffer eignen.

Das ursprüngliche und die abgewandelten UhrensSignale kann man invertieren oder durch ANDs kombinieren um viele neue Signale, teils mit kürzeren Pegelintervallen oder mit Impulsen zu den Zeiten der verschiedenen Flanken der UhrensSignale zu erzeugen. Auf diese Weise kann ein grobes Taktsignal verfeinert werden und können kürzere Zeitintervalle als ein voller Taktzyklus zuverlässig gemessen werden.

Ein wichtiger Grund, warum wir eine solche feinere Zeitmessung brauchen, ist dass die meisten Rechner versuchen, viele Verarbeitungsschritte innerhalb eines einzigen Taktzyklus auszuführen. So will man gerne Speicherzugriffe in einem Taktzyklus ausführen können oder einfache Maschinenbefehle (für die immerhin ein kleines Mikroprogramm ablaufen muss) in nur einem Zyklus ausführen.

Dazu ist aber mehr als nur ein feinerer Takt nötig. Damit die richtigen Schritte in der richtigen Reihenfolge und zur richtigen Zeit ausgeführt werden, und damit die falschen Schritte blockiert werden, wenn sie nicht an der Reihe sind, muss man nicht nur das Ticken der Uhr hören können, sondern auch wissen, genau wie spät es ist, d. h., wie oft die Uhr schon getickt hat. Natürlich geht es nicht um die Uhrzeit im Alltagssinn (obwohl sie für die Benutzer und für die Buchhaltung wichtig sein kann und deshalb meistens auch verfügbar ist), aber man braucht schon Kennzeichnungen, die verschiedene Taktunterteilungen eindeutig identifizieren, um damit einen schrittweisen Prozessablauf richtig steuern zu können.

Zu diesem Zweck benutzt man **Mikrozähler** (μZ), Schaltungen mit mehreren Ausgängen, die bei jeder Taktunterteilung einen Zähler hochzählen und den aktuellen binären Wert des Zählers an den Ausgängen angeben. Wenn ein Ablauf mit n Schritten innerhalb eines Grundtaktzyklus zu steuern ist, zählt der Mikrozähler von 0 bis $n - 1$ genau n Zustände durch und beginnt dann wieder mit 0 am Anfang des nächsten Taktzyklus. Weil sie zyklisch (also um einen Ring von Werten herum) zählen, werden Mikrozähler auch **Ringzähler** genannt.

Der Mikrozähler zählt natürlich binär und braucht deshalb k Ausgänge, um seinen Zustand wiederzugeben, wobei k die kleinste Zahl ist, so dass $2^k \geq n$ (eine praktischere Beschreibung wäre: k ist die Anzahl der Bits in der Binärdarstellung von $n - 1$, der größten zu zählenden Zahl).

Intern rechnet der Mikrozähler auch mit k Bits. Er braucht k Flipflops, um den Zählerstand zu speichern, bis er wieder ein Signal erhält, dass er weiterzählen soll, und er implementiert k boolesche Funktionen von k Variablen, um den neuen Zählerwert als Funktion des alten Zählerwertes zu berechnen. In anderen Worten, sein gespeicherter Zustand ist gleichzeitig die Eingabe zu einer *kombinatorischen* Schaltung, die den nächsten Zählerstand berechnet.

Als Beispiel beschreiben wir näher einen Mikrozähler, der 6 Zustände zählen soll. Dann ist $n = 6$ und wir müssen von 0 bis 5 zählen und danach wieder ab 0. Die Binärdarstellung 101 von 5 hat drei Bits; also brauchen wir drei Dateneingänge x_1, x_2, x_3 für den kombinatorischen „hochzählenden“ Teil der Schaltung, und dieser Schaltungsteil hat drei Ausgänge z_1, z_2, z_3 , die das Ergebnis des Hochzählens wiedergeben. Die Wertetabelle der booleschen Funktionen für das Hochzählen sieht so aus:

x_1	x_2	x_3	z_1	z_2	z_3
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	0	0	0

Die KDNs für die Ausgangsbits sind

$$z_1 = \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 \bar{x}_3, \quad z_2 = \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3, \quad z_3 = \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 \bar{x}_3$$

und nur der letzte Ausdruck lässt sich vereinfachen, und zwar zu

$$z_3 = \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_3 = (\bar{x}_1 + \bar{x}_2) \bar{x}_3.$$

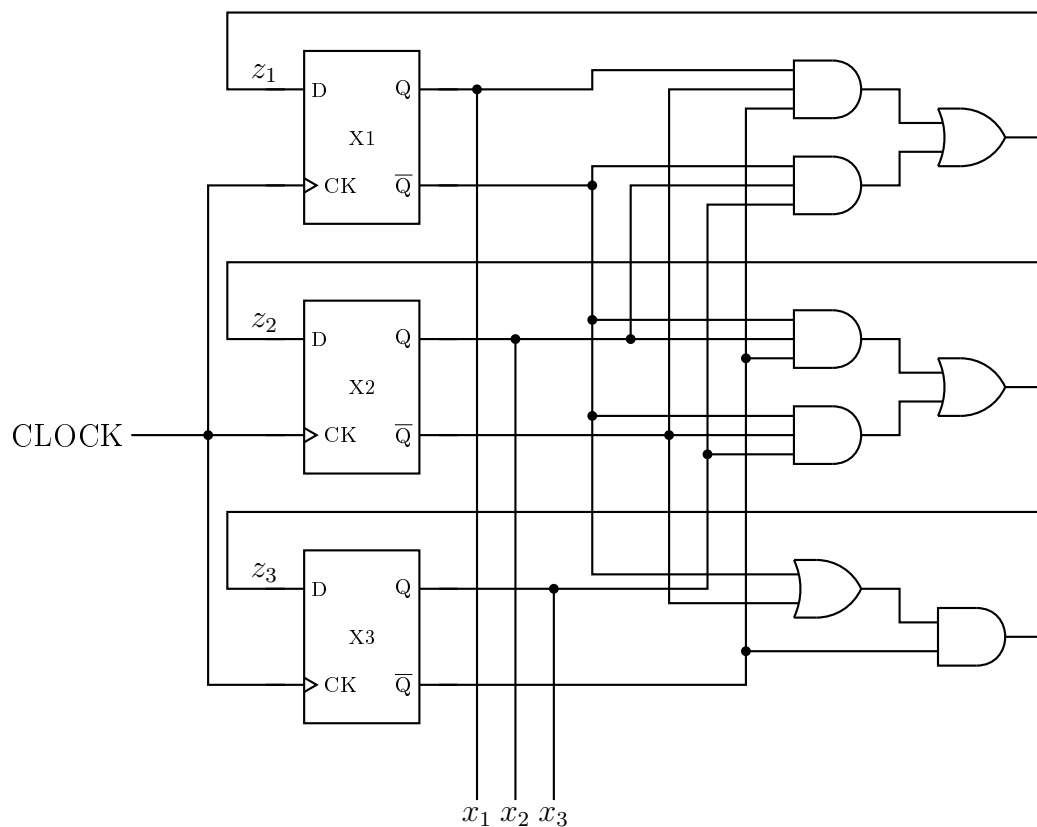


Abbildung 2.22: Ein Mikrozähler mit 6 Zuständen

Eine Schaltung aus D -Flipflops (zur Speicherung des momentanen Zählerstands) und Gattern (zur Berechnung der booleschen Funktionen z_i und

somit des nächsten Zählerstands), die den Mikrozähler realisiert, sehen Sie in Abbildung 2.22 auf der vorherigen Seite.

Beachten Sie, dass die *D*-Flipflops zu jedem Zeitpunkt den *aktuellen* Zählerstand gebildet aus x_1 , x_2 und x_3 als gespeicherte Bits enthalten und diese Bits (und ihren komplementären Wert) an ihren *Ausgängen* angeben, von wo aus man sie ablesen kann. Gleichzeitig wird dieser aktuelle Zustand an die Gatter auf der rechten Seite weitergeleitet, die den *neuen, zukünftigen* Zustand berechnen und am Ausgang der am weitesten rechts liegenden Gatter ausgeben, von wo aus sie an die *Eingänge* der Flipflops gebracht werden.

Sie werden aber noch *nicht* in den Flipflops gespeichert, sondern stehen nur bereit für eine Speicherung, die erst dann stattfindet, wenn das CLOCK Signal auf 1 geht! Der neue Zählerstand kann dann auch kurze Zeit später am Ausgang der ganzen Schaltung (am unteren Rand des Diagramms) abgelesen werden. Diese Daten stehen zwar nicht sofort bereit, wenn CLOCK auf 1 geht, aber sie werden nur um die Zeitspanne verzögert, die zur stabilen Speicherung in den Flipflops erforderlich ist, nicht aber um die Zeit, die zur *Berechnung* des neuen Zählerstandes oder zum Durchlaufen der Gatter rechts benötigt wird; diese Berechnung fand ja schon vorher statt, als das CLOCK Signal noch auf 0 stand. Es muss nur darauf geachtet werden, dass CLOCK nicht zu früh die Speicherung freigibt, sondern lange genug auf 0 bleibt, damit die Signale alle Gatter passieren können und ein stabiles Rechenergebnis an den Flipflopeingängen vorliegt.

Um die Schaltung nicht noch unübersichtlicher zu machen, haben wir keine Vorrichtung vorgesehen, um den Zähler auf 0 zu setzen; bei einem wirklichen Zähler wäre das auch implementiert.

Wir haben jetzt einige der wichtigsten „Hilfsschaltungen“ beschrieben, die nicht direkt zum Rechenergebnis einer Berechnung oder einer Datenverarbeitung beitragen, aber die fast wichtiger als die eigentliche Recheneinheit sind, weil sie die innere Organisation und den Prozessablauf im Rechner regeln helfen, weshalb ohne sie kein Rechner funktionieren kann.

Die Beschreibung des eigentlichen Rechenwerks werden wir bald nachholen. Aber schon jetzt ist klar, dass ein Rechner sehr viele und sehr verschiedenartige Schaltungen braucht, und dass einige von ihnen maßgeschneidert sein müssen für bestimmte Aufgaben, die nur im speziell vorliegenden Rechnermodell auftreten und in anderen Modellen mit anderer Dimensionierung oder anderen Strukturmerkmalen ganz anders ausfallen können.

Die Herstellung spezieller Bausteine für spezielle Aufgaben ist kostspielig. Deshalb nutzt man die theoretische Erkenntnis aus, dass jede boolesche Funktion eine effiziente Beschreibung als disjunktive Normalform hat, um *allgemeine* digitale Logikbausteine zu konstruieren, die „programmiert“ werden können, um sie für beliebige spezielle Aufgaben verwendbar zu machen.

Ein wichtiges Beispiel für solche Bausteine sind die **Programmable Logic Arrays** („programmierbare Logikanordnungen“), kurz **PLA** genannt, die wir jetzt beschreiben wollen.

Ein PLA besteht aus einer großen Anzahl von Feldern („Zellen“) mit Gitterpunkten, in denen sich waagerechte und senkrechte Verbindungen kreuzen und je nach Programmierung unverändert weitergeleitet werden oder durch AND oder OR Gatter geschleust und auf diese Weise logisch kombiniert werden können.

In der grafischen Darstellung von PLAs und den Gitterpunkten oder Verbindungskreuzungen werden wir diese Kreuzungen tatsächlich immer als kleine Zellen zeichnen und diese Zellen in größeren Zeichnungen aneinander reihen, weil die Signalwege so am besten dargestellt werden können. In einem wirklichen PLA kann die geometrische Anordnung der Kreuzungen ganz anders sein, so dass das Wort *Zelle*, das wir zur Bezeichnung verwenden, nicht allzu wörtlich verstanden werden darf. Es handelt sich nur um Leitungsverbindungen oder Kreuzungen, die nach einem festen Muster im PLA angeordnet sind oder miteinander verbunden sind. Dieses Muster muss nicht so aussehen, wie in unseren Zeichnungen.

In dem (typischen) Aufbau, den wir beschreiben wollen, gibt es drei Sorten von Gitterpunkten oder Zellen, wie in Abbildung 2.23 auf der nächsten Seite gezeigt.

Diese Typen werden wir auch mit $\boxed{0}$, $\boxed{1}$ und $\boxed{2}$ kennzeichnen.

Das Funktionsprinzip eines PLA besteht darin, dass die Gitterpunkttypen nicht ein für alle Mal festgelegt sind, sondern sich leicht verändern lassen. In dieser Veränderung der Zelltypen besteht die Programmierung, die es erlaubt, in einem PLA Schaltkreise zu realisieren, die beliebige boolesche Funktionen berechnen können.

Für die Programmierung werden verschiedene Konstruktionen verwendet. Bei der Herstellung eines PLAs enthalten alle Zellen ein OR oder ein AND Gatter, wie in Abbildungen 2.23(b) oder 2.23(c), aber diese Gatter sind nicht immer aktiviert (oder mit beiden Dateneingängen verbunden). Wenn sie aktiviert oder verbunden sind, verhält sich die Zelle wie Typ (1) oder (2), aber wenn die Gatter deaktiviert oder mit einem Eingang nicht verbunden sind, lassen sie den anderen Eingang unverändert durch und die Zelle verhält sich wie Typ (0).

Es gibt PLAs, bei denen jede Zelle neben den Dateneingängen auch einen Steuereingang hat, und wenn das auf die Zelle gelegte Steuersignal den Wert 0 hat, dann verhält sich die Zelle wie Typ (0), aber wenn der Steuereingang 1 ist, dann wird das in der Zelle befindliche OR oder AND Gatter freigegeben und die Zelle verhält sich wie Typ (1) oder (2). In diesem Aufbau werden keine Typ (0) Zellen eingerichtet, aber die Steuerung kann durch deaktivierung der

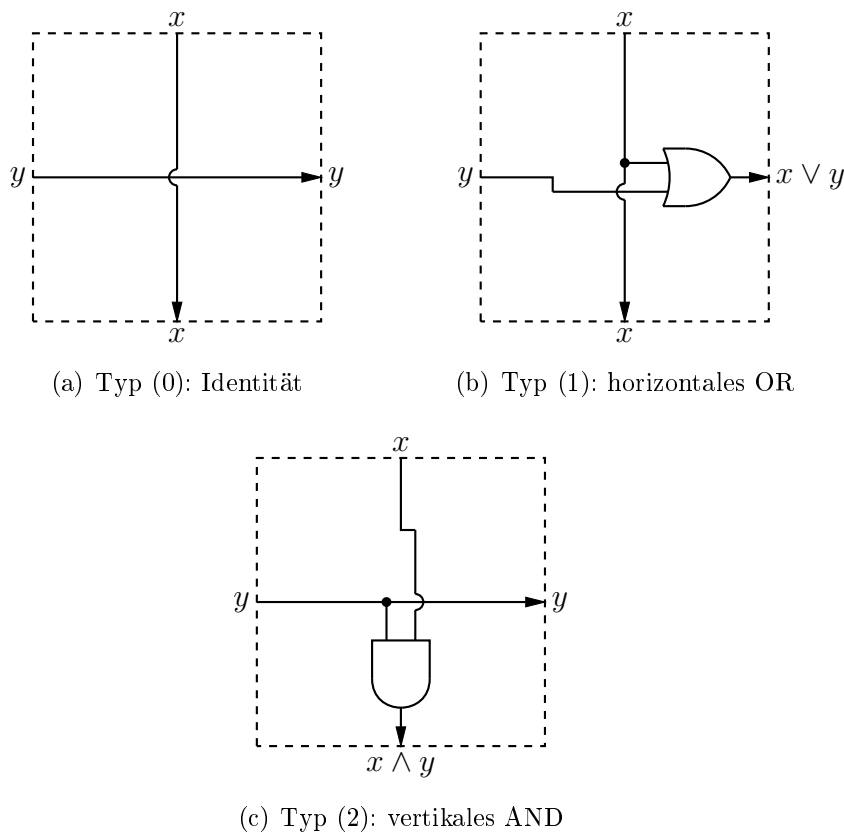


Abbildung 2.23: PLA Gitterpunkttypen

Gatter bewirken, dass die Zelle sich wie Typ (0) verhält.

Ein Nachteil dieses Systems ist, dass das PLA seine Programmierung nur so lange behält, wie der Strom eingeschaltet ist und die Steuersignale erzeugt werden. Wird der Rechner ausgeschaltet und später wieder eingeschaltet, muss das PLA neu programmiert werden.

Gleichzeitig ist dies aber auch ein Vorteil; funktioniert die Schaltung nicht so, wie gewünscht, oder wird ihr Design verbessert, so kann man den neuen Entwurf sofort durch Umprogrammierung im gleichen PLA einrichten, ohne ein neues PLA kaufen zu müssen.

Um die Stabilität der logischen Schaltkreise zu erhöhen setzt man auch gerne PLAs ein, die eine permanente Programmierung haben und nicht bei jedem Einschalten des Rechners neu eingerichtet werden müssen. Natürlich können solche PLAs dann nur einmal programmiert werden, und wenn etwas an der Programmierung nicht stimmt oder verbessert werden muss, muss man das PLA wegwerfen und einen neuen kaufen. Das ist nicht wirklich ein

großer Nachteil, denn PLAs sind Standardbausteine und nicht sehr teuer in der Herstellung (die empfohlenen Wiederverkaufspreise im gegenwärtigen im Internet abrufbaren Angebot von Texas Instruments rangieren zwischen \$ 1.34 und \$ 16.16 pro Stück bei Abnahme von tausend Stück).

Eine früher gebräuchliche Technik für die dauerhafte Programmierung bestand darin, auf den Verbindungen zwischen den vertikalen Leitungen und den OR Gattern bei Typ (1), oder zwischen den horizontalen Leitungen und den AND Gattern bei Typ (2), kleine Sicherungen zu setzen. So lange diese intakt waren, wurden beide in die Zelle eingehende Datenbits durch das Gatter geschickt und die Zelle hatte den Typ (1) oder (2), je nach dem enthaltenen Gatter.

Zur Programmierung konnte man an die zu deaktivierenden Zellen eine hohe Spannung anlegen, die die Sicherungen zerstörten und somit die Querverbindungen in der Zelle sperrten. Fortan wurde jedes der Datensignale für sich und unabhängig vom anderen unverändert durch die Zelle geschickt und die Zelle wurde zu einer Typ (0) Zelle.

Weil aber die zerstörten Sicherungen im Laufe der Zeit durch kristallines Wachstum wieder zuwachsen konnten und manchmal wieder leitend wurden, bevorzugt man heute eine umgekehrte Technik. Anstelle der Sicherungen befinden sich Dioden, die im Urzustand keinen Strom durchlassen. Dann verhalten sich alle Zellen wie Typ (0).

Um einige Zellen zu aktivieren und in Typ (1) oder (2) zu verwandeln, legt man an die Dioden eine hohe Spannung an, die sie zerstört. Die Verbindung wird dann leitend und die Gatter kombinieren beide Dateneingänge der Zelle, so dass die Zelle den zu ihrem Gatter passenden Typ (1) oder (2) annimmt. Wir zeigen in Abbildung 2.24 auf der nächsten Seite noch einmal diese Konstruktion (vor Zerstörung der Dioden, so dass die Zellen in der gezeigten Konfiguration sich wie Typ (0) verhalten).

PLAs sind programmierbar in dem Sinne, dass der Typ ihrer Gitterpunkte sich verändern lässt, aber die Typen sind nicht beliebig austauschbar. Es ist vielmehr so, dass jede Zelle zu einem festen *potentiellen* Typ (1) oder (2) gehört, und die Programmierung besteht darin, dass diese potentielle Eigenschaft selektiv bei manchen Zellen aktiviert oder zur Geltung gebracht werden kann, und bei anderen Zellen unterdrückt werden kann, so dass die Zelle den Typ (0) annimmt.

Um es anders zu sagen: bei jeder Zelle kann zwischen Typ (0) und nicht Typ (0) gewählt werden, aber welcher der Typen (1) oder (2) die für die Zelle geltende Alternative zu Typ (0) ist, ist für jede Zelle vorbestimmt und kann nicht frei ausgesucht werden.

Damit ein PLA als allgemeine logische Schaltung verwendet werden kann, ist er intern so organisiert, dass boolesche Funktionen in der günstigen und immer realisierbaren *disjunktiven Normalform* leicht „einprogrammiert“ wer-

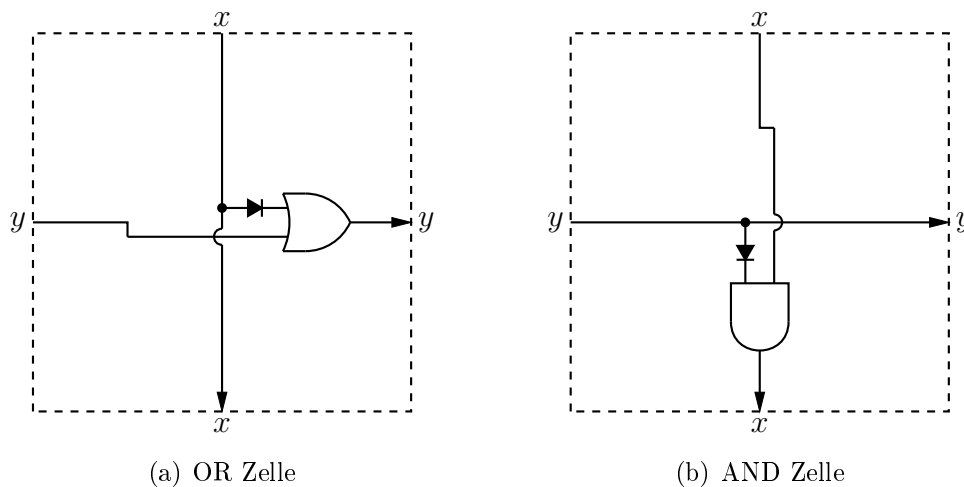


Abbildung 2.24: PLA Zellen mit Dioden zur Programmierung

den können. Die Eingangssignale werden zunächst in ein Gebiet geleitet, in denen es nur potentielle Typ (2) Zellen gibt (die aber auch Typ (0) annehmen können); dieser Teil des PLA nennt sich die **Und-Ebene**. Typ (2) Zellen lassen waagrecht passierende Daten unverändert aber verknüpfen senkrecht passierende Daten mit Und.

Das durch die Und-Ebene verarbeitete Ergebnis verlässt diesen Bereich also am unteren Ende, und dort schließt die **Oder-Ebene** an, ein Bereich, in dem alle Zellen den potentiellen Typ (1) haben (aber durch Programmierung auch zu Typ (0) bestimmt werden können). Typ (1) Zellen verarbeiten waagrecht passierende Daten und lassen die senkrecht passierenden unverändert. Das Ergebnis der Verarbeitung erscheint also am rechten Rand der Oder-Ebene und bildet dort den Ausgang des PLAs.

Die *Zeilen* der Und-Ebene haben als Eingänge an der linken Seite die Eingangsvariablen und ihre Komplemente, und diese werden zu allen Spalten durchgeschleust ohne Veränderung. Jede Spalte der Und-Ebene hat am oberen Ende eine Verbindung zur Betriebsspannung, d. h., einen Eingang des Bits 1 als Initialzustand der Bearbeitung, und die Spalte enthält einige Typ (0) Zellen und einige Typ (2) Zellen. Die Typ (0) Zellen machen nichts und die Typ (2) Zellen „unden“ das Ergebnis des oberhalb von ihnen liegenden Teils der Spalte mit der Variablen oder negierten Variablen, die von links in die Zelle eingeht. Am unteren Ende der Spalte kommt also heraus das Und aller Variableneingänge der Typ (2) Zellen. Was herauskommt ist also (als Und von Variablen) ein *Konjunktionsterm*, und man richtet eine Spalte ein, um einen bestimmten Konjunktionsterm zu berechnen, indem man die Zellen

zu Variablen, die in dem Konjunktionsterm erscheinen, zu (2) Zellen macht, und alle anderen Zellen in der Spalte zu (0) Zellen macht.

Zur Illustration: folgende Spalte der Und-Ebene (Abbildung 2.25) berechnet den Konjunktionsterm $\bar{a}b\bar{c}$.

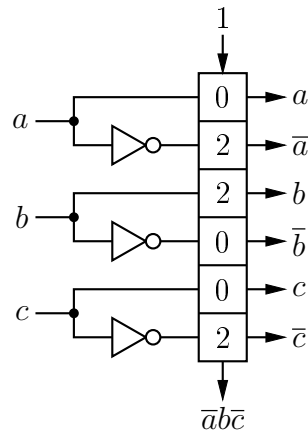


Abbildung 2.25: Eine programmierte Spalte der Und-Ebene

Die Signale, die von links in die Zellen eingehen, kommen unverändert auf der rechten Seite wieder heraus. Die Signale, die von oben in die Zellen hineingehen, werden unverändert nach unten weitergereicht, wenn die Zelle eine $\boxed{0}$ Zelle ist, aber bei einer $\boxed{2}$ Zelle wird das Und des von oben und von links eingehenden Signals nach unten weitergereicht.

Betrachtet man die sechs Zellen in der abgebildeten Spalte in der Reihenfolge von oben nach unten, so haben ihre unteren Ausgänge die Werte 1 für die oberste $\boxed{0}$ Zelle, $\bar{a} \wedge 1 = \bar{a}$ für die darunterliegende $\boxed{2}$ Zelle, dann $b \wedge \bar{a} = \bar{a}b$, welches unverändert über zwei Zellen hinweg weitergereicht wird, und schließlich $\bar{c} \wedge \bar{a}b = \bar{a}b\bar{c}$ am unteren Ausgang der letzten Zelle und somit auch der Spalte.

Die unteren Ausgänge der Und-Ebene werden in die Oder-Ebene weitergeleitet. Die *Spalten* der Oder-Ebene erhalten diese Daten als Eingänge an der oberen Seite, und diese Eingänge werden ohne Veränderung zu allen Zeilen nach unten durchgereicht.

Jede Zeile der Oder-Ebene hat am linken Ende eine Verbindung zur Masse, d. h., einen Eingang des Bits 0 als Initialzustand der Bearbeitung. Im Allgemeinen enthält die Zeile einige Typ (0) Zellen und einige Typ (1) Zellen. Die Typ (0) Zellen machen nichts und die Typ (1) Zellen „odern“ das Ergebnis des links von ihnen liegenden Teils der Zeile mit den Daten, die von oben in die Zelle eingehen. Am rechten Ende der Zeile kommt also das Oder aller Dateneingänge der Typ (1) Zellen heraus. Weil diese Dateneingänge

von den Ausgängen der Und-Ebene kommen, sind sie Konjunktionsterme, und die Zeile gibt deshalb an ihrem rechten Rand eine Summe von Konjunktionstermen heraus; das ist auf jeden Fall eine disjunktive Normalform (oder äquivalent zu einer disjunktiven Normalform, wenn mehrere Spalten den gleichen Konjunktionsterm berechnen).

Man kann eine Zeile dazu einrichten, eine bestimmte Summe von Eingängen zu berechnen, indem man die Zellen, deren Eingänge als Summanden gewünscht sind, zu (1) Zellen macht, und Zellen, in die unerwünschte Summanden eingehen, zu (0) Zellen macht.

Zur Illustration der Summenbildung: folgende Zeile der Oder-Ebene (Abbildung 2.26) berechnet die Summe der mittleren beiden von vier Eingängen:

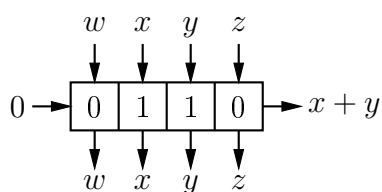


Abbildung 2.26: Eine programmierte Zeile der Oder-Ebene

Die Signale, die von oben in die Zellen eingehen, kommen unverändert unten wieder heraus. Die Signale, die von links in die Zellen hineingehen, werden unverändert nach rechts weitergereicht, wenn die Zelle eine $\boxed{0}$ Zelle ist, aber bei einer $\boxed{1}$ Zelle wird das Oder des von oben und von links eingehenden Signals nach rechts weitergereicht.

Betrachtet man die vier Zellen in der abgebildeten Zeile in der Reihenfolge von links nach rechts, so haben ihre rechten Ausgänge die Werte 0 für die $\boxed{0}$ Zelle ganz links, $x \vee 0 = x$ für die rechts davon liegende $\boxed{1}$ Zelle, dann $x \vee y = x + y$ für die nächste $\boxed{1}$ Zelle, und dieses Ergebnis wird unverändert durch die letzte $\boxed{0}$ Zelle weitergereicht und erscheint als Ausgang am rechten Ende der Zeile.

Offensichtlich kann man durch eine geeignete Kombination von programmierten Spalten der Und-Ebene und programmierten Zeilen der Oder-Ebene jede DN und somit jede boolesche Funktion berechnen; eine beliebige Gruppe von kombinatorischen Schaltkreisen kann also in einem ausreichend großen PLA realisiert werden.

Dabei lohnt es sich natürlich, für die zu berechnenden booleschen Funktionen möglichst effiziente disjunkte Normalformen zu suchen. Bei der Einrichtung des PLAs kann man zusätzlich Platz sparen, wenn man gleiche Konjunktionsterme, auch wenn sie in mehrere verschiedene DN eingehen, nur einmal

(d. h., nur in *einer* Spalte der Und-Ebene) berechnet, von wo aus man sie in mehreren Zeilen „abzapfen“ kann.

Beispiel 2.24 Als zusammenhängendes Beispiel für die Verwendung eines PLA zur gleichzeitigen Berechnung mehrerer boolescher Funktionen implementieren wir mit einem PLA den kombinatorischen Teil (also den „hochzählenden“ Teil rechts in Abbildung 2.22, ohne die Flipflops) vom Mikrozähler mit sechs Zuständen, der auf Seite 96 vorgestellt wurde.

Gleichzeitig implementieren wir in dem PLA einen vom Mikrozählerstand abhängenden Ausgang y , der sich vielleicht als Steuersignal eignen könnte, und der bei den Zählerständen $2 = 010_2$ und $5 = 101_2$ (und nur bei diesen) den Wert 1 annehmen soll. Die KDN für dieses Steuersignal als boolesche Funktion von x_1 , x_2 und x_3 ist

$$y = \bar{x}_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 x_3$$

und man sieht schnell, dass keine kürzere disjunktive Normalform diese boolesche Funktion ergeben würde.

Wir fassen kurz zusammen: wir haben drei Dateneingänge x_1 , x_2 und x_3 und vier Datenausgänge

$$\begin{aligned} z_1 &= \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 \bar{x}_3 \\ z_2 &= \bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 \\ z_3 &= \bar{x}_2 \bar{x}_3 + \bar{x}_1 \bar{x}_3 \\ y &= \bar{x}_1 x_2 \bar{x}_3 + x_1 \bar{x}_2 x_3. \end{aligned}$$

Das PLA braucht in der Und-Ebene eine Zeile für jede Eingangsvariable und je eine Zeile für ihre Komplemente, also sechs Zeilen. In der Oder-Ebene wird eine Zeile für jeden Ausgang benötigt, so dass die Oder-Ebene vier Zeilen hat.

Für jeden *verschiedenen* Konjunktionsterm in den Ausdrücken für die Ausgänge ist eine Spalte erforderlich. Insgesamt haben wir zwar 8 Konjunktionsterme, aber der zweite Summand von z_2 und der erste Summand von y sind gleich und können in *einer* Spalte berechnet werden, so dass unser PLA nur sieben Spalten haben muss.

Das programmierte PLA sehen Sie in Abbildung 2.27 auf der nächsten Seite. Dass wirklich die richtigen booleschen Funktionen am rechten Rand ausgegeben werden lässt sich leicht nachprüfen.

PLAs finden mannigfache Anwendungen in Computern. Die meisten Computer werden eingesetzt zur Steuerung von Echtzeitprozessen, zum Beispiel von Verkehrsampeln oder Waschmaschinen oder Automobilbremsen —

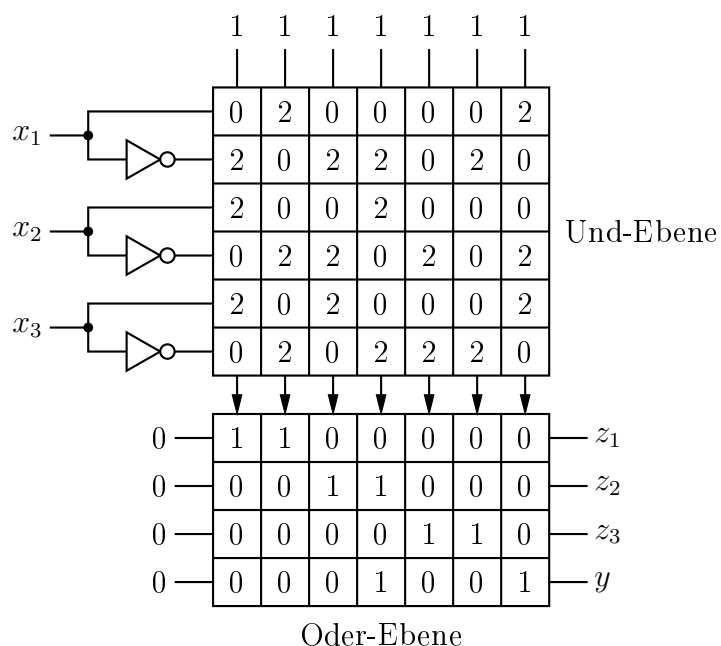


Abbildung 2.27: Ein PLA für den Mikrozähler mit 6 Zuständen und ein Steuersignal

viele Alltagsanwendungen werden heute computergesteuert, weil die einfachen dafür erforderlichen Computer sehr billig herzustellen sind. Für solche festen aber speziellen Anwendungen lohnt es sich, spezielle Logikbausteine als PLAs zu realisieren, die dann meistens schon bei der Herstellung nach Kundenwünschen fest programmiert werden.

Die typische Größe eines PLAs kann bis zu etwa 100 mal 100 Gitterpunkte betragen, wobei die im PLA realisierten Minterme und disjunktive Normalformen in der Regel sehr kurz sind, so dass viele Schaltungen in ein PLA hineinpassen.

Die Idee zur Anwendung von programmierbaren Logikbausteinen stammt von Maurice Wilkes, der sie 1951 neben dem Prinzip der Mikroprogrammierung als effizienten Grundsatz für die Konstruktion von Rechenmaschinen vorgeschlagen hat, um die Mikroarchitektur eines Rechners durch Austausch oder Neuprogrammierung des PLAs leicht verbesserbar und anpassbar zu machen. Man kann tatsächlich einen Großteil der kombinatorischen Logik in einer CPU mit PLAs realisieren.

Mit dieser Diskussion von PLAs wollen wir diesen Abschnitt über digitale Logik beenden, obwohl wir damit noch nicht ganz fertig sind. Wir haben jetzt die wichtigsten Bausteine zusammen, die unterstützende Dienste im Rechner

verrichten, aber noch keine Bausteine, die wirklich „rechnen“.

Weil hier aber nicht nur die Elektronik eine Rolle spielt, sondern auch die Art, wie die zu berechnenden Zahlen in einem Rechner dargestellt werden sollen, wollen wir diesem umfangreichen Thema ein eigenes Kapitel widmen, der nun folgt.

Kapitel 3

Rechnen mit Zahlen

Eine Sorte digitaler Logik haben wir in Kapitel 2 nicht besprochen, nämlich die *Recheneinheit* des Computers. Bevor wir das aber nachholen können, müssen wir etwas über die Zahlendarstellung im Rechner wissen.

In der Einleitung haben wir gesehen, dass elektronische Rechner am bequemsten im *Binärsystem* (Basis 2) rechnen können, aber damit ist noch nicht alles gesagt.

Wenn ein Mensch binär rechnet, dann schreibt er lange Folgen von Nullen und Einsen auf Papier, und wenn das Papier nicht ausreicht, nimmt er ein neues Blatt und schreibt weiter. Theoretisch können die behandelten Zahlen beliebig lang sein; bei langen Zahlen dauert die Berechnung etwas länger, aber mit Geduld bleibt sie durchführbar.

Computer können natürlich auch so rechnen, aber in der Regel nur mit Hilfe von *Software* (die gängigen Mathematikprogramme wie MAPLE und Mathematica bieten zum Beispiel arithmetische Operationen mit Zahlen beliebiger Länge an, und MAPLE kann ohne Weiteres auf einem mit 1,5 GHz getakteten Laptop in etwa 5 Sekunden π zu 100.000 Dezimalstellen ausrechnen).

Aber die *Hardwarerecheneinheit* verarbeitet Zahlen in Binärregistern, und diese haben eine feste Bitlänge von 8, 16, 32 oder 64 Bits in der Regel (manchmal können Register zu größeren Einheiten kombiniert werden, aber die Gesamtlänge ist trotzdem weniger als ein paar Hundert Bits).

Diese feste Bitlänge schränkt nicht nur die Größe oder die Genauigkeit der Rechenoperanden ein, sondern macht Probleme, wenn die Operanden zwar noch in die Register passen, aber das Ergebnis zu groß für die Register ist. Dieser Zustand nennt sich ein **Überlauf** (englisch **Overflow**) und führt zwangsweise zu falschen Rechenergebnissen (weil ein Bit verlorengeht, da kein Platz vorhanden ist, um ihn zu speichern); deshalb wird bei jeder Rechenoperation auf Überläufe geachtet und ein Zustands- oder Statusbit

gesetzt, wenn einer passiert.

Im folgenden gehen wir davon aus, dass die Recheneinheit unseres Rechners oder die Register, in denen die Operanden und das Ergebnis stehen, eine Bitbreite von r Bit haben.

In diesen r Bit müssen wir verschiedene Arten von Zahlen darstellen können. Dazu gehören

- unsignierte ganze Zahlen, also natürliche Zahlen
- signierte ganze Zahlen, also ganze Zahlen die entweder ≥ 0 aber auch < 0 sein dürfen
- Festkommazahlen, d. h., Binärbrüche in „Komma“-Darstellung, aber mit bekannter Lage des Kommas und einer festen Anzahl von Stellen hinter dem Komma
- Gleitkommazahlen, d. h., Binärbrüche in der „wissenschaftlichen“ oder Ingenieurnotation $\text{Mantisse} \times \text{Basis}^{\text{Exponent}}$. Diese Zahlen haben eine feste Anzahl von signifikanten Stellen und eine feste relative Genauigkeit, aber als absolute Zahlen können sie sowohl sehr groß wie auch sehr klein (also nahe bei 0) sein.

Wenn wir nur mit natürlichen Zahlen rechnen wollen, können wir sie als **unsignierte Zahlen** darstellen. Solche Zahlen sind nichtnegative ganze Zahlen mit einer Binärdarstellung, die maximal r Binärziffern hat. Die kleinste solche Zahl ist 0, die größte ist eine Folge von r Einsen und hat den Wert $2^r - 1$. In dieser Darstellung werden alle r verfügbaren Bits für die Präzisierung des absoluten Zahlenwertes verwendet, so dass der darstellbare Zahlenbereich relativ groß ist.

Um unsignierte Zahlen zu addieren, brauchen wir ein Rechenwerk, das die einfache binäre Addition, wie wir sie in Kapitel 1 erläutert haben, mit r -Bit Zahlen durchführen kann. Über den Entwurf solcher Rechenwerke werden wir etwas später sprechen.

Ein Überlauf passiert bei unsignierten r -Bit Zahlen genau dann, wenn das richtige Ergebnis $\geq 2^r$ ist, also wenn es einen *Übertrag* aus der höchsten Stelle gibt. Das dargestellte Ergebnis ist dann um 2^r zu klein.

Für **signierte (ganze) Zahlen** brauchen wir einen Bit, um das Vorzeichen der Zahl anzugeben, so dass nur noch $r - 1$ Bits für die Präzisierung des absoluten Werts verfügbar sind. Es gibt verschiedene Strategien dafür, wie die Kombination aus Vorzeichen und absolutem Wert am effizientesten durch eine Bitfolge dargestellt werden kann.

Die einfachste Idee besteht darin, das höchste Bit als Vorzeichenbit zu nehmen und den absoluten Wert in den niederen $r - 1$ Bits anzugeben. Das ist aber nicht effizient (und deshalb kaum gebräuchlich), weil für die Addition von Zahlen mit verschiedenen Vorzeichen eine spezielle *Subtrahiereinheit* und eine Fallunterscheidung (welcher Summand hat den größeren Absolutbetrag) erforderlich sind.

Es gibt bessere Systeme, die es möglich machen, für signierte Zahlen die gleiche Addiereinheit zu verwenden, wie für unsignierte (eventuell mit einem zusätzlichen aber schnellen und einfachen Rechenschritt zur Anpassung des Ergebnisses).

Die wichtigste und gebräuchlichste dieser Ideen ist die Darstellung von signierten Binärzahlen im **Zweierkomplement**. Die gleiche Idee kann man auch in anderen Zahlenbasen anwenden, und wir erläutern sie deshalb zunächst an kurzen Beispielen im Dezimalsystem mit drei Ziffern (weil das dezimale Rechnen uns am geläufigsten ist).

Betrachten wir also Zahlen geschrieben mit r Ziffern in einer Basis b . Nichtnegative Zahlen a werden im b -erkomplementsystem normal dargestellt, aber dürfen nur $r - 1$ Ziffern lang sein, weil die r -te Ziffer (von rechts) das *Vorzeichen* angibt. Eine *negative* Zahl $-a$ (wo a selber positiv und höchstens $r - 1$ Ziffern lang sein soll) wird durch die positive Zahl

$$b^r - a \tag{3.1}$$

dargestellt. Da $a < b^{r-1}$ ist, muss die höchste (also r -te) Ziffer von (3.1) gleich $b - 1$ sein, und daran kann man auch erkennen, dass die Ziffernfolge eine negative Zahl darstellt. Die höchste Ziffer $b - 1$ ist das Minuszeichen in dieser Darstellung, und bei nichtnegativen Zahlen ist die höchste Ziffer 0; das ist das Pluszeichen. Andere Werte als 0 oder $b - 1$ (welches immer ungleich 0 ist) sind für die höchste Ziffer nicht zulässig.

Bei Binärzahlen schöpfen die Möglichkeiten 0 und $b - 1 = 2 - 1 = 1$ für das höchste Bit alle Möglichkeiten für eine Binärziffer aus, so dass es im Zweierkomplement keine verbotenen höchsten Ziffern gibt.

Wir illustrieren die b -Komplementdarstellung mit Dezimalzahlen ($b = 10$) und drei Ziffern insgesamt ($r = 3$).

Die Zahl 37 wird einfach durch die Ziffernfolge 037 dargestellt. Für -37 berechnen wir

$$10^3 - 37 = 1000 - 37 = 963.$$

Die Berechnung von der Komplementdarstellung für eine negative Zahl ist einfacher, als es hier aussieht, auch für Leute, die einen Horror davor haben, Zahlen von Minuenden mit vielen Nullen abzuziehen, denn das Komplement kann man (im Dezimalsystem) auch so ausrechnen:

1. Bilde das „Neunerkomplement“ von der Zahl a , indem man jede Ziffer c durch $9 - c$ ersetzt (also 0 durch 9 und umgekehrt, 1 durch 8 und umgekehrt, 2 durch 7 usw.).
2. Addiere 1 zum Ergebnis.

Das funktioniert (analog auch in anderen Basen), weil es beim Abzug von 9 (oder $b - 1$) nie nötig ist, eine Eins von der nächsten Stelle zu „borgen“. Die Bildung des Neunerkomplements entspricht also den Abzug von a von der Zahl mit allen Ziffern 9, hier also von 999 (in anderen Basen handelt es sich um die Zahl mit allen Ziffern $b - 1$, und diese Zahl hat den Wert $b^r - 1$). Im zweiten Schritt addiert man 1 zum Ergebnis; also hat man gebildet

$$[(b^r - 1) - a] + 1 = b^r - a,$$

wie erforderlich.

Mit unserer Beispielzahl $37 = 037$ erhalten wir als Neunerkomplement 962, und wenn wir Eins dazu addieren erhalten wieder das Zehnerkomplement 963 von 37 als Darstellung für -37 .

Ein Vorteil von der Basis b Komplementdarstellung ist, dass es eine *eindeutige* Darstellung (mit allen Ziffern 0) für die Zahl 0 gibt, d. h., -0 und $+0$ haben die gleiche Darstellung. Wenn man 0 von b^r abzieht erhält man zwar eine Eins als $r + 1$ -te Ziffer, aber diese muss man wegwerfen, weil nur r Ziffern speicherbar sind. Das heißt, das Komplement von 0 ist wieder 0.

In dem naiven System, wo wir absolute Werte unverändert schreiben und das Vorzeichen durch einen reservierten Bit darstellen, hätte -0 tatsächlich eine andere Darstellung als $+0$, nämlich mit einem anderen Vorzeichenbit.

Das aber nur nebenbei; der *wichtigste* Vorteil des b -er Komplementsystems besteht darin, dass *signierte Zahlen genau so addiert werden können wie unsignierte*, wenn man auf Überläufe achtet und sie abfängt.

Das erklärt sich dadurch, dass die Komplementdarstellung einer negativen Zahl $-a$ sich auf sehr simple Art von dem wahren Wert von $-a$ unterscheidet: sie ist genau um b^r größer. Wenn man nun weiterrechnet und eine Addition von zwei Zahlen mit den Darstellungswerten durchführt, macht man zum wahren Wert des Ergebnisses einen Fehler von $0 \cdot b^r$ (wenn beide Summanden nichtnegativ waren), von $1 \cdot b^r$ (wenn ein Summand nichtnegativ und der andere negativ war), oder von $2 \cdot b^r$ (wenn beide Summanden negativ waren), aber diese Fehler sind im Ergebnis nicht sichtbar, weil die b^r -Stelle die $r + 1$ -te Stelle ist und außerhalb des Darstellungs- oder Registerbereichs ist.

Gerade darauf basiert die b -er Komplementdarstellung; wir machen eine negative Zahl für Berechnungen erfassbar, indem wir sie so (zu einer positiven

Zahl) verfälschen, dass der Fehler infolge der begrenzten Möglichkeiten des Rechners nicht sichtbar ist.

Trotzdem dürfen wir die Rechenergebnisse einer b -er Komplementberechnung nicht ganz ungeprüft übernehmen, denn es können schon *Überläufe* vorkommen, wie bei jeder Berechnung mit begrenzter Bitlänge.

Um zu verstehen, *wann* ein Überlauf stattgefunden hat, muss man sich erinnern, dass die b -erkomplementdarstellung nur bestimmte Ziffern, nämlich 0 oder $b-1$, an der r -ten Stelle zulässt, und dass sie, wenn sie diese Bedingung erfüllt (was sie bei $b = 2$ immer tut), trotz der „unsichtbaren“ Bits links von der r -ten Stelle eine *eindeutige* Darstellung einer bestimmten Zahl ist und wir die echte Zahl aus der b -erkomplementdarstellung wiedergewinnen können. Ein Überlauf findet genau dann statt, wenn die durch r -Bit Rechnen bestimmte Summe entweder die Ziffernbedingung verletzt (bei $b \neq 2$), oder wenn sie bei $b = 2$ das Zweierkomplement einer Zahl ist, die *nicht* die echte Summe der echten Summanden ist.

Die genaue Untersuchung der Situationen, in denen dies passiert, erfordert eine Fallunterscheidung je nach den Vorzeichen der Summanden und je nach ihrer Größe, die etwas kompliziert und nicht sehr interessant ist, weshalb wir die Details hier nicht wiedergeben.

Aber wir können sofort eine Situation nennen, in denen ein Überlauf stattgefunden haben *muss*, nämlich wenn die Summe von zwei positiven Zahlen oder von zwei negativen Zahlen nicht mehr in $r-1$ Ziffern darstellbar ist (unter Ignorierung des Vorzeichens) und deshalb keine korrekte b -Komplementdarstellung haben kann.

Diese Situation verursacht den *falschen* Übertrag in die r -te Stelle, was man bei $b \neq 2$ daran erkennt, dass die errechnete r -te Ziffer eine (verbotene) 1 oder $b-2$ ist, und bei $b = 2$ daran erkennt, dass die errechnete Summe von zwei positiven Zahlen ein negatives Ergebnis hat, oder dass die errechnete Summe von zwei negativen Zahlen ein positives Ergebnis hat (dann kann es ja nicht die b -Komplementdarstellung der wirklichen Summe sein).

Mit der oben erwähnten komplizierten Fallunterscheidung kann man mit genügend Ausdauer zeigen, dass *in keinem anderen Fall* als den eben genannten ein Überlauf eintritt. Wir wissen also genau, welche Rechenergebnisse richtig und vertrauenswürdig sind, und welche nicht, und die falschen können wir zuverlässig abfangen.

Wir illustrieren das Gesagte an einigen Beispielen. Um die Beispiele verständlicher zu machen, führen wir eine suggestive Notation ein. *Umrahmte Zahlen* wie 037 oder 963 sollen als Zehnerkomplementdarstellungen gelesen werden. Eine Gleichheit zwischen umrahmten und nicht umrahmten Zahlen soll ausdrücken, dass die umrahmte Zahl die Zehnerkomplementdarstellung der nichtumrahmten ist.

Beispiel 3.1 In diesem Beispiel wollen wir mit dreiziffrigen Dezimalzahlen im Zehnerkomplement rechnen. Von den drei erlaubten Ziffern sind die unteren beiden Nutzziffern, aber die „Hunderter“ Stelle dient nur als Vorzeichen. Diese Stelle darf deshalb nur die Werte 0 und 9 annehmen.

Wir betrachten als Summanden die Zahlen 37, 3 und 66 (deren Zehnerkomplementdarstellungen $\boxed{037}$, $\boxed{003}$ und $\boxed{066}$ sich ergeben, wenn wir eine Null bzw. Nullen voranstellen), sowie die Negativen dieser Zahlen,

$$-37 = \boxed{963}, \quad -3 = \boxed{997}, \quad \text{und} \quad -66 = \boxed{934}.$$

Hier einige Summen mit diesen Summanden, ausgeführt sowohl mit echten Zahlen im Bereich der ganzen Zahlen, wie auch mit den Zehnerkomplementen in dreistelliger Arithmetik. Wenn bei einer Summe mit den Zehnerkomplementzahlen ein Übertrag in die Tausenderstelle entsteht, schreiben wir diese vierte Ziffer außerhalb des Rahmens; im gedachten Computerregister ist nur der Inhalt des Rahmens festgehalten, und man kann leicht nachprüfen, in welchen Fällen er die Zehnerkomplementdarstellung der wahren Summe ist und in welchen Fällen er das wegen Überlaufs *nicht* ist.

$$\begin{array}{rcl}
 \begin{array}{r}
 \boxed{037} \\
 + \boxed{003} \\
 \hline
 \boxed{040}
 \end{array} & = & \begin{array}{r} 37 \\ 3 \\ \hline 40 \end{array} &
 \begin{array}{r}
 \boxed{037} \\
 + \boxed{997} \\
 \hline
 1 \boxed{034}
 \end{array} & = & \begin{array}{r} 37 \\ -3 \\ \hline 34 \end{array} &
 \begin{array}{r}
 \boxed{963} \\
 + \boxed{003} \\
 \hline
 \boxed{966}
 \end{array} & = & \begin{array}{r} -37 \\ 3 \\ \hline -34 \end{array} \\
 \\
 \begin{array}{r}
 \boxed{963} \\
 + \boxed{997} \\
 \hline
 1 \boxed{960}
 \end{array} & = & \begin{array}{r} -37 \\ -3 \\ \hline -40 \end{array} &
 \begin{array}{r}
 \boxed{963} \\
 + \boxed{066} \\
 \hline
 1 \boxed{029}
 \end{array} & = & \begin{array}{r} -37 \\ 66 \\ \hline 29 \end{array} &
 \begin{array}{r}
 \boxed{037} \\
 + \boxed{934} \\
 \hline
 \boxed{971}
 \end{array} & = & \begin{array}{r} 37 \\ -66 \\ \hline -29 \end{array} \\
 \\
 & & \begin{array}{r}
 \boxed{037} \\
 + \boxed{066} \\
 \hline
 \boxed{103}
 \end{array} & = & \begin{array}{r} 37 \\ 66 \\ \hline 103 \end{array} &
 \begin{array}{r}
 \boxed{963} \\
 + \boxed{934} \\
 \hline
 1 \boxed{897}
 \end{array} & = & \begin{array}{r} -37 \\ -66 \\ \hline -103 \end{array}
 \end{array}$$

Wir sehen, dass das b -Komplementsystem einwandfrei funktioniert, außer bei den Berechnungen in der letzten Zeile, wo die Summanden das gleiche Vorzeichen haben und der Absolutbetrag ihrer Summe nicht mehr mit zwei Dezimalziffern darstellbar ist. Dann haben die Zehnerkomplementsummen unmögliche Hunderterziffern, woran der Überlauf sich zu erkennen gibt.

Beispiel 3.2 In einem weiteren Beispiel wollen wir mit 4-Bit Binärzahlen im Zweierkomplement rechnen. Von den vier erlaubten Ziffern sind die unteren drei Nutzziffern, aber die „Achter“ Stelle dient nur als Vorzeichen.

Wir betrachten als Summanden die Zahlen

$$5 = 101_2, \quad 2 = 10_2 \quad \text{und} \quad 7 = 111_2$$

(deren Zweierkomplementdarstellungen $\boxed{0101}$, $\boxed{0010}$ und $\boxed{0111}$ sich ergeben, wenn wir eine Null bzw. Nullen voranstellen), sowie die Negativen dieser Zahlen,

$$-5 = \boxed{1011}, \quad -2 = \boxed{1110}, \quad \text{und} \quad -7 = \boxed{1001}.$$

Hier einige Summen mit diesen Summanden, ausgeführt im Dezimalsystem mit den echten Zahlen und binär mit den Zweierkomplementen.

$$\begin{array}{rcl}
 \begin{array}{r} \boxed{0101} \\ + \boxed{0010} \\ \hline \end{array} & = & \begin{array}{r} 5 \\ 2 \\ \hline \end{array} \\
 \begin{array}{r} \boxed{0111} \\ \hline \end{array} & = & 7 \\
 \\
 \begin{array}{r} \boxed{1011} \\ + \boxed{1110} \\ \hline \end{array} & = & \begin{array}{r} -5 \\ -2 \\ \hline \end{array} \\
 1 \begin{array}{r} \boxed{1001} \\ \hline \end{array} & = & -7 \\
 \\
 \begin{array}{r} \boxed{0101} \\ + \boxed{0111} \\ \hline \end{array} & = & \begin{array}{r} 5 \\ 7 \\ \hline \end{array} \\
 \begin{array}{r} \boxed{1100} \\ \hline \end{array} & \neq & 12 \\
 \\
 \begin{array}{r} \boxed{0101} \\ + \boxed{1110} \\ \hline \end{array} & = & \begin{array}{r} 5 \\ -2 \\ \hline \end{array} \\
 1 \begin{array}{r} \boxed{0100} \\ \hline \end{array} & \neq & -12 \\
 \\
 \begin{array}{r} \boxed{0101} \\ + \boxed{1001} \\ \hline \end{array} & = & \begin{array}{r} 5 \\ -7 \\ \hline \end{array} \\
 1 \begin{array}{r} \boxed{0100} \\ \hline \end{array} & \neq & -12
 \end{array}$$

Diesmal sind die wegen Überlaufs ungültigen Summen in der letzten Zeile zwar echte Zweierkomplemente, aber von Zahlen mit dem falschen Vorzeichen, nämlich von -4 (statt 12) und von 4 (statt -12).

Es gibt ein zweites Darstellungssystem für signierte Zahlen, das ähnliche Vorteile hat und nur unwesentlich komplizierter ist (dieser kleine Preis bringt aber auch einen kleinen Vorteil). Wir diskutieren dieses System nur für *binär* dargestellte Zahlen, weil die weitere Diskussion zeigen wird, dass die Anwendung in anderen Basen zu viel Rechenarbeit erfordern und sich deshalb im Vergleich zum Komplementsystem nicht lohnen würde.

Die Idee hinter diesem neuen Darstellungssystem besteht darin, den für *unsigned* Zahlen gültigen Wertebereich so zu verschieben, dass er neben positiven Zahlen auch einige negative Zahlen abdeckt, wobei man versucht, in etwa die gleiche Anzahl von negativen wie positiven Zahlen zu umfassen. Das erreicht man, indem man zu jeder echten Zahl, die man erfassen will, ein festes

Bias, d. h., einen festen *Zuschlag* addiert, um die Zahl als eine nichtnegative Zahl mit der verfügbaren Anzahl r von Bits darstellen zu können.

Um die gewünschte Symmetrie zu erhalten (gleich viele positive wie negative Zahlen darstellbar) sollte der Zuschlag etwa halb so groß sein, wie die Anzahl der mit r Bits darstellbaren Zahlen. Da man mit r Bits 2^r verschiedene Zahlen schreiben kann, wählt man einen Zuschlag von 2^{r-1} (aber bei unserer wichtigsten späteren Anwendung für dieses System wird der Zuschlag $2^{r-1} - 1$ sein, weil dort die Extremwerte eine spezielle Bedeutung haben und nicht zu den regelmäßigen Zahlendarstellungen gezählt werden werden).

Dieses System mit dem Bias oder Zuschlag von $d \approx 2^{r-1}$ nennt sich das **Bias d** oder **Exzess d** System.

Zum Beispiel, für 8-Bit Zahlen würden wir einen Bias von $2^7 = 128$ wählen, und im Exzess 128 System hätte die kleinste darstellbare Zahl die Darstellung 0 und den Wert -128 , während die größte darstellbare Zahl die Darstellung $1111111_2 = 255$ und den Wert $255 - 128 = 127$ hätte.

Die Darstellung von -10 in diesem System wäre $118 = 01110110_2$ und die Darstellung von 37 wäre $165 = 10100101_2$.

Wie rechnet man im Exzess 2^{r-1} System? Da jede Zahlendarstellung um 2^{r-1} größer als die dargestellte Zahl ist, ist die Summe von zwei Zahlendarstellungen um 2^r größer als die echte Summe, also um 2^{r-1} größer als die *Darstellung* der echten Summe. Um die richtige Darstellung zu erhalten, muss man nur 2^{r-1} von der normal gebildeten Summe abziehen, d. h., man muss eine Eins an der höchsten Bitstelle abziehen.

Wenn man Überträge aus dem höchsten Bit ignoriert (was man ohnehin macht, da man diesen Übertrag in der zulässigen Anzahl von Bits nicht darstellen kann), bewirkt der Abzug von 2^{r-1} nur die Umkehrung des höchsten Bits. Also haben wir folgende Additionsregel für Exzess 2^{r-1} Zahlen:

- Addiere „normal“ wie für unsigned r -Bit Zahlen; anschließend kehre das höchste Bit um.

Wenn wir diese Regel für die Beispielzahlen -10 und 37 anwenden, erhalten wir bei der binären unsigned Addition zunächst:

$$\begin{array}{r} 01110110 \\ +10100101 \\ \hline 1\ 00011011, \end{array}$$

wobei wir den Übertrag ignorieren und nur die acht Bits 00011011 behalten. Hier ist anschließend das höchste Bit umzukehren. Das Ergebnis $10011011_2 = 155$ ist die Exzess 128 Darstellung von $27 = -10 + 37$.

Auch im Exzess 2^{r-1} System ist ein Überlauf möglich, nämlich dann, wenn die Exzess 2^{r-1} Darstellung der wirklichen Summe (die sich durch Subtraktion einer 1 an der r -ten Stelle vom Ergebnis der vorangehenden Addition ergibt) nicht in dem Darstellungsbereich liegt, also entweder negativ oder $\geq 2^r$ ist.

Das passiert in zwei Fällen: wenn *vor* der Umkehrung des höchsten Bits dieses Bit *und* der Übertrag aus der höchsten Stelle beide 1 sind, dann würde die Subtraktion diesen Übertrag nicht löschen und das Ergebnis wäre $\geq 2^r$; wenn vor der Umkehrung des höchsten Bits dieses Bit und der Übertrag aus der höchsten Stelle beide 0 waren, dann würde die Subtraktion ein negatives Ergebnis liefern. In allen anderen Fällen liefert die Subtraktion ein Ergebnis zwischen 0 und $2^r - 1$ und das Umkehren des höchsten Bits liefert das gleiche Ergebnis; es gibt dann keinen Überlauf und das Ergebnis ist gültig.

Also kann man die Bedingung für einen Überlauf ganz knapp formulieren:

- Genau dann gibt es einen Überlauf bei der Exzess 2^{r-1} Addition, wenn das höchste Bit nach der unsignierten Addition gleich dem Übertrag aus der höchsten Stelle ist.

Die Exzess 2^{r-1} Darstellung unterscheidet sich nur geringfügig von der Zweierkomplementdarstellung. Die Exzess 2^{r-1} Darstellung ist *immer* um 2^{r-1} größer als der richtige Zahlenwert; die Zweierkomplementdarstellung benutzt den gleichen Darstellungsbereich aber ist *gleich* der echten Zahl, wenn sie nichtnegativ ist, und ist um 2^r (statt 2^{r-1}) größer als die echte Zahl, wenn sie negativ ist.

Beide Darstellungen unterscheiden sich also um $\pm 2^{r-1}$, d. h., um 1 an der höchsten Stelle. In anderen Worten: die unteren $r - 1$ Bits sind bei beiden Darstellungen gleich und die r -ten Bits sind komplementär.

Wegen der großen Ähnlichkeit zwischen diesen beiden Systemen und der untergeordneten Bedeutung des Exzess 2^{r-1} Systems verzichten wir auf weitere Rechenbeispiele.

Das Addieren im Exzess 2^{r-1} System erfordert eine zusätzliche Operation: das Umkehren des höchsten Bits. Dafür behält dieses System aber die Ordnungsverhältnisse zwischen positiven und negativen Zahlen bei; im Exzess 2^{r-1} System hat immer die algebraisch größere Zahl die größere Darstellung. Im Zweierkomplementsystem wird die Größerkleinerrelation zwischen Zahlenpaaren, in denen mindestens eine Zahl negativ ist, immer umgekehrt, was es erheblich erschwert, Größenvergleiche zu machen.

Das ist der Grund für die Bevorzugung der Exzess 2^{r-1} Darstellung in einer ganz bestimmten Situation, in der solche Größenvergleiche oft vorzunehmen sind, nämlich bei der Darstellung von „Kommazahlen“ in der naturwissenschaftlichen Schreibweise mit Exponenten. Dazu kommen wir bald.

Neben dem Rechnen mit signierten ganzen Zahlen ist auch das Rechnen mit „Bruchzahlen“, also mit rationalen und reellen Zahlen, wichtig. Natürlich können wir nicht mit allen reellen Zahlen rechnen, sondern nur mit den endlich vielen, die im endlichen Speicherraum eines Rechners darstellbar sind, aber wir werden versuchen, es so einzurichten, dass die darstellbaren Zahlen einen möglichst großen Zahlenbereich abdecken.

Die einfachste Darstellungsform für nichtganze Zahlen ist das **Festkommaformat**, bei denen Zahlen bearbeitet werden, die eine feste Anzahl von Stellen hinter dem Komma haben. Solche Zahlen kann man so behandeln, als wären sie ganz, d. h., als wäre das Komma hinter der letzten der endlich vielen Stellen, und man muss nur eine (sogar von Berechnung zu Berechnung variierende) Konvention darüber treffen, wo die wirkliche gedachte Lage des Kommas sich befindet. Notieren muss man diese Lage beim Speichern der Zahlen nicht, solange man (zum Beispiel durch Software) dafür sorgt, dass bei der Ausgabe von Zahlen in Druckformat das Komma an die richtige Stelle geschrieben wird und dafür, dass bei Multiplikationen und Divisionen die Ergebnisse so verschoben und ausgerichtet werden, dass das Komma an der gleichen gedachten Stelle steht, wie bei anderen Zahlen, die man gespeichert hat.

Hier handelt es sich um einfache technische Details, die keine grundlegenden Probleme aufwerfen, und die wir deshalb nicht weiter diskutieren wollen.

Festkommaarithmetik ist geeignet für Berechnungen mit Beträgen einer beschränkten Größenordnung und einer festen Nachkommagenauigkeit, wie zum Beispiel für Geldbeträge und dergleichen.

Für wissenschaftliche Berechnungen taugt sie aber nicht, weil dort sehr sehr große und sehr sehr kleine Zahlen vorkommen können, die sich um viele Größenordnungen unterscheiden können, und weil es dort auch nicht auf eine bestimmte Anzahl von Nachkommastellen ankommt, sondern mehr um die *relative* Genauigkeit von Zahlen ankommt, d. h., um die Anzahl von **signifikanten Stellen** einer Zahl, unabhängig von der wirklichen Lage des Kommas.

Für diesen Zweck eignet sich ein Zahlenformat, der sich an der „naturwissenschaftlichen Schreibweise“ von Zahlen orientiert, im Format

$$\pm \text{Mantisse} \times \text{Basis}^{\text{Exponent}}. \quad (3.2)$$

Hierbei ist die **Mantisse** eine Kommazahl, die sich normalerweise in einem bestimmten Zahlenbereich in der Nähe von 1 befinden soll, und die wahre Größenordnung durch den Exponentialterm geregelt wird. Die Mantisse dient zur Präzisierung der „signifikanten“ Ziffern der Zahl.

Die auf diese Schreibweise basierende Darstellung von Zahlen im Rechner nennt sich, unabhängig von den näheren Details, eine **Gleitkommadarstellung**.

lung (auf englisch *floating point*), weil durch Veränderung des Exponenten die genaue Kommalage sehr einfach verändert werden kann und auf die Bedürfnisse der jeweiligen Aufgabe angepasst werden kann.

Vor etwa 1980 hat jeder Rechnerhersteller sein eigenes Gleitkommaformat entwickelt und in seinen Rechnern implementiert, und obwohl alle Gleitkommaformate auf die gleiche Grundidee basierten, variierten sie stark in den Details und waren nicht direkt auf andere Rechner übertragbar.

Weil es auch nur endlich viele Gleitkommazahlen auf einem endlichen Rechner geben kann, erscheinen bei Gleitkommaberechnungen ähnliche Probleme wie beim Rechnen mit signierten ganzen Zahlen — nicht jedes Rechenergebnis lässt sich im gewählten Schema darstellen und aus diesem Grund sind Entscheidungen nötig, wie man mit „ungültigen“ oder nicht darstellbaren Ergebnissen umgehen soll. Weil die Gleitkommadarstellung komplizierter ist als die Systeme zur Darstellung signierter Zahlen, gibt es hier mehr „Ausnahmeergebnisse“ (zum Beispiel können neben Überläufen auch Unterläufe vorkommen, bei denen die Ergebnisse zu *klein* für die Darstellungsart sind), und jeder Hersteller hat eigene Regeln für die Behandlung solcher Fälle verwendet, die weder mit den Regeln anderer Hersteller verträglich sein mussten noch unbedingt zu richtigen und sinnvollen Ergebnissen geführt haben. Die Gleitkommarechnung birgt viele subtile Probleme, deren richtige Behandlung nicht ganz einfach ist.

Um dem Wirrwarr ein Ende zu bereiten hat das IEEE (Institute of Electrical and Electronics Engineers, der internationale Berufsverband der Elektroingenieure) einen Ausschuss gebildet, der einen neuen Standard für Gleitkommadarstellungen entwickeln sollte. Der Antrieb dazu kam von der Firma Intel, die einen möglichst vollkommenen Gleitkommaprozessor bauen wollte und dazu den Rat von Experten suchte — der Experte, der maßgeblich an der Entwicklung des Standards beteiligt war und dessen Entwurf schließlich weitgehend übernommen wurde und 1985 als IEEE Standard 754 veröffentlicht wurde, hieß William Kahan (Professor der Mathematik und Informatik an der University of California in Berkeley).

Die IEEE Formate unterscheiden sich wesentlich von früheren Gleitkommasystemen und haben einige zusätzliche Fähigkeiten; der Standard hat sich inzwischen weitgehend durchgesetzt und wird von den meisten heutigen Rechnern befolgt.

Um einen Vergleich zu ermöglichen, wollen wir sowohl den IEEE Standard wie auch ein „klassisches“ Gleitkommaformat, nämlich das der IBM 360 Rechner und ihren Nachfolgern, näher beschreiben.

Allen gebräuchlichen Gleitkommaformaten sind folgende Einzelheiten gemeinsam:

- Sie haben ein Vorzeichenbit, der für nichtnegative Zahlen auf 0 und für negative Zahlen auf 1 gesetzt wird.
- Sie haben ein Bitfeld einer bestimmten Länge für die Darstellung des Exponenten im Ausdruck (3.2), und der Exponent wird in diesem Feld als eine Exzess n signierte Zahl gespeichert, wo der Bias n so gewählt wird, dass etwa gleich viele positive wie negative Exponenten darstellbar sind.
- Sie haben ein Bitfeld einer bestimmten Länge für die Darstellung der Mantisse, wobei die Mantisse in der Regel **normalisiert** sein soll auf eine bestimmte Größenordnung angrenzend an 1, die eine Potenz der Basis b überspannt (also von $1/b$ bis 1 oder von 1 bis b reicht).
- Die meisten Gleitkommasysteme bieten mehrere Genauigkeiten an, die sich in ihrer Wirkung in den darstellbaren Größenordnungen und in der Anzahl der speicherbaren signifikanten Stellen unterscheiden, während sie sich in ihrer rechnerinternen Gestalt durch die Anzahl der für den Exponenten und für die Mantisse reservierten Bits unterscheiden.

Fast immer gibt es eine Darstellungsform **einfacher Genauigkeit**, eine in der Regel doppelt so lange Darstellungsform **doppelter Genauigkeit**, und eine noch längere Form **erweiterter Genauigkeit**, die innerhalb der Gleitkommaeinheit verwendet wird, um Rundungsfehler zu vermeiden.

Trotz dieser vielen Gemeinsamkeiten gibt es noch einige wesentliche Details, die zwischen verschiedenen Ansätzen stark variieren können. Manche dieser Details sind in den gespeicherten Gleitkommazahlen sichtbar: zum Beispiel, wie eine normalisierte Mantisse aussieht und gespeichert wird, welcher Bias bei der Darstellung des Exponenten verwendet wird, welche Größenordnung genau eine normalisierte Zahl hat (weil dies sich auf den Wert des Exponenten für eine bestimmte Zahl auswirkt). Andere Details, die zwischen den Formaten variieren, betreffen die Behandlung und die Reaktion auf Ausnahmeergebnisse, also Ergebnisse wie Überläufe und Unterläufe, die zu groß oder zu klein sind oder aus anderen Gründen nicht im normalen darstellbaren Bereich liegen. Solche „bösen“ Rechenergebnisse können entweder zu einem Programmabbruch wegen Ungültigkeit führen, oder durch „nahe“ gültige Ergebnisse ersetzt oder „gerundet“ werden (ein Unterlauf zum Beispiel durch 0), oder durch ein Sonderformat doch dargestellt werden, in der Regel dann mit verminderter Genauigkeit. Gerade das IEEE Format bietet hier mehr Möglichkeiten, als die klassischen Gleitkommaformate.

Für zwei wichtige Gleitkommasysteme wollen wir jetzt diese Details genau beschreiben, zunächst als klassische Variante das Gleitkommaformat der in der zweiten Hälfte der 60er Jahren eingeführten IBM 360 Rechnerfamilie. Dieses Gleitkommaformat wird sogar von heutigen IBM Großrechnern, den späten Nachfolgern der 360 Familie, noch unterstützt (obwohl die neuen Rechner auch das IEEE Format verwenden können).

Die Besonderheit der IBM 360 besteht darin, dass die Basis für den Potenzterm in (3.2) nicht 2 sondern 16 ist. Es gibt drei Präzisionsstufen: **short**, **long** und **extended**. Alle drei Stufen beginnen mit einem Vorzeichenbit und dem Exponentenfeld, das in allen Genauigkeitsstufen 7 Bit breit ist und den Exponenten im Exzess 64 Format enthält. Als letztes folgt das Mantissenfeld.

Die drei Stufen unterscheiden sich nur in der Länge der Mantisse: sie beträgt 24 Bits im Format „short“, 56 Bits im Format „long“ und 112 Bits im Format „extended“.

Eine IBM Gleitkommazahl heißt **normalisiert**, wenn ihre Mantisse f im Bereich

$$\frac{1}{16} \leq f < 1$$

liegt. Das Standardgleitkommaformat hat normalisierte Mantissen, bei denen alle Nachkommabits in das Mantissenfeld zu schreiben sind. Die kleinste normalisierte Mantisse hat den Wert $1/16$ und das kleinste normalisierte Mantissenfeld hat 0001 als die ersten 4 Bits und danach nur Nullen. Eine normalisierte Zahl in den IBM Gleitkommaformaten ist daran zu erkennen, dass entweder *alle* Mantissenbits 0 sind (dann ist die dargestellte Zahl 0), oder dass die ersten 4 Bits des Mantissenfeldes nicht alle 0 sind.

Nichtnormalisierte Mantissen können normalisiert werden, indem das Komma in Einheiten von 4 Bits so oft nach links oder nach rechts geschoben wird, bis das höchste nichtverschwindende Bit sich in dem ersten 4-Bit Feld in der Mantisse befindet. Für jede Bewegung des Kommas um 4 Stellen nach rechts ist der Exponent um 1 zu vermindern, und für jede Bewegung des Kommas um 4 Stellen nach links ist der Exponent um 1 zu erhöhen.

Der größte mögliche Wert des Exponentenfeldes ist 1111111_2 , der die Zahl $127 - 64 = +63$ darstellt. Der kleinste Wert des Exponentenfeldes ist 0 und stellt die Zahl -64 dar.

Somit ist die größte Zahl, die im IBM Gleitkommaformat dargestellt werden kann, knapp unter $1 \times 16^{63} = 7,237 \times 10^{75}$. Die kleinste normalisiert darstellbare positive Zahl ist $\frac{1}{16} \times 16^{-64} = 16^{-65} = 5,3976 \times 10^{-79}$.

Als Beispiel schreiben wir die Zahl -37 und die Zahl π im IBM Gleitkommaformat *short*.

Als Binärzahl ist $-37 = -100101_2$ und als nicht normalisierte Gleitkommazahl mit Exponentenbasis 16 ist also $-37 = -100101,0 \times 16^0$. Wir müssen

das Komma nach links verschieben, aber immer jeweils um vier Stellen!, bis es vor dem ersten Bit von der Zahl steht. Dazu sind zwei Verschiebungen nötig, die wir kompensieren müssen, indem wir den Exponenten um 2 erhöhen. Als normalisierte Binärzahl in der naturwissenschaftlichen Notation mit Basis 16 ist -37 also gleich

$$-0,0010\,0101 \times 16^2.$$

Diese Daten müssen wir in das IBM Gleitkommaformat übertragen. Der Exponent 2 ist als Exzess 64 Zahl zu schreiben, also als $66 = 1000010_2$. Das Vorzeichenbit ist 1. Alles zusammen ergibt für -37 die *short* Bitfolge

$$1\,100\,0010\,0010\,0101\,0000\,0000\,0000\,0000 = C2250000_{16}.$$

Die Zahl π schreibt sich als Hexadezimalbruch mit 8 Stellen hinter dem Komma wie folgt:

$$\pi = 3,243F6A89.$$

Für das IBM Gleitkommaformat *short* müssen wir sie normalisieren und sechs Hexadezimalstellen hinter dem Komma behalten, d. h., wir schreiben sie als

$$\pi = 0,3243F7_{16} \times 16^1.$$

Die Gleitkommadarstellung (mit dem Exponenten im Exzess 64 Format) ist dann

$$413243F7,$$

wenn wir die Bitfolge hexadezimal schreiben.

Obwohl noch in Gebrauch, ist das IBM Gleitkommaformat nicht mehr von primärer Bedeutung, da das IEEE Format inzwischen wirklich zu einem Standard geworden ist und von fast allen Gleitkommaanwendungen verwendet wird. In der Beschreibung, die jetzt folgt, wird deutlich, welche Vorteile der IEEE Standard hat.

Das IEEE Format implementiert neben normalisierten Gleitkommazahlen noch einige weitere rechnerische Größen, die als Darstellung für Ausnahmeergebnisse dienen und eine einheitliche Behandlung von Ausnahmesituationen in allen Plattformen, die sich an den IEEE Standard halten, ermöglichen.

Folgende Zahlarten sind im IEEE Standard definiert:

- normalisierte Gleitkommazahlen;
- denormalisierte (also entnormalisierte oder nicht normalisierte) Gleitkommazahlen;
- 0;

- $\pm\infty$, d. h., plus oder minus Unendlich;
- **NaN** oder „Not a Number“; das sind ungültige oder unbestimmte Ergebnisse, und es lässt sich zwischen diesen Arten sogar unterscheiden!

Die Notwendigkeit, so viele Zahlenarten möglichst effizient in der vorgegebenen Bitzahl darzustellen, erklärt viele der Besonderheiten des IEEE Schemas.

Wie beim IBM System haben IEEE Gleitkommazahlen ein Vorzeichenbit, gefolgt durch ein Exponentenfeld mit Exponenten in einem Exzess n Format, gefolgt durch ein Mantissenfeld, und es gibt drei Hauptformate mit folgenden Feldgrößen und Daten:

Genauigkeit	Vorzeichen	Exponent	Exzess	Mantisse	Gesamtlänge
einfach	1 Bit	8 Bits	127	23 Bits	32 Bits
doppelt	1 Bit	11 Bits	1023	52 Bits	64 Bits
erweitert	1 Bit	15 Bits	16383	64 Bits	80 Bits

Tabelle 3.1: Die IEEE Gleitkommaformate

Es fällt auf, dass bei einem r -Bit langen Exponentenfeld der Bias nicht 2^{r-1} beträgt, sondern $2^{r-1} - 1$. Dafür gibt es einen Grund: die beiden Extremwerte im Exponentenfeld, nämlich alle Bits 0 oder alle Bits 1, sind ausgeschlossen für die Darstellung normalisierter Zahlen, so dass die Anzahl der gültigen Exponentenwerte für normale Zahlen nicht 2^r ist, sondern $2^r - 2$; der Bias ist genau die Hälfte von *diesem* Wert, mit der Wirkung, dass, wie man es wünscht, genau so viele negative Exponenten darstellbar sind, wie nichtnegative.

Die verbotenen Exponentenwerte haben trotzdem eine Funktion — sie sind reserviert für die Darstellung der Ausnahmezahlenarten, die zur Reichweite und Leistungsfähigkeit des IEEE Standards wesentlich beitragen.

Wie wollen jetzt sowohl den Sinn wie auch die rechnerinterne Gestalt der verschiedenen Zahlenarten genau beschreiben.

Normale Rechenergebnisse werden immer als **normalisierte Gleitkommazahlen** geschrieben, sofern das mit den verfügbaren Feldbreiten und mit den erlaubten Exponentenwerten möglich ist, und deshalb ist dies auch die wichtigste Zahlenart. Was aber eine normalisierte Zahl ausmacht, ist im IEEE Standard anders definiert, als in den klassischen Gleitkommasystemen wie das der IBM 360.

Im IEEE Format sind die Exponenten immer als Exponenten zur Basis 2 zu verstehen. Eine Zahl in der naturwissenschaftlichen Schreibweise

$$\pm f \times 2^d$$

heißt **normalisiert** im Sinne des IEEE Standards genau dann, wenn

$$1 \leq f < 2.$$

Wenn man eine normalisierte Zahl als „Binärkommazahl“ schreibt, dann hat sie immer eine 1 unmittelbar *vor* dem Komma, und eine beliebige Bitfolge hinter dem Komma. In anderen Worten, sie hat die Gestalt

$$1,m \tag{3.3}$$

mit einer gewissen Bitfolge m .

Weil die 1 vor dem Komma bei normalisierten Zahlen *immer* vorhanden ist, ist es nicht nötig, sie wirklich hinzuschreiben, da sie nicht zur Unterscheidung von normalisierten Zahlen beiträgt. Dadurch, dass man diese 1 nicht im Rechner speichert, spart man eine Bitstelle, in die man ein weiteres Bit *hinten* in der Bruchentwicklung speichern kann; so kann man mit s Bitstellen eine Zahl mit $s + 1$ signifikanten Stellen (deren erste immer eine 1 ist) eindeutig festhalten.

Genau das macht der IEEE Standard. In das Mantissenfeld einer IEEE normalisierten Gleitkommazahl werden nur die passende Anzahl von Nachkommastellen, also die Bitfolge m hineingeschrieben. Diese Bitfolge werden wir als das **Mantissenfeld** bezeichnen, aber sie ist nicht die wirkliche Mantisse. Die Mantisse ergibt sich daraus als der Binärbruch $1,m$. Aus verschiedenen Gründen, aber primär weil die Gleitkommamantisse nicht der klassischen Bedeutung als die Mantisse eines Logarithmus entspricht, wird sie im IEEE Standard nicht „Mantisse“, sondern **Signifikant** genannt. Man beachte, dass der Signifikant nicht nur aus den Nachkommastellen m im Mantissenfeld besteht, sondern auch die „versteckte“ Eins und Komma davor beinhaltet.

Zahlen, die nicht normalisiert vorgegeben sind, und die nicht Null sind, muss man normalisieren, indem man das Komma direkt *hinter* die erste 1 in der Bitfolge schiebt, und die Kommaverschiebung muss man kompensieren durch Erhöhung des Exponenten (für eine Linksverschiebung des Kommas) oder Verminderung des Exponenten (für eine Rechtsverschiebung des Kommas) um Eins für jede bei der Verschiebung übersprungene Bitstelle. Hier muss man, anders als beim IBM Format, den Exponenten für jede *einzelne* Bitstelle um Eins verändern, weil die Basis für den Potenzterm jetzt 2 ist und nicht 16.

Eine Gleitkommazahl lässt sich genau dann als normalisierte Gleitkommazahl im IEEE Format abspeichern, wenn nach der Kommaverschiebung der justierte Exponent sich im zuständigen Exzess n Format mit der zulässigen Anzahl von Bits schreiben lässt; ferner darf diese Exzess n -Darstellung e weder 0 noch der maximalen Wert sein (bei dem alle Bits 1 sind).

Die Zahl 0 kann nicht normalisiert werden, weil sie gar keine Einsbits hat. Sie wird dargestellt durch ein Wort (der zu der Genauigkeit passenden Länge), in dem alle Bits bis auf das Vorzeichenbit null sind. Das Vorzeichenbit darf 0 oder 1 sein; d. h., es gibt sowohl eine $+0$ wie auch eine -0 .

Es gibt auch andere Zahlen als Null, die nicht normalisierbar sind, und solche Zahlen können als Rechenergebnisse auftreten, auch wenn die Rechenoperanden normalisiert waren. Zwei derartige Situationen können eintreten: ein Rechenergebnis kann im Absolutbetrag (d. h., , unter Ignorierung des Vorzeichens) zu *groß* für das gewählte Gleitkommaformat sein (dann sprechen wir von einem **Überlauf**), oder es kann zwar ungleich 0 aber im Absolutbetrag zu *klein* sein, um normalisiert zu werden (dann sprechen wir von einem **Unterlauf**).

Auslöser für solche Fälle ist nicht die Größe oder die Bitzahl der Mantisse oder des Signifikanten, denn diese Bitzahl bestimmt nur die Genauigkeit, mit der der Zahlenwert bekannt ist. Zu lange Mantissen können einfach auf die Mantissenfeldlänge gestützt werden, ohne den Zahlenwert wesentlich zu verändern. Zu kurze Mantissen können rechts mit Nullen aufgefüllt werden.

Vielmehr passieren Überläufe und Unterläufe dann, wenn der *Exponent* zu groß oder zu klein wird und nicht mehr im gültigen Bereich liegt. Früher wurde auf Überläufe mit einem Berechnungsstopp und Abbruch des Programms reagiert, und Unterläufe verursachten entweder auch einen Programmabbruch oder eine Rundung des Ergebnisses auf 0, bevor die Berechnung fortgesetzt wurde. Der IEEE Standard sieht eine subtilere und flexiblere Behandlung beider Fälle vor.

Bei einem Unterlauf kann das Ergebnis (welches dann nicht 0 ist!) zwar nicht normalisiert werden, aber der Zahlenwert kann in den meisten Fällen trotzdem dargestellt werden, wenn man auf die Normalisierung verzichtet.

Der IEEE Standard sieht für Unterläufe vor, dass die darzustellende (und zu kleine) Zahl so bezüglich des Kommas verschoben wird, dass der Exponent d die Exzess n Darstellung $e = 1$ erhält (und somit den wahren Wert $-n + 1$ hat; für die einfache Genauigkeit ist dieser Wert -126 und für die doppelte Genauigkeit ist er -1022). Das Komma muss sich dann irgendwo *links* vom ersten Einsbit befinden, weil sonst die Zahl ja hätte normalisiert werden können und kein Unterlauf vorgelegen hätte.

Zur Darstellung dieser Zahl nach dem IEEE Standard werden die nach der beschriebenen Vorbehandlung sich ergebenden Nachkommabits in das Mantissenfeld geschrieben, das Vorzeichen wird auf den richtigen Wert 0 oder 1 gesetzt und *das Exponentenfeld wird mit Nullen gefüllt*, was für normalisierte Zahlen ja nicht erlaubt ist.

Wenn die darzustellende Zahl sehr klein war, kann es nach der Kommaverschiebung sein, dass nur Nullbits in das Mantissenfeld geschrieben werden. In

diesem Fall haben wir den wahren Wert der Unterlaufszahl tatsächlich ganz verloren und die Zahl effektiv zu ± 0 gerundet, denn das ist die Gleitkommazahl, die sich ergeben hat.

Aber es gibt einen ganzen Bereich von Größenordnungen unterhalb der kleinsten normalisierbaren Zahl, in dem nach der Kommaverschiebung noch einige Einsbits so nah am Komma liegen, dass sie im Mantissenfeld festgehalten werden. Wir haben dann eine Gleitkommazahl erzeugt, mit den Eigenschaften:

- das Vorzeichenbit ist beliebig;
- das Exponentenfeld enthält nur Nullen;
- das Mantissenfeld ist nicht 0!

Gleitkommazahlen dieser Gestalt heißen **denormalisierte Zahlen**. Sie schaffen es, wenigstens etwas auszusagen über Zahlen, die zu klein sind für die normalisierte Darstellung, aber die erweiterte Darstellbarkeit ist nicht ganz kostenlos, denn die Anzahl der signifikanten Stellen ist auf jeden Fall kleiner, als bei normalisierten Zahlen, und wird immer weniger, desto kleiner die Zahl wird (bis sie nur noch als 0 darstellbar ist).

Die gerade aufgezählten Eigenschaften charakterisieren denormalisierte Zahlen und unterscheiden sie von normalisierte Zahlen (bei denen das Exponentenfeld nicht 0 sein darf) und von der Null (bei der das Mantissenfeld nur Nullen enthält).

Wichtig! Für denormalisierte Zahlen gelten andere Leseregeln, als für normalisierte Zahlen! Bei denormalisierten Zahlen ist keine 1 vor dem Komma zu ergänzen, sondern eine 0. Somit enthält das Mantissenfeld den wirklichen Wert der Mantisse. Ferner ist der wahre Wert des Exponenten nicht -127 oder -1023 , wie man es nach den allgemeinen Regeln bei Exponentenfeld 0 eigentlich erwarten würde, sondern um Eins höher, also -126 bei Zahlen einfacher Genauigkeit und -1022 bei Zahlen doppelter Genauigkeit.

Das hat einen Sinn. Für die einfache Genauigkeit, zum Beispiel, hat die kleinste normalisierte Zahl den Wert $2^{1-127} = 2^{-126}$. Nach der tatsächlich verwendeten Regel hat die größte denormalisierte Zahl den Wert $0,9999998808 \times 2^{-126}$, und das ist fast genau so groß. Würde man den Exponenten 0 als eine Darstellung von -127 bewerten, so wäre jede denormalisierte Zahl kleiner als 2^{-127} und keine Zahl zwischen 2^{-127} und 2^{-126} wäre überhaupt darstellbar.

Für Überläufe hätte man eine ähnliche fein abgestufte Darstellungsregel einführen können, aber man hat darauf verzichtet und etwas viel Sinnvolleres gemacht. Die *größte* normalisiert darstellbare Zahl ist $3,4 \times 10^{38}$ in einfacher Genauigkeit und etwa $1,8 \times 10^{308}$ in doppelter Genauigkeit; zumindest die

letzte Zahl übersteigt alles, was in einer naturwissenschaftlichen Anwendung vorkommen könnte.

Statt noch größere Zahlen als diese darstellbar zu machen, hat man es deshalb vorgezogen, die „Zahl“ Unendlich im Standard zu erfassen, denn diese Zahl geht doch oft in theoretische Überlegungen ein und ist auch für mathematische Berechnungen nützlich, wenn man richtig damit umgeht.

Die im IEEE Standard vorgesehene Darstellung von Unendlich (oder ∞) hat alle Exponentenbits Eins und alle Mantissenbits Null; nur das Vorzeichenbit kann 0 oder 1 sein, so dass sowohl $+\infty$ und $-\infty$ darstellbar sind.

Damit sind aber nur zwei Bitmuster mit dem maximalen Exponentenwert belegt; was macht man mit den anderen?

Der Standard schreibt nicht nur eine Zahlendarstellung vor, sondern auch Rechenregeln für den Umgang mit den dargestellten Zahlen. Auch für die Zahl ∞ gibt es einige sinnvolle Rechenregeln. Zum Beispiel:

$$\begin{aligned}\infty + \infty &= \infty \\ \infty \cdot \infty &= \infty \\ \infty - \text{beliebige endliche Zahl} &= \infty \\ \frac{1}{\infty} &= 0\end{aligned}$$

und so weiter.

Aber nicht jede Berechnung mit ∞ hat ein eindeutiges Ergebnis. Denn ∞ ist eigentlich nur eine Umschreibung für eine sehr große Zahl. Die Differenz zweier sehr großer Zahlen kann sehr groß oder sehr klein sein oder beliebige endliche Werte annehmen, so dass es keine eindeutige sinnvolle Bedeutung für $\infty - \infty$ geben kann. Solche Ausdrücke können bei Berechnungen vorkommen, aber sie haben ein *unbestimmtes* Ergebnis.

Andere Ausdrücke mit unbestimmten Ergebnis wären

$$\frac{\infty}{\infty}, \quad \frac{0}{0} \quad \text{oder} \quad 0 \cdot \infty.$$

Neben Berechnungen mit unbestimmten Ergebnissen gibt es auch Berechnungen, die *kein* Ergebnis haben, d. h., die schlicht nicht erlaubt sind. Dazu gehören zum Beispiel die Bildung der Quadratwurzel oder des Logarithmus einer negativen Zahl (wenn man mit reellen Zahlen rechnet) oder jede andere Anwendung einer Funktion auf eine Zahl, die nicht in ihrem Definitionsbereich ist, wie zum Beispiel $\arccos 5$.

Die Standardbehandlung solcher Fälle bestand früher auch in einem Berechnungsstopp und einer Programmunterbrechung, aber auch da bietet der IEEE Standard eine Alternative. Eine Gleitkommazahl, bei der alle Exponentenbits 1 sind aber *nicht* alle Mantissenbits 0 sind, heißt eine **Unzahl**

(**NaN** oder **Not a Number**), und solche Unzahlen werden benutzt als Rechenergebnisse von unbestimmten oder unerlaubten Berechnungen.

Es gibt sogar zwei Varianten von **NaN**, nämlich **signalling NaN** oder **NaNs** (signalisierende Unzahlen), die normalerweise zwingend zu einer Programmunterbrechung führen, und **quiet NaN** oder **NaNQ** (stille Unzahlen), mit denen weitergerechnet werden darf (die Ergebnisse solcher Berechnungen sind meistens wieder NaNQ). In der Regel werden unbestimmte Ergebnisse als NaNQ gekennzeichnet und unerlaubte Operationen als NaNs, aber die Behandlung dieser Ausnahmen ist sehr kompliziert und nicht einheitlich geregelt, so dass auch Abweichungen von diesem Verhalten vorkommen können.

Quiet NaN sind erkennbar daran, dass das höchste Mantissenfeldbit 1 ist; bei signalling NaN ist es 0. Die weiteren Bits des Mantissenfeldes können benutzt werden, um zu kennzeichnen, was die Ursache für das ungültige oder unbestimmte Ergebnis war, damit bei einer Programmunterbrechung eine auf den Fall zugeschnittene Behandlung stattfinden kann.

Programmierer initialisieren oft Werte oder Variablen in ihren Programmen, die vom Programm oder vom Benutzer vor der Verwendung gesetzt werden müssen, zur Sicherheit mit NaNs, um eine Programmunterbrechung zu provozieren, falls das richtige Setzen dieser Werte vor der Verwendung vergessen wird!

Damit haben jetzt alle Bitmuster eine Verwendung gefunden und wir sind fertig mit der Beschreibung des IEEE Standards.

Wir wissen jetzt, wie Gleitkommazahlen aussehen, aber wir haben noch nicht darüber gesprochen, wie man mit ihnen rechnet. Das wollen wir direkt anhand von Beispielen kurz erklären. Die Beispiele werden wir mit dem IEEE Standard einfacher Genauigkeit rechnen und haben damit auch Beispiele für die Umwandlung von Zahlen in dieses Format.

Beispiel 3.3 a) Wir wollen einige Gleitkommaberechnungen mit den Operanden 6 und 11 durchführen und überführen diese Zahlen zunächst in das IEEE Format einfacher Genauigkeit.

Wir beginnen mit der 6. Diese Zahl ist positiv, so dass das Vorzeichenbit 0 ist.

Als Binärzahl in naturwissenschaftlicher Notation ist 6 gleich

$$110 \times 2^0 \quad \text{oder normalisiert} \quad 1,1 \times 2^2$$

(der Exponent erhöht sich in der normalisierten Darstellung auf 2, weil wir das Komma um zwei Stellen nach links geschoben haben).

Die Exzess 127 Darstellung von 2 in acht Bits ist

$$127 + 2 = 129 = 10000001 = 81_{16}.$$

Die Zahl 6 als Gleitkommazahl setzt sich zusammen aus dem Vorzeichen 0, den 8 Bits des Exponenten, und der Mantisse *ohne* die führende 1, aufgefüllt mit Nullen auf 23 Bits. Das Ergebnis ist

$$0\ 10000001\ 100\ 0000\ 0000\ 0000\ 0000\ 0000 = 40C00000_{16}$$

Die gleiche Berechnung mit der Dezimalzahl 11 liefert wieder ein Vorzeichenbit 0 und für den Absolutbetrag die Darstellung

$$1011 \times 2^0 = 1,011 \times 2^3.$$

Die Exzess 127 Darstellung von 3 in acht Bits ist

$$127 + 3 = 130 = 10000010 = 82_{16}.$$

Die Gleitkommadarstellung von 11 setzt sich wieder zusammen aus dem Vorzeichen 0, den 8 Bits des Exponenten, und der Mantisse ohne die führende 1, aufgefüllt mit Nullen auf 23 Bits, mit Ergebnis

$$0\ 10000010\ 011\ 0000\ 0000\ 0000\ 0000\ 0000 = 41300000_{16}$$

Wie wir es hier gemacht haben, werden wir Gleitkommazahlen oft hexadezimal schreiben, um sie in der Notation kürzer und lesbarer zu machen. Man muss sich aber im Klaren sein, dass das Exponentenfeld nicht auf ein Nibble ausgerichtet ist und dass man deshalb den Exponenten und das Mantissenfeld nicht so leicht aus der Hexadezimalzahl ablesen kann; für das Entziffern von IEEE Gleitkommazahlen ist es deshalb besser, die Bitfolge explizit hinzuschreiben.

- b) Wir wollen die gerade bestimmten Gleitkommazahlen addieren, also die Gleitkommasumme

$$40C00000 + 41300000 \tag{3.4}$$

bilden (wo beide Summanden hexadezimal geschrieben wurden).

Natürlich wissen wir, um welche Zahlen es sich handelt und was ihre Summe ist, aber wir wollen die Additionsoperation so beschreiben und so durchführen, wie ein Computer es mit diesen oder anderen beliebigen Gleitkommazahlen machen würde. Das heißt, wir wollen das *allgemeine*

Das Angleichen in diese und nicht in die umgekehrte Richtung bewirkt, dass bei beiden Summanden schlimmstenfalls eine 1 vor dem Komma steht und dass auch bei der Summe höchstensfalls noch eine Übertragsstelle hinzukommt, so dass die Operanden und das Ergebnis direkt in die vorhandene Register passen. Durch ein Angleichen in der anderen Richtung könnte der Signifikant eines Summanden sehr groß werden, was auf jeden Fall weitere Arbeitsschritte bei der Behandlung erfordern würde.

In unserem Beispiel sehen wir sofort, dass der zweite Exponent 82_{16} um Eins größer ist als der erste. Also machen wir auch den ersten Exponenten zu 82 und schieben den ersten Signifikanten um eine Stelle nach rechts (wodurch das letzte Bit aus dem Register geschoben wird und verloren geht); er wird dann zu 0,1100000000000000000000.

Jetzt haben beide Summanden einen gemeinsamen Exponenten (der auch für die Summe gelten wird) und wir können die Signifikanten addieren:

$$\begin{array}{r} 1,011000000000000000000000 \\ + 0,110000000000000000000000 \\ \hline 10,001000000000000000000000 \end{array}$$

Wegen des Übertrags aus der Vorkommastelle ist die Signifikantensumme nicht normalisiert. Um das Ergebnis wieder als IEEE Gleitkommazahl zu erhalten, müssen wir das Komma um eine Stelle nach links schieben, das Exponentenfeld um Eins zu 83_{16} erhöhen und nur die Nachkommastellen der normalisierten Mantisse 1,0001000000000000000000 in das Mantissenfeld schreiben.

Die Gleitkommadarstellung der Summe ist also

$$0\ 10000011\ 000\ 1000\ 0000\ 0000\ 0000\ 0000 = 41880000_{16}.$$

Wir können das Ergebnis noch „entziffern“ und auf Richtigkeit überprüfen.

Die Bitfolge der Gleitkommazahl liegt uns ja noch vor, und wir lesen ab: das Vorzeichenbit 0, also ein + Zeichen; das Exponentenfeld 83_{16} , entsprechend einem Exponentenwert von $83_{16} - 7F_{16} = 4$; und das Mantissenfeld 0001 (die nachfolgenden Nullbits müssen wir nicht mitschreiben), entsprechend einem Signifikantenwert von binär 1,0001.

Der Wert der Gleitkommazahl ist

$$+1,0001_2 \times 2^4 = 10001_2 = 17,$$

wie er sein soll.

Wir halten noch die Schritte fest, die, wie wir hier gesehen haben, allgemein für eine Gleitkommaaddition auszuführen sind.

- Wenn beide Summanden das gleiche Vorzeichen haben, ist dies das Vorzeichen der Summe, und die weiter unten auszuführende algebraische Operation ist wirklich eine Addition. Wenn die Summanden verschiedene Vorzeichen haben, ist das Vorzeichen des im Absolutwert größeren Summanden das Vorzeichen der Summe, und die weiter unten auszuführende algebraische Operation ist eine Subtraktion des im Absolutwert kleineren Summanden vom größeren.
 - Die Signifikanten sind aus den Mantissenfeldern zu bestimmen durch Voransetzen einer „1“.
 - Die Exponenten sind zu vergleichen (sie können direkt in der Exzess n Darstellung verglichen werden) und wenn sie verschieden sind, ist der Signifikant des Summanden mit dem kleineren Exponenten um so viele Stellen nach rechts zu verschieben, wie dieser Exponent kleiner ist als der andere. Ein Computer kann das machen, indem er den Signifikanten schrittweise jeweils um eine Stelle nach rechts schiebt und den kleineren Exponenten um Eins erhöht, bis die Exponenten gleich werden. Der ursprünglich größere Exponent ist auch der vorläufige Exponent der Summe.
 - Je nachdem welche Operation beim Vorzeichenvergleich ermittelt wurde, sind die angepassten Signifikanten zu addieren oder der kleinere ist von dem größeren abzuziehen.
 - Wenn das Ergebnis der letzten Operation 0 ist, setzt man das ganze Ergebnis der Berechnung zu 0. Wenn das Ergebnis der letzten Operation nicht 0 und nicht normalisiert ist, muss es durch Verschiebung des Kommas normalisiert werden, wobei der Exponent anzupassen ist. Wird er dabei zu groß oder zu klein, muss eventuell ein unendliches oder denormalisiertes Ergebnis ausgegeben werden; bleibt der Exponent im gültigen Bereich oder muss er nicht angepasst werden, ist das Gleitkommaergebnis der Berechnung aus dem Vorzeichen, dem ermittelten Exponentenfeld und den Nachkommastellen des normalisierten Additions- oder Subtraktionsergebnisses zusammenzusetzen.
- c) Eine *Gleitkommasubtraktion* kann man ausführen, indem man das Vorzeichen des Subtrahenden umkehrt und dann eine Addition nach den

obigen Anweisungen ausführt. Hierfür geben wir kein Beispiel an.

d) Für die Multiplikation von Gleitkommazahlen gilt die Regel:

- das Vorzeichen des Produkts ist Plus, wenn beide Faktoren das gleiche Vorzeichen haben, und Minus, wenn die Faktoren verschiedene Vorzeichen haben;
- die Exponenten sind zu addieren (mit Exzess n Arithmetik), und
- die aus den Mantissenfeldern ermittelten Signifikanten sind miteinander zu multiplizieren, um die Mantisse des Produkts zu erhalten; dieses Produkt ist mindestens 1, aber da der maximale Wert eines (normalisierten) Signifikanten knapp unter 2 liegt, kann das Produkt der Signifikanten maximal knapp unter 4 liegen, mit der Konsequenz:
- *eventuell* (aber nicht immer) muss die Produktmantisse normalisiert werden durch eine Kommaverschiebung um eine Stelle nach links (mit Anpassung des Exponenten).

Für das Produkt $6 \cdot 11$ ergeben sich folgende Schritte:

Die Vorzeichen beider Faktoren sind +, und weil sie gleich sind, ist das Vorzeichen des Produkts +.

Die Exponentenfelder der Faktoren haben die Werte 81_{16} für den Faktor 6 und 82_{16} für den Faktor 11. Die Exzess 127 (oder hexadezimal Exzess 7F) Summe dieser Werte ist

$$81 + 82 - 7F = 103 - 7F = 84.$$

Das Produkt der oben schon ermittelten Signifikanten 1,1 und 1,011 ist

$$\begin{array}{r} 1,011 \\ \times 1,1 \\ \hline 1\ 011 \\ 1011 \\ \hline 10,0001 \end{array}$$

Dies ist nicht normalisiert. Nach Verschiebung des Kommas erhalten wir eine normalisierte Mantisse oder einen Signifikanten von 1,00001 (die Nachkommastellen kommen in das Mantissenfeld) und einen neuen Exponenten von 85_{16} .

Bevor wir diese Daten in eine Gleitkommazahl packen, bestimmen wir zur Sicherheit den numerischen Wert. Der tatsächliche numerische Wert

des Exponenten ist $85_{16} - 7F_{16} = 6$. Das Vorzeichen ist positiv. Der Wert der Gleitkommazahl, deren Daten wir bestimmt haben, ist

$$+1,00001_2 \times 2^6 = 1000010_2 = 64 + 2 = 66,$$

also der richtige Wert des Produktes $6 \cdot 11$.

Die ermittelten Daten ergeben für das Produkt die IEEE Gleitkommazahl

$$0\ 10000101\ 000\ 0100\ 0000\ 0000\ 0000\ 0000 = 42840000_{16}.$$

e) Die Division von Gleitkommazahlen ähnelt der Multiplikation, mit einem paar offensichtlichen Änderungen und Anpassungen. Zur Vollständigkeit beschreiben wir die Schritte, aber wir rechnen kein Beispiel aus.

- Das Vorzeichen des Quotienten ist Plus, wenn Divisor und Dividend das gleiche Vorzeichen haben, und Minus, wenn sie verschiedene Vorzeichen haben.
- Wenn der Divisor 0 ist und der Dividend ungleich 0, dann ist der Absolutbetrag des Quotienten ∞ . Wenn Divisor und Dividend beide 0 sind, ist der Quotient NaN (in der Regel NaNQ). Im Folgenden gehen wir davon aus, dass der Divisor nicht 0 ist.
- Der Exponent des Divisors ist vom Exponenten des Dividenden abzuziehen (mit Exzess n Arithmetik), um den (vorläufigen) Exponenten des Quotienten zu erhalten.
- Der Signifikant des Dividenden ist durch den Signifikanten des Divisors zu dividieren, um die Mantisse des Quotienten zu erhalten. Dieser Quotient ist zwischen $1/2$ und 2 .
- *Eventuell* (wenn der Signifikant des Divisors größer als der Signifikant des Dividenden war) muss die Mantisse des Quotienten normalisiert werden durch eine Kommaverschiebung um eine Stelle nach rechts (mit Anpassung des Exponenten). Wenn nun der Exponent nicht im zulässigen Bereich liegt, muss der Quotient durch ∞ ersetzt werden oder denormalisiert wiedergegeben werden (aber das ist nur die Ausnahmesituation).

Bemerkung 3.4 Die Notwendigkeit, die Signifikanten oder Mantissen der Summanden einer Gleitkommaaddition manchmal zu verschieben, um die Exponenten anzugleichen, kann zu Rundungsfehlern mit unerwarteten Auswirkungen führen.

Wir illustrieren das an einem Beispiel, und damit Sie nicht durch andere technische Details von der wesentlichen Ursache dieses Phänomens abgelenkt werden, führen wir das Beispiel nicht in einem von den bisher vorgestellten Gleitkommasystemen durch, sondern mit dezimalen „Gleitkommazahlen“ in der naturwissenschaftlichen Notation, aber mit einer fest vorgeschriebenen und beschränkten Anzahl von signifikanten Stellen.

Wir werden aber, wie im Rechner, diese Zahlen normalisieren auf Mantissenwerte zwischen 0,1 und 1, und wir werden die Mantissenlänge auf vier Nachkommastellen beschränken, indem wir (in Gedanken) die Zahlen so an den Rand des Papiers setzen, das kein Platz vorhanden ist, um eine fünfte Nachkommaziffer hinzuschreiben. Das entspricht genau der Funktionsweise eines Rechners mit einer festen Registerbreite.

Die Berechnung, die wir ausführen wollen, ist $1234 + 12000 - 13230$. Die Antwort sollte eigentlich 4 sein.

Wenn wir den Ausdruck entsprechend der natürlichen Klammerung

$$(1234 + 12000) - 13230$$

ausrechnen, entstehen folgende Rechenschritte.

Zunächst addieren wir die ersten beiden Summanden, die wir zunächst normalisieren:

$$1234 = 0,1234 \times 10^4 \quad \text{und} \quad 12000 = 0,1200 \times 10^5.$$

Den ersten Summanden müssen wir so anpassen, dass sein Exponent auch 5 wird, und die Berechnung ergibt

$$\begin{array}{r|l} 0,0123 & 4 \times 10^5 \\ +0,1200 & \times 10^5 \\ \hline 0,1323 & \times 10^5 \end{array}$$

Der senkrechte Strich stellt den Papierrand (oder den Registerrand) dar und die Mantissenziffern, die rechts davon rutschen, sind verloren.

Für die anschließende Subtraktion von

$$13230 = 0,1323 \times 10^5.$$

ist keine Exponentenanpassung nötig und wir erhalten

$$\begin{array}{r|l} 0,1323 & \times 10^5 \\ -0,1323 & \times 10^5 \\ \hline 0 & \end{array}$$

Wir erhalten also eine Antwort 0, obwohl das richtige Ergebnis nicht 0 ist! Dieses Fehlverhalten nennt sich aus offensichtlichen Gründen das ***dirty zero problem***.

Hätten wir die Berechnung anders geklammert, als $1234 + (12000 - 13230)$, wäre sie anders ausgegangen. Als ersten Schritt hätten wir

$$\begin{array}{r|l} 0,1200 & \times 10^5 \\ -0,1323 & \times 10^5 \\ \hline -0,0123 & \times 10^5 \end{array}$$

gehabt, dieses Ergebnis hätten wir normalisiert zu $0,1230 \times 10^4$, und die noch auszuführende Addition mit dem ersten Summanden hätte ergeben

$$\begin{array}{r|l} 0,1234 & \times 10^4 \\ -0,1230 & \times 10^4 \\ \hline 0,0004 & \times 10^4 \end{array}$$

oder normalisiert $0,4 \times 10^1 = 4$, der richtige Wert!

Also nicht nur können Gleitkommaberechnungen zu falschen Ergebnissen führen, die Ergebnisse können, anders als in der algebraischen Arithmetik, von der Klammerung oder auch von der Reihenfolge der Summanden abhängen.

Das „dirty zero problem“ und verwandte Probleme haben auch eine praktische Konsequenz für das Programmieren. Wenn, wie es oft in der Praxis vorkommt, Ergebnisse durch sukzessive Annäherungen bestimmt werden sollen, oder wenn Berechnungsschritte zu wiederholen sind, bis zwei Bruchzahlen gleich werden, dann ist es gefährlich, wirklich auf Gleichheit zu testen, weil sie sich wegen Rundungsfehler vielleicht nie ergeben wird. Stattdessen setzt man von vornherein eine genügend kleine aber positive *Fehlergrenze* und bricht die Berechnung ab, oder akzeptiert das Ergebnis, wenn die berechneten Differenzen unterhalb dieser Grenze liegen.

Jetzt sind wir über die Darstellung von Zahlen im Rechner einigermaßen im Bilde, und wir wollen uns nun mit der Frage beschäftigen, wie das Rechnen mit Zahlen in einem Rechner implementiert ist. In Kapitel 2 hatten wir uns mit den Schaltkreisen beschäftigt, die helfen, Daten im Rechner zu speichern und Grundfunktionen der Steuerung auszuführen; diese Grundlage wollen wir jetzt durch eine Untersuchung und Beschreibung der *Rechenwerke* ergänzen.

Die wichtigste Rechenoperation ist natürlich die Addition von unsignierten Zahlen oder von signierten Zweierkomplementzahlen, weil sie die Grundlage für die meisten anderen Rechenoperationen bildet — wir werden uns im Moment auch nicht mehr mit Gleitkommazahlen beschäftigen, weil sie für

den Anfang viel zu kompliziert sind und weil Gleitkommaeinheiten auch auf den viel einfacheren Rechenwerken für unsignierte Zahlen basieren.

Wir beginnen mit dem elementarsten Fall, der Konstruktion eines Addierers für *1-Bit* Zahlen. Hier ist, zur Orientierung, die Additionstabelle für Einbitzahlen:

a	b	s_1	s_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Tabelle 3.2: Die 1-Bit Additionstabelle mit 2-Bit Ergebnis

Hier sind a und b die beiden 1 Bit langen Summanden und s_1 und s_0 sind die beiden Bits der Summe (wenn beide Summanden 1 sind, haben wir auch eine 2-Bit Summe, so dass wir wirklich *zwei* Ausgangsbits benötigen).

Diese Funktionstabelle können wir durch eine einfache Schaltung realisieren, aber bevor wir das machen, sollten wir kurz überlegen. Ein 1-Bit Addierer eignet sich vielleicht, um das Prinzip zu verstehen, aber für praktische Anwendungen reicht er natürlich nicht aus. Wir benötigen in Wirklichkeit Addierer für Mehrbitzahlen (früher 8 Bit oder 16 Bit, heute eher 32 Bit oder 64 Bit, auf jeden Fall weit mehr als einer).

Trotzdem ist der 1-Bit Addierer die Grundlage für jede Addition im Rechner, aber meistens werden wir für jede Bitstelle in einer längeren Zahl einen 1-Bit Addierer einsetzen, um die Summe an dieser Stelle auszurechnen. Das heißt, die meisten n -Bit Addierer bestehen aus n miteinander gekoppelten 1-Bit Addierern.

In dieser Konstellation verwandelt sich die Bedeutung der oben tabellierten Funktion s_1 ; sie wird nicht verwendet werden als das oberste Bit des Ergebnisses, sondern als ein **Übertrag** an die nächsthöhere Bitstelle, der dort in die Summe eingebaut werden muss.

Die Werte in Tabelle 3.2 ändern sich dadurch nicht, nur die Funktion des Wertes s_1 im gesamten Addierwerk. Wir nennen diesen Wert deshalb um in c für **Carry** (= **Übertrag**) und weil jetzt nur s_0 als der Stellenwert der Summe interpretiert wird, lassen wir den Index weg und nennen diesen Wert schlicht s für **Summe**. Die neue Funktionsausrichtung steht in Tabelle 3.3 auf der nächsten Seite.

Es ist sehr leicht, ein Schaltkreis (mit zwei Ausgängen) zu bauen, das die booleschen Funktionen in Tabelle 3.3 berechnet.

a	b	c	s
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Tabelle 3.3: Die 1-Bit Additionstabelle mit Summe und Übertrag

Der Ausgang c ist genau dann 1, wenn beide Eingänge 1 sind, so dass c durch ein AND-Gatter realisiert werden kann.

Der Ausgang s ist genau dann 1, wenn beide Eingänge verschieden sind, d.h., wenn ein Eingang 1 ist, aber nicht beide. Diese Funktion heißt das **exklusive Oder** der Eingänge („ a oder b aber nur eines von beiden“) und wird abgekürzt **xor** genannt. Wir können ein **xor** Schaltkreis leicht aus der disjunktiven Normalform $\bar{a}b + a\bar{b}$ dieser Funktion konstruieren, aber es gibt auch XOR-Gatter (mit dem Symbol $\Rightarrow \text{D}$), die genau diese Funktion berechnen, und ein solches Gatter wollen wir hier einsetzen.

Die Schaltung für den Einbitaddierer sieht dann einfach so aus:

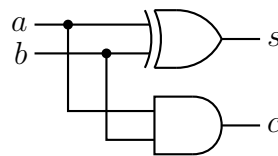


Abbildung 3.1: Ein Halbaddierer

Diese Schaltung (oder jede andere mit der gleichen Funktion) nennt sich ein **Halbaddierer** oder englisch **Half Adder**, und wir werden den Halbaddierer in Zukunft einfach durch einen Kasten mit den nötigen beiden Eingängen und beiden Ausgängen darstellen.

Warum nur „halb“? Als ein *Einbitaddierer* ist die Schaltung völlig in Ordnung, und sie würde sich auch für das niederste Bit einer Mehrbitaddition eignen, aber für die höheren Bits reicht sie nicht aus, denn sie verarbeitet nur die Summandeneingänge, nicht aber einen eventuellen Übertrag von der nächsten Stelle rechts.

Um auch Überträge berücksichtigen zu können, brauchen wir eine Schaltung mit *drei* Eingängen (für zwei Summanden und einen Übertrag) und zwei Ausgängen, die an den Ausgängen wieder die Bitsumme an der gegebenen Stelle ausgibt, sowie einen eventuellen Übertrag in die nächst höhere Stelle. Zwei Ausgänge reichen auch für die Summe von drei Eingängen, weil auch

die Summe von drei Einsen nur $3 = 11_2$ ausmacht und immer noch in zwei Bitstellen passt.

Ein solcher Addierer, der die Summe von drei Eingängen bilden kann, nennt sich ein **Volladdierer** oder **Full Adder** (mit Abkürzung **FA**). Die Wertetabelle für den Full Adder sieht wie folgt aus:

u	a	b	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 3.4: Die Wertetabelle eines Full Adders

Hier haben wir den Übertrags*eingang* u genannt, damit er nicht mit dem Übertrags*ausgang* c verwechselt wird.

Übrigens, die Funktionen in den c und s Spalten haben wir schon in Übungsaufgaben behandelt: offensichtlich ist c die *Mehrheitsfunktion* der drei Eingänge (denn c ist genau dann 1, wenn mindestens zwei Eingänge, also wenn die Mehrheit der Eingänge, den Wert 1 hat) und s ist die *Paritätsfunktion* der Eingänge, die genau dann 1 ist, wenn eine ungerade Anzahl von Eingängen 1 ist.

Trotzdem wollen wir nicht die DN Schaltung für diese Funktionen bauen, sondern einfach bemerken, dass man einen Full Adder billig aus zwei Half Adder zusammensetzen kann, wie in folgendem Diagramm:

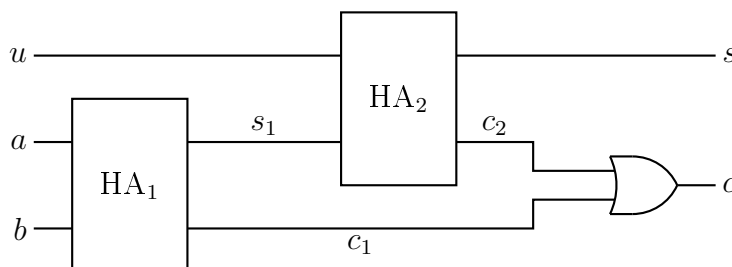


Abbildung 3.2: Ein Volladdierer aus zwei Halbaddierern

Hier haben wir immer den oberen Ausgang der Half Adder und des zusammengesetzten Full Adders als den Summenausgang s gewählt, und den

unteren Ausgang als den Carry Ausgang c .

Die Eingänge des zweiten Halbaddierers HA_2 in Abbildung 3.2 sind der dritte Eingang u des Volladdierers und der Summenausgang des ersten Halbaddierers, also die Summe der Eingänge a und b des Volladdierers. Deshalb ist der s -Ausgang des zweiten Halbaddierers tatsächlich die Summe aller drei Eingänge des Volladdierers.

Bei der Addition findet ein Übertrag statt, entweder wenn es schon bei der Addition von a und b einen Übertrag gibt (also wenn $c_1 = 1$) oder wenn die Berücksichtigung des dritten Eingangs u einen Übertrag produziert (also wenn $c_2 = 1$). Das ODER-Gatter liefert genau dann 1 an seinem Ausgang, wenn eine dieser Bedingungen gilt; sein Ausgang c ist also tatsächlich der Übertrag aus der Addition aller drei Eingänge zum Full Adder.

In Zukunft werden wir auch Full Adder nur durch einen Kasten darstellen, jetzt mit *drei* Eingängen und zwei Ausgängen.

Einbitaddierer sind natürlich nur der Anfang. In der Praxis muss man wesentlich längere Zahlen addieren, und wir wollen deshalb überlegen, wie man am effizientesten Addierwerke für Mehrbitzahlen zusammensetzt. In der Regel (aber nicht immer) bestehen sie aus miteinander verbundenen Half Adder oder Full Adder.

Die einfachste und billigste Möglichkeit ist ein **serieller Addierer**, der nur einen einzigen Full Adder benötigt und mit ihm n -Bit Summen eine Stelle nach der anderen bitweise berechnet. Folgendes Schema kommt dabei zur Anwendung.

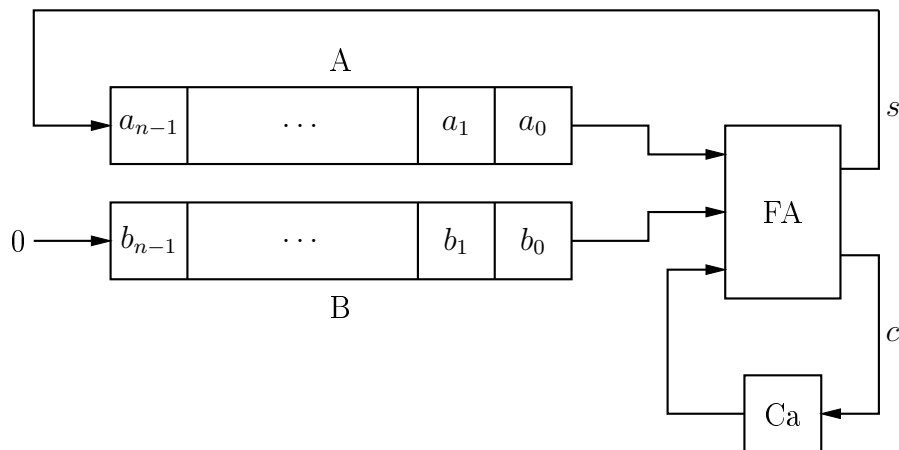


Abbildung 3.3: Ein serieller Addierer

Die beiden Summanden $a = a_{n-1} \dots a_1 a_0$ und $b = b_{n-1} \dots b_1 b_0$ stehen in zwei n -Bit Schieberegistern A und B. Der Baustein FA ist ein Full Adder, dessen Summenausgang s an den Eingang von Register A geführt wird und

dessen Übertragsausgang c an eine 1-Bit breite Speicherzelle Ca (für „Carry“) geleitet wird. Der Eingang des B Registers ist ständig mit einem Signal 0 verbunden.

Die Ausgänge (also die rechten Zellen) der beiden Schieberegister und der Ausgang des Carry Speichers Ca werden an die drei Eingänge des Full Adders geleitet, der an seinen Ausgängen die Summe s und den Übertrag c der Addition dieser drei Bits präsentiert.

Nicht gezeigt in der Zeichnung ist das Taktsignal, das die Schaltung steuert. Die Schaltung berechnet (oder genauer, speichert) bei jedem Takt ein Bit der Summe, indem sie Folgendes gleichzeitig macht:

- Der Inhalt aller Zellen der Register A und B wird um eine Position nach rechts verschoben.
- Der s Ausgang von FA wird in die linke Zelle von Register A gespeichert.
- Eine 0 wird in die linke Zelle von Register B gespeichert.
- Der c Ausgang von FA wird in die Speicherzelle Ca gespeichert.

Die bisherigen Inhalte der ganz rechten Zellen der Register A und B gehen bei dieser Aktion verloren.

Die Eingänge von FA verändern sich durch das Weiterschieben und Speichern von Bits und nach einer kurzen Verzögerung (aber vor dem Beginn des nächsten Takts) erscheinen neue Additionsergebnisse an den Ausgängen von FA, die beim nächsten Takt auf die gleiche Weise weitergereicht werden.

Natürlich darf diese Prozedur nicht unbegrenzt wiederholt werden. Auch nicht in Abbildung 3.3 aufgeführt ist ein Mikrozähler Z , der vor Beginn der Addition mit der Bitanzahl n beladen wird.

Bei jedem Takt wird Z um Eins heruntergezählt und die Verschiebung von Daten wird nur so lange ausgeführt, wie $Z > 0$. Das heißt, wenn Z Null wird bleibt alles stehen, und in diesem Moment befindet sich die Summe der ursprünglichen Inhalte von A und B in Register A, Register B ist mit Nullen gefüllt, und Speicherzelle Ca enthält den Übertrag aus der höchsten Stelle der Summanden und kann deshalb als *Überlaufregister* für die gesamte Addition interpretiert werden.

Hier noch die Skizze eines Mikroprogramms, das diese Addition durchführt (die Anweisungen **reg** := Ausdruck sind wie üblich so zu verstehen, dass der Wert des Ausdrucks den bisherigen Inhalt des Registers **reg** ersetzt):

Schritt 0. $[a_{n-1}, \dots, a_1, a_0] :=$ erster Summand
 $[a_{n-1}, \dots, a_1, a_0] :=$ zweiter Summand
 $Ca := 0$
 $Z := n$

Schritt 1.

Die Ausgänge s und c sind die vom Full Adder aus den Eingängen a_0, b_0 und Ca berechnete Einbitsumme und der Übertrag. Es passiert *gleichzeitig* Folgendes:

$$\begin{aligned} a_i &:= a_{i+1} && \text{für } 0 \leq i \leq n-2 \\ b_i &:= b_{i+1} && \text{für } 0 \leq i \leq n-2 \\ a_{n-1} &:= s \\ b_{n-1} &:= 0 \\ Ca &:= c \\ Z &:= Z - 1 \end{aligned}$$

Schritt 2.

Wenn $Z > 0$ gehe zu Schritt 1; sonst halte.

Der serielle Addierer ist einfach und billig zu bauen, weil er nur einen Einbitaddierer beinhaltet, aber dafür braucht er sehr lange, um eine Summe zu berechnen; die Bearbeitungszeit wächst linear mit der Anzahl der Bits in den Summanden.

Es gibt mehrere Ansätze, nach denen man schnellere Addierer bauen kann. Alle Varianten brauchen für jedes Bit mindestens einen Halbaddierer oder Volladdierer, manchmal auch mehr als einen.

Eine nicht sehr komplizierte aber trotzdem schnelle Konstruktion ist die des **von-Neumann-Addierers**, der für eine n -Bit Summe n Half Adder in parallel einsetzt. Diagramm 3.4 auf der nächsten Seite kann man als Bauplan dafür nehmen.

Der von Neumann Addierer besteht aus zwei n -Bit Register A und B für die Summanden, aus n Halbaddierern (einen für jede Bitposition) und aus einem Einbitspeicher Ov für den **Overflow** oder Überlaufbit. Nicht gezeichnet in der Zeichnung sind die Datenzuleitungen, um die Summandenregister zu laden (oder zu lesen) oder um das Overflow-Bit zu lesen (oder zu initialisieren).

Auch dieser Addierer macht mehrere Durchläufe, gesteuert von einem Taktgeber (aber die Anzahl der Durchläufe ist in der Regel viel kleiner als bei dem seriellen Addierer).

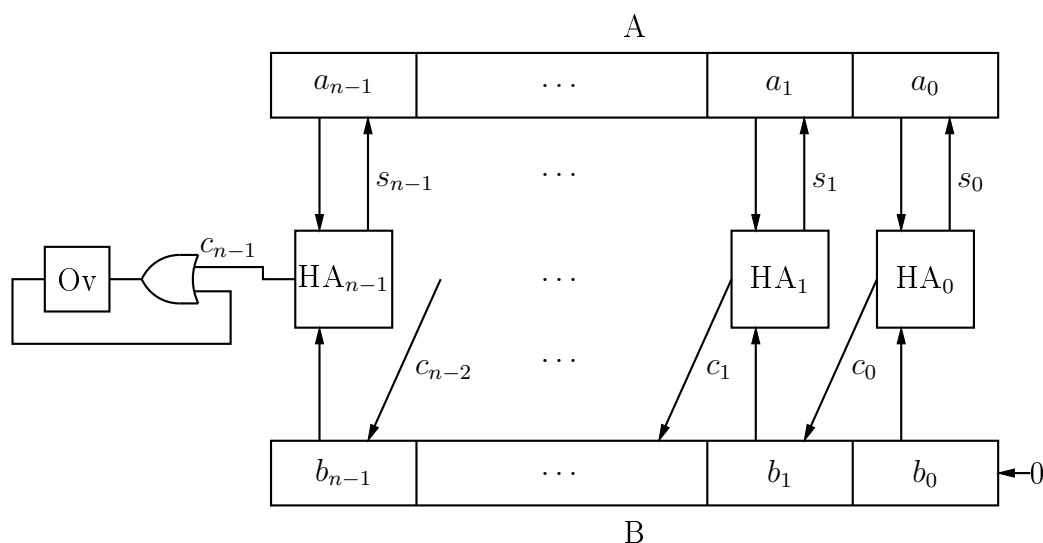


Abbildung 3.4: Ein von Neumann Addierer

Vor Beginn der Addition werden die Summanden in Register A und B geladen und der Overflow Speicher wird genullt.

Bei jedem Durchlauf berechnen die Halbaddierer HA_0 bis HA_{n-1} die bitweise Summe der momentanen Inhalte der Register A und B. Die Summenbits werden zurück in A gespeichert und die Carry Bits werden, um eine Stelle nach links verschoben, in Register B gespeichert. Dabei wird eine 0 in der niedrigsten Bitzelle von B gespeichert und der Übertrag aus dem Halbaddierer an höchster Stelle wird mit dem bisherigen Inhalt des Overflow Speichers geodert und dann wieder dorthin gespeichert.

Diese Operation wird beim nächsten Takt mit den neuen Daten wiederholt, bis alle Bits von B Null sind. Dann steht die Summe der ursprünglichen beiden Summanden in Register A und ein eventueller Übertrag aus der höchsten Stelle steht in Ov.

Die Abbruchbedingung „alle B-Bits 0“ kann leicht mit einem OR-Gatter mit n Eingängen geprüft werden.

Es ist einfach zu verstehen, warum dieser Addierer funktioniert. Zunächst eine informale Begründung:

Bei der ersten Addition werden die Summanden *ohne Berücksichtigung von Überträgen* addiert und die Summe in Register A gespeichert, aber die Überträge werden nicht weggeworfen, sondern an die Bitstelle in Register B gespeichert, an der sie wirken sollen.

Wenn es Überträge gegeben hat, wird noch einmal addiert, um sie in der

Summe zu berücksichtigen. Dabei können aber neue Überträge entstehen, die danach in Register B stehen.

Überträge aus der höchsten Bitposition produzieren einen **Überlauf**, der nicht zur n -Bit Summe gehört aber anzeigt, dass das Ergebnis nicht gültig ist. Dieser Überlauf kann zu jedem Zeitpunkt während der wiederholten Additionen entstehen; damit er nicht vergessen wird, wenn eine spätere Wiederholung der Addition keinen Übertrag produziert, wird er vor dem Speichern immer mit dem bisherigen Inhalt des Overflow-Speichers geodert.

Die Additionen mit den Halbaddierern werden so lange wiederholt, bis alle irgendwann entstandenen Überträge in der Summe erfasst wurden und keine neuen mehr hinzugekommen sind. Dann hat die berechnete Summe den richtigen Wert (wenn man einen eventuell entstandenen Überlauf mit berücksichtigt).

Streng genommen gilt die letzte Behauptung nur, wenn man den Überlauf nicht in einem Einbitspeicher sammelt, sondern die Möglichkeit ins Auge fasst, dass mehrere Überläufe stattfinden könnten, und diesem Umstand Rechnung trägt, indem man Überlaufesamtwerte größer als Eins zulässt und die *Summe* der entstandenen Überläufe in Ov aufnimmt. Wir werden aber gleich sehen, dass in Wirklichkeit diese Möglichkeit *nicht* vorkommen kann und höchstens ein Überlauf während der Addition passieren kann (dann erfüllt unsere Oder-Operation die gewünschte Aufgabe, alle Überläufe zu „summieren“).

Hier jetzt ein genauerer Beweis, dass der von Neumann Addierer die richtige Summe berechnet. Wir betrachten zu jedem Zeitpunkt während der Berechnung die Summe

$$\sum_{i=0}^{n-1} a_i \cdot 2^i + \sum_{i=0}^{n-1} b_i \cdot 2^i + \text{Ov} \cdot 2^n, \quad (3.5)$$

wobei wir in Anbetracht der Bemerkung im vorherigen Absatz zulassen, dass Ov auch Werte größer als Eins annehmen darf.

Zu Beginn der Berechnung ist der Wert des Ausdrucks (3.5) die Summe der in A und B stehenden Operanden (Ov ist zu diesem Zeitpunkt 0). Zum Abschluss der Berechnung ergibt die erste Summation in (3.5) den Wert des *berechneten* Ergebnisses, das im A Register abgelegt wurde, und die zweite Summation ist $\text{Ov} \cdot 2^n$, weil alle $b_i = 0$ sind. Der Gesamtwert von (3.5) ist die berechnete Summe korrigiert mit dem Beitrag aller Überläufe, die stattgefunden haben. Wir behaupten, dass diese um die Überläufe korrigierte berechnete Summe gleich der Summe der ursprünglichen Operanden ist.

Das gilt, weil der Wert von (3.5) bei jedem Rechendurchgang unverändert bleibt! Denn jeder Beitrag $a_i \cdot 2^i + b_i \cdot 2^i$ (für $0 \leq i \leq n-1$) vor dem Durch-

gang wird als Ergebnis der Bearbeitung mit den Halbaddierern durch einen resultierenden Beitrag $s_i \cdot 2^i + c_i \cdot 2^{i+1}$ ersetzt. Der bisherige Überlaufbeitrag geht nicht verloren, weil der Inhalt von Ov nicht durch die Halbaddierer geschickt wird; er kann sich nur eventuell erhöhen durch einen Übertrag aus der $n - 1$ -ten Stelle, der aber oben schon erfasst ist.

Jetzt müssen wir nur überlegen, wie s_i und c_i von a_i und b_i abhängen. Solange a_i und b_i nicht *beide* 1 sind, ist $s_i = a_i + b_i$ und $c_i = 0$, so dass der Wert des i -ten Beitrags sich nicht ändert. Wenn $a_i = b_i = 1$, dann ist $s_i = 0$ und $c_i = 1$, und wir haben effektiv zwei Summanden 2^i durch *einen* Summanden 2^{i+1} ersetzt; der Gesamtwert bleibt also auch in diesem Fall, wie er war.

Da in jeder Runde des Addierens der Wert von (3.5) unverändert bleibt, sind die Werte zu Anfang (Summe der Inhalte der Register A und B) und am Schluss (Ergebnis in A plus Überlaufbeitrag) gleich, wie wir behauptet haben.

Jetzt ist leicht zu begründen, warum höchstens *einmal* ein Überlauf vorkommen kann. Da beide Summanden kleiner als 2^n sind, ist ihre am Schluss in Ov und Register A stehende Summe kleiner als $2 \cdot 2^n = 2^{n+1}$; deshalb ist die im Überlaufregister stehende Zahl (auch wenn wir größere Werte zugelassen hätten) höchstens 1, und ein Einbitregister reicht tatsächlich, um diesen Wert zu erfassen.

Wir präsentieren wieder die Skizze eines Mikroprogramms, das die von-Neumann-Addition durchführt (das Zeichen \oplus steht für **xor** und $+$ steht für **or**, ist also als *boolesche* Operation zu verstehen):

Schritt 0.

$$\begin{aligned} [a_{n-1}, \dots, a_1, a_0] &:= \text{erster Summand} \\ [a_{n-1}, \dots, a_1, a_0] &:= \text{zweiter Summand} \\ \text{Ov} &:= 0 \end{aligned}$$

Schritt 1.

Es passiert *gleichzeitig*:

$$\begin{aligned} a_i &:= s_i = a_i \oplus b_i && \text{für } 0 \leq i \leq n-1 \\ b_0 &:= 0 \\ b_i &:= c_{i-1} = a_i b_i && \text{für } 1 \leq i \leq n-1 \\ \text{Ov} &:= \text{Ov} + c_{n-1} \end{aligned}$$

Schritt 2.

Wenn $b_0 + b_1 + \dots + b_{n-1} \neq 0$ gehe zu Schritt 1; sonst halte.

Die von-Neumann-Addition ist im Mittel erstaunlich schnell, weil es relativ selten vorkommt, dass ein Übertrag über mehr als zwei oder drei Bitpositionen hinweg sich „fortsetzt“ und immer neue Überträge provoziert.

Beispiel 3.5 Zur Illustration rechnen wir ein Beispiel vor, die Addition der unsigneden 8-Bit Zahlen $81 = 01010001_2$ und $211 = 11010011_2$.

Wir präsentieren für jeden Durchgang durch das Addierwerk den Inhalt von Register A und Register B, ergänzt links durch das Overflow Bit (abgesetzt durch einen Strich):

	Vor Beginn	Runde 1	Runde 2	Runde 3
A:	01010001	10000010	00100000	00100100
OV B:	0 11010011	0 10100010	1 00000100	1 00000000

Bei einer typischen Addition werden weniger Runden als die hier erforderlichen drei gebraucht. Beachten Sie, dass das Overflow Bit bei der zweiten Runde gesetzt wurde, aber in der dritten Runde nicht gelöscht wurde, obwohl in dieser Runde kein Übertrag aus der höchsten Stelle stattfand.

Die niederen 8 Bits der Summe sind richtig berechnet, wie man leicht nachprüft.

Der von Neumann Addierer benutzt pro Bit einen Halbaddierer. Man kann auch versuchen, die Überträge gleich zu verarbeiten, indem man nach folgendem Schema Full Adder anstelle der Half Adder einsetzt:

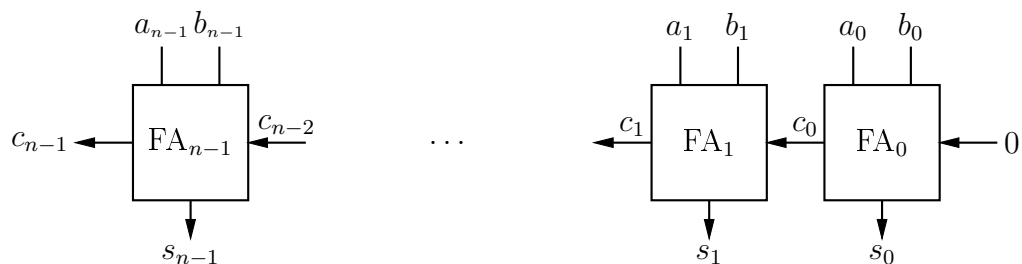


Abbildung 3.5: Ein Ripple Carry Addierer

Das Problem bei diesem Aufbau ist, dass er nicht schneller ist, als der von-Neumann-Addierer, weil die Full Adder nicht wirklich parallel arbeiten können; jeder (bis auf den ersten) muss auf seinen rechten Nachbar warten, um den gültigen Wert des von rechts kommenden Übertrags zu erfahren, bevor er selber operieren kann.

Der Übertrag muss sich also von rechts nach links „ausbreiten“, wie eine Welle oder Kräuselung (englisch *ripple*) auf dem Wasser, wenn man einen Stein hineinwirft. Deshalb heißt dieser Addierer ein ***Ripple Carry Adder***.

Es gibt viele Strategien, um Ripple Carry Adder zu beschleunigen (denn in purer Form haben sie eine Laufzeit proportional zur Anzahl der Bits und sind deshalb kaum schneller als serielle Addierer).

Eine solche Idee besteht darin, einen Ripple Carry Adder (für eine gerade Anzahl von Bits, sagen wir $2k$ Bits) in zwei Hälften für die unteren k Bits der Summanden und für die oberen k Bits der Summanden zu bauen, die dann parallel arbeiten können.

Das einzige Hindernis ist die Tatsache, dass die untere Hälfte des Rechenwerks eventuell einen Übertrag hat, den die obere Hälfte berücksichtigen müsste. Dieser Übertrag ist aber erst bekannt, wenn die untere Hälfte fertig mit der Bearbeitung ist.

Um dieses Hindernis aus dem Weg zu räumen, baut man die obere Hälfte des Addierers in *doppelter* Ausfertigung, und füttert ein Exemplar mit einem festen Übertrag 0 als Eingang zur niedersten Stelle und das andere Exemplar mit einem festen Übertrag 1. Eine Variante wird richtig rechnen, die andere nicht.

Trotzdem können die untere Hälfte und beide Ausführungen der oberen Hälfte jetzt tatsächlich parallel, also gleichzeitig, ihre Daten bearbeiten. Danach ist der wirkliche Wert des Übertrags aus der unteren Hälfte bekannt, und man gibt ihn weiter als Selektiereingang an einen Multiplexer, der das Ergebnis der passenden oberen Hälfte selektiert und ausgibt.

Diese Konstruktion nennt sich ein ***Carry Select Adder*** (Abbildung 3.6 auf der nächsten Seite), und er hat etwa die halbe Bearbeitungszeit eines konventionellen Ripple Carry Adders für die gleiche Bitzahl, dafür aber etwas über 50% mehr materiellen Aufwand.

Die Carry Ausgänge in der Zeichnung sind tatsächlich nur ein Bit breit, aber die Dateneingänge a und b und die Summenausgänge s sind jeweils k Bits breit. Weil das Diagramm sich so leichter zeichnen ließ, haben wir die drei Ripple Carry Adder so gezeichnet, dass der obere Ausgang der Übertrag ist und der untere Ausgang die Summe. Die Indizes u und l bezeichnen die obere („upper“) und untere („lower“) Hälfte der Summanden und der Summe.

Wenn die Gesamtbitzahl eine Zweierpotenz ist, wie heute üblich, dann kann man auch die Hälften des Addierers auf die gleiche Weise *wieder* halbieren, und dieses Aufteilen noch einige Male wiederholen, um den Addierer entsprechend schneller zu machen.

Eine andere Strategie, um einen Ripple Carry Adder schneller zu machen, besteht darin, die Überläufe vor der Addition zu berechnen, so dass nicht auf den sich durch die Bitstellen fortpflanzenden Überlauf gewartet werden muss.

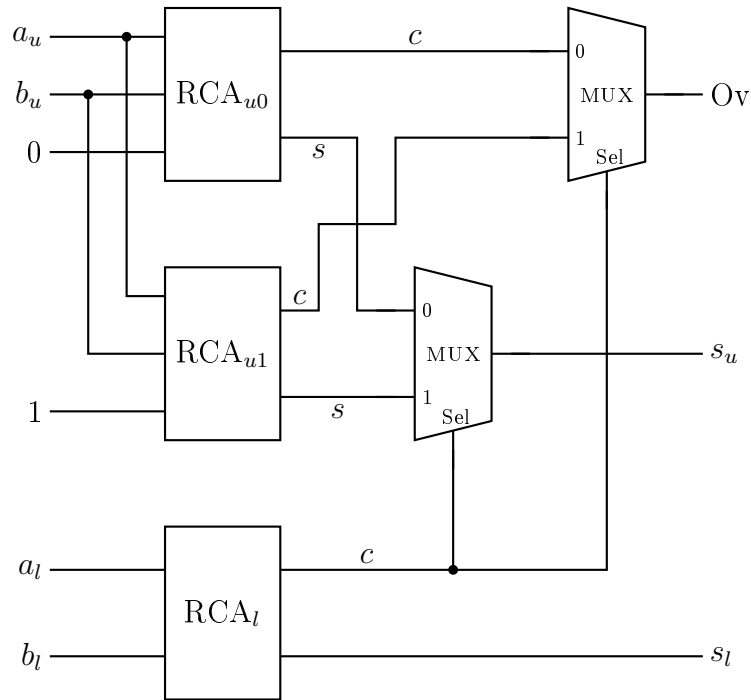


Abbildung 3.6: Ein Carry Select Addierer

Wir benutzen dafür folgendes Berechnungsschema: wir verwenden für jede Bitstelle zwei Einbit Hilfsdaten f_i und h_i , wobei $f_i = 1$, wenn bekannt ist, dass der Übertrag c_i aus der i -ten Bitstelle 1 sein wird, und $h_i = 1$, wenn bekannt ist, dass der Übertrag 0 sein wird.

Solange der Status des Übertrags nicht bekannt ist, gilt $f_i = h_i = 0$.

Schon vor Beginn der Berechnung ist bekannt, dass es keinen Übertrag in die 0-te Stelle hinein geben wird, also, dass es keinen Übertrag aus der -1 -ten Stelle gibt. Deshalb haben wir am Anfang

$$f_{-1} = 0 \quad \text{und} \quad h_{-1} = 1.$$

Manchmal entscheiden die Summandenbits a_i und b_i alleine darüber, ob ein Übertrag stattfindet. Wenn $a_i = b_i = 1$, dann gibt es immer einen Übertrag, also ist in diesem Fall $f_i = 1$ und $h_i = 0$. Wenn $a_i = b_i = 0$, dann kann es keinen Übertrag geben, auch wenn von rechts ein Übertrag kommt, also ist in diesem Fall $f_i = 0$ und $h_i = 1$.

In den anderen Fällen, also wenn a_i und b_i verschieden sind, einer 0, der andere 1, dann gibt es *genau dann* einen Übertrag, wenn von rechts einer kommt, und wir *kennen* den Status des Übertrags also genau dann, wenn wir den Status des Übertrags an der rechten Stelle kennen.

Wir können die genannten Regeln in booleschen Formeln festhalten. Es gilt

$$\begin{aligned} f_i &= a_i b_i + (a_i \oplus b_i) f_{i-1} \\ h_i &= \bar{a}_i \bar{b}_i + (a_i \oplus b_i) h_{i-1} \end{aligned} \quad (3.6)$$

Jeden dieser Ausdrücke kann man mit einer Schaltung aus zwei AND Gattern und einem XOR Gatter berechnen. Alternativ kann man die Ausdrücke oben durch die äquivalenten booleschen Formeln

$$\begin{aligned} f_i &= a_i b_i + (a_i + b_i) f_{i-1} \\ h_i &= \bar{a}_i \bar{b}_i + (\bar{a}_i + \bar{b}_i) h_{i-1} \end{aligned} \quad (3.7)$$

ersetzen und dann beide Funktionen durch folgende Schaltung realisieren:

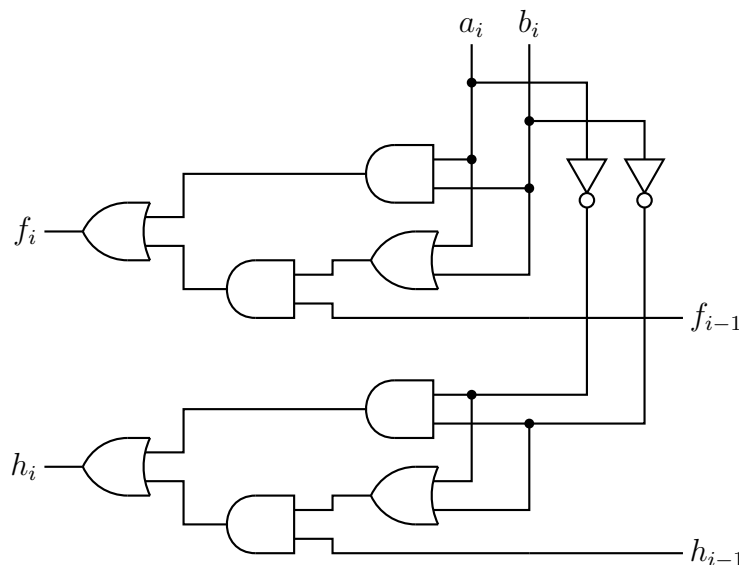


Abbildung 3.7: Die Überlauferkennungsschaltung für eine Bitstelle eines asynchronen Carry Ripple Addierers

Diese Schaltungen muss man miteinander verketteten, indem man die Ausgänge jeder Schaltung mit den entsprechenden f und h Eingängen der nächsten Schaltung verbindet (und den f_{-1} Eingang der nullten Schaltung mit Masse und den h_{-1} Eingang der nullten Schaltung mit der Betriebsspannung verbindet).

Dann lässt man die Schaltungen gesteuert durch ein Taktsignal wiederholt operieren, bis für alle i gilt $f_i + h_i = 1$ (was man mit n OR-Gattern und einem AND-Gatter mit n Eingängen erkennen kann).

Anschließend ergibt sich das i -te Summenbit s_i für die Summe $a + b$ als der Wert des s -Ausgangs eines Full Adders, an dessen Eingängen a_i , b_i und f_i anliegen. Das Overflow Bit ist f_{n-1} .

Dieser Addierer, der alle endgültigen Überlaufbits *vor* Ausführung der Addition ermittelt, heißt ein ***asynchroner Carry Ripple Addierer*** (abgekürzt: ***ACR Addierer***). Er macht im Wesentlichen das Gleiche, wie der von Neumann Addierer, aber ohne die Additionen durchzuführen, bis ganz am Schluss; seine Laufzeit ist auch vergleichbar mit der eines von Neumann Addierers.

Wir wollen ein Beispiel durchrechnen, aber bevor wir damit beginnen, wollen wir noch etwas über die Funktionsweise dieses Addierers bemerken. In den Formeln (3.6) für f_i und h_i können die beiden Summanden auf der rechten Seite nie gleichzeitig 1 sein. Das kann man so verstehen, dass es zwei verschiedene Arten von Additionsspalten gibt, bei denen der endgültige Wert von f_i und h_i auf zwei verschiedene Weisen zustande kommt.

Wenn beide Summandenbits gleich sind (also beide 1 oder beide 0), dann bestimmt der linke Summand in (3.6) den Wert von f_i und h_i schon beim ersten Durchlauf, unabhängig von der Situation an anderen Bitpositionen. Genau dann, wenn die Summandenbits verschieden sind, wird der Wert von f_i und h_i von der nächsten Stelle nach rechts „geerbt“.

Das heißt, dass man das Verfahren auch so durchführen kann: in einem ersten Schritt setzt man die endgültigen Werte von f_i und h_i in allen Spalten, in denen die Summandenbits gleich sind. Auch in der -1 -ten Spalte stehen die endgültigen Werte von vornherein fest.

Im zweiten Schritt kopiert man *gleichzeitig* die im ersten Schritt gesetzten Daten nach links, bis man wieder zu einer Spalte kommt, in der die Werte schon beim ersten Schritt gesetzt wurden und nicht verändert werden dürfen.

Wenn man nur auf Papier rechnet und die Funktionsweise des Addierers nachmacht, ist diese Variante des Verfahrens natürlich viel einfacher und bequemer, als die schrittweise Berechnung der Werte durch die von uns vorgestellte Schaltung. Aber es ist auch nicht schwierig, Schaltungen zu bauen, die das bequemere Verfahren anwenden und in den gerade genannten zwei Schritten (eventuell sogar kombiniert zu einem einzigen Schritt) alle f_i und h_i Werte gleichzeitig bestimmen.

Dafür werden die Schaltungen allerdings wesentlich umfangreicher und komplizierter. Man spart also Rechenzeit, aber man bezahlt dafür mit einem erhöhten Schaltungsaufwand.

Die Idee der gleichzeitigen Berechnung aller Werte ist ähnlich wie das Prinzip der Carry Lookahead Addierer, die wir weiter unten (Seite 150) erläutern.

Beispiel 3.6 Wir wollen einen ACR Addierer benutzen, um die Summe aus Beispiel 3.5 zu berechnen. Die Summanden waren $81 = 01010001_2$ und $211 = 11010011_2$.

Wir schreiben die Summanden übereinander und ermitteln dann schrittweise die f_i und h_i Werte. Zu Beginn sind sie alle 0, außer in der -1 -ten Spalte. In jedem weiteren Schritt werden dann neue Spalten bestimmt, bis wir alle kennen.

```

die Summanden
a:  01010001
b:  11010011

vor Beginn
f:  00000000  0
h:  00000000  1

erste Runde
f:  01010001  0
h:  00101100  1

zweite Runde
f:  11010011  0
h:  00101100  1

```

Schon nach dem zweiten Durchgang sind wir fertig (eine Runde weniger, als beim von Neumann Addierer; allerdings haben wir noch nicht addiert).

Die bei der Addition zu berücksichtigenden Überträge können wir jetzt aus f ablesen, das wir um eine Stelle nach links verschieben müssen, um die Überträge dorthin zu bringen, wo sie wirken.

Das ergibt folgendes Schema von drei Summanden, zu dem wir in jeder Spalte nur das *niedere* Bit der Spaltensumme hinschreiben müssen, um die Summe der ursprünglichen Zahlen zu bekommen, da alle Überträge schon in der Spaltensumme berücksichtigt werden. Das niedere Bit der Spaltensumme ist einfach die Parität der Anzahl von Einsen in der Spalte, also 1, wenn die Anzahl der Einsen ungerade ist, und 0, wenn sie gerade ist.

```

      a:    01010001
      b:    11010011
Übertrag:  1 10100110
      -----
Summe:    1|00100100

```

Als letzten „Bautyp“ wollen wir einen Addierer betrachten, der alle Überträge sofort berechnet und deshalb sehr schnell ist. Aus der Diskussion des asynchronen Carry Ripple Addierers ist klar geworden, unter genau welchen Umständen an einer bestimmten Bitstelle ein Übertrag entstehen wird, und das dafür geltende Prinzip lässt sich sehr leicht in einen digitalen Schaltkreis übersetzen, der alle Überträge unmittelbar und „sofort“ berechnet, also ohne mehrmalige Durchläufe.

Überträge entstehen an Stellen, wo beide Summanden einen 1-Bit haben und entstehen nie an Stellen, wo beide Summanden 0 sind; Stellen, wo die Summandenbits verschieden sind, übernehmen den Status der rechten Nachbarstelle.

Um zu wissen, ob bei Stelle j ein Übertrag entsteht, müssen wir beginnend bei dieser Stelle nach rechts lesen und an jeder Stelle die Summanden vergleichen. Stellen, an denen ein Summand ein 1-Bit und der andere ein 0-Bit hat, ignorieren wir und lesen weiter, bis wir zur ersten Stelle kommen, wo beide Summanden übereinstimmen. Wenn dort beide Summandenbits 0 sind, dann gibt es keinen Übertrag, aber wenn bei der ersten Übereinstimmung beide Summanden ein 1-Bit haben, dann wird ein Übertrag stattfinden, auch an der Stelle, wo wir angefangen haben zu lesen!

Falls wir keine übereinstimmende Stelle finden, dann findet kein Übertrag statt (weil an der -1 -ten Stelle es eine Übereinstimmung mit zwei 0-Bits gibt).

Also: wir haben genau dann einen Übertrag bei Stelle j , wenn es bei Stelle j oder rechts davon eine Stelle $m \leq j$ gibt, wo beide Summanden ein 1-Bit haben, und wenn zwischen dieser Stelle und Stelle j keine Stelle ist, wo beide Summanden 0 sind.

An einer beliebigen Stelle i sind nicht beide Summanden 0 genau dann, wenn der Disjunktionsterm

$$d_i := a_i \vee b_i = 1,$$

und beide Summanden sind 1 genau dann, wenn der Konjunktionsterm

$$k_i := a_i \wedge b_i = 1.$$

Die Terme d_i und k_i lassen sich jeweils durch ein einzelnes Gatter berechnen.

Die Bedingung, die für Stelle m oben verlangt wird, lässt sich also ausdrücken als

$$\left(\prod_{i=m+1}^j d_i \right) k_m = k_m d_{m+1} \cdots d_j = 1.$$

Dies muss nicht für irgendein spezielles $m \leq j$ gelten, sondern es muss nur ein m geben mit dieser Eigenschaft. Das heißt, die genaue und vollständige

Bedingung für das Zustandekommen eines Übertrags an Stelle j ist

$$c_j := \sum_{m=0}^j \left(\prod_{i=m+1}^j d_i \right) k_m = \sum_{m=0}^j k_m d_{m+1} \cdots d_j = 1. \quad (3.8)$$

Diese Formel lässt sich auch anders herleiten. Gleichung (3.7) für f_j lautet

$$f_j = k_j + d_j f_{j-1}. \quad (3.9_j)$$

Wenn alle Berechnungsrunden des asynchronen Carry Ripple Addierers für die f_i und h_i fertig sind, ist f_j das Gleiche wie c_j , der Übertrag, der an der j -ten Stelle (eventuell) entsteht.

Man sieht leicht ein, dass wenn man die rekursive Formel (3.9_j) auflöst, indem man für das f_{j-1} auf der rechten Seite seinen Wert gegeben durch die rechte Seite von Gleichung (3.9_{j-1}) einsetzt, und danach auf entsprechende Weise die dabei neu erscheinenden f_i -Terme auf der rechten Seite immer weiter durch ihren Wert aus der passenden Gleichung (3.9_i) ersetzt, bis alle f_i -Terme auf der rechten Seite aufgelöst sind und keine mehr entstehen, dann erhält man für $c_j = f_j$ genau den Ausdruck (3.8).

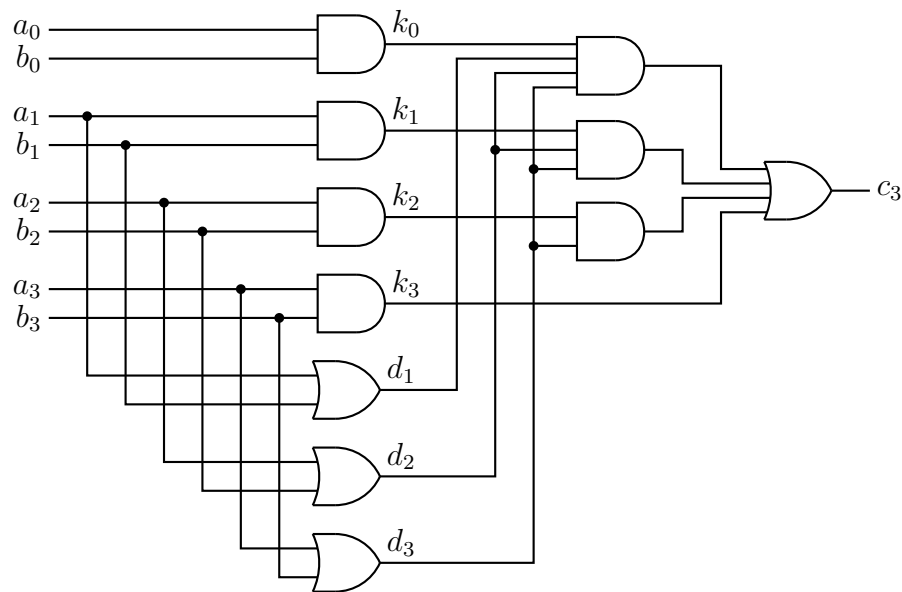
Die ersten Fälle kann man leicht hinschreiben:

$$\begin{aligned} c_0 &:= f_0 = k_0 \\ c_1 &:= f_1 = k_1 + d_1 f_0 = k_1 + d_1 k_0 \\ c_2 &:= f_2 = k_2 + d_2 f_1 = k_2 + d_2 k_1 + d_2 d_1 k_0 \\ c_3 &:= f_3 = k_3 + d_3 f_2 = k_3 + d_3 k_2 + d_3 d_2 k_1 + d_3 d_2 d_1 k_0, \end{aligned}$$

und der allgemeine Fall folgt leicht durch mathematische Induktion.

Die rechte Seite von der Übertragsformel (3.8) ist nur wenig komplizierter als eine disjunktive Normalform; sie wäre eine DN, wenn d_j eine boolesche Variable wäre und nicht selber ein (sehr einfacher) Disjunktionsterm. Entsprechend leicht ist es, eine digitale Schaltung zu entwerfen, die diese boolesche Formel direkt berechnet. Als Beispiel zeigen wir in Abbildung 3.8 auf der nächsten Seite die Schaltung für c_3 .

Wenn man nun einen Addierer aus einem Block aneinander gereihter Full Adder baut, aber den dritten Eingang des Full Adders für jedes Bit nicht mit dem Übertragsausgang des rechten FA verbindet, sondern mit dem Ausgang einer wie oben entworfenen digitalen Schaltung, die das Übertragsbit direkt aus den Daten sofort berechnet, dann reicht ein Durchgang durch die Full Adder, um die richtige Summe als die Bitfolge der s -Ausgänge der Full Adder zu bestimmen.

Abbildung 3.8: Schaltung für den Übertrag c_3

Dies ist also von der Geschwindigkeit her ein optimaler und sehr schneller Addierer. Addierer dieser Art schauen sich im Voraus alle beteiligten Datenbits an, um auf dieser Basis die Überträge ohne Verzögerung zu bestimmen, und sie heißen deshalb **Carry Lookahead Addierer** oder abgekürzt CLA.

Der einzige Nachteil von Carry Lookahead Addierern ist der Materialaufwand an Gattern für die Schaltungen, die die Überträge berechnen (man siehe als Beispiel die Schaltung für c_3 !). Diesen Aufwand können wir leicht schätzen. Wir zählen die benötigten AND und OR Gatter und nehmen diese Gesamtanzahl als Maß für den Aufwand der Schaltungen.

Allerdings gehen wir bei dieser Zählung von Gattern mit genau *zwei* Eingängen aus. Aus den obigen booleschen Formeln ist aber klar, dass wir auch Gatter mit mehr als zwei Eingängen einsetzen müssen. Solche Gatter denken wir uns aus Gattern mit zwei Eingängen zusammengesetzt (das entspricht einer Klammerung eines booleschen Terms in Paare von Operanden). Für jeden neuen Eingang brauchen wir ein zusätzliches Gatter, um das neue Signal mit dem Ausgangssignal aus den bisherigen Eingängen zu kombinieren. Deshalb ist der Aufwand für ein Gatter mit r Eingängen um 1 weniger als die Anzahl der Eingänge, also $r - 1$ (denn die ersten beiden Eingänge brauchen zur Verarbeitung nur ein Gatter).

Diese Zahl $r - 1$ ist gleichzeitig die Anzahl der Operationen $+$ oder \cdot in dem Disjunktionsterm oder Konjunktionsterm, der von dem Mehreingangsgatter berechnet wird. Der Aufwand der Schaltungen für die c_j ist deshalb gleich

der Gesamtanzahl von binären Operationen in den Formeln (3.8).

Die Formel für c_j hat j ODER oder $+$ Operationen zwischen den Konjunktionstermen, und der m -te Konjunktionsterm hat $j - m$ AND oder \cdot Operationen, so dass es Konjunktionsterme gibt mit 0 (bei $m = j$), 1, 2, \dots , j Operationen (die höchste Anzahl bei $m = 0$).

Die Gesamtanzahl von Konjunktionsoperationen in allen Konjunktionstermen in (3.8) ist also

$$0 + 1 + 2 + \dots + j = \frac{j(j+1)}{2}.$$

Hinzu kommen noch die j $+$ -Operationen zwischen den Konjunktionstermen. Die Gatter, die für die Berechnung der k_j und d_j benötigt werden, fallen nicht mehr stark ins Gewicht und werden deshalb hier gar nicht mitgezählt.

Der Aufwand nur für das Carry-Bit c_j beträgt also schon mehr als

$$\frac{j(j+1)}{2} + j = \frac{j(j+3)}{2}.$$

Für die Addition von n -Bit Zahlen braucht der Addierer insgesamt die Carry-Bits c_0, c_1, \dots, c_{n-1} (wobei c_{n-1} der Übertrag *aus* der höchsten Bitstelle ist und nur für den Überlaufstatus benötigt wird).

Der Gesamtaufwand an Gattern für diese Carry-Bits zusammen ist

$$\sum_{j=0}^{n-1} \frac{j(j+3)}{2} = \frac{1}{6}n^3 + \frac{1}{2}n^2 - \frac{2}{3}n$$

(wie man aus den Angaben in einer Formelsammlung leicht ausrechnen kann).

Wenn wir noch die n AND-Gatter für k_0, k_1, \dots, k_{n-1} und die $n - 1$ OR-Gatter für d_1, d_2, \dots, d_{n-1} hinzurechnen, kommen wir auf einen Gesamtbedarf von

$$\frac{1}{6}n^3 + \frac{1}{2}n^2 - \frac{2}{3}n + 2n - 1 = \frac{1}{6}n^3 + \frac{1}{2}n^2 + \frac{4}{3}n - 1 > \frac{1}{6}n^3 \text{ Gattern.}$$

Die Anzahl wächst kubisch mit n .

Für $n = 4$ ist der Aufwand noch klein und beträgt 23 Gatter. Für einen 8-Bit Addierer werden 127 Gatter benötigt, bei 16 Bit sind es schon 831 Gatter, ein CLA für 32-Bit Zahlen braucht für das Berechnen der Überträge 6015 Gatter und ein 64-Bit CLA würde dafür schon 45823 Gatter benötigen.

Auch die Laufzeit können wir schätzen, zum Beispiel mit der maximalen Anzahl der Gatter, die ein Signal hintereinander durchlaufen muss. Wenn wir

davon ausgehen, dass Mehreingangsgatter genau so schnell sind, wie Zweieingangsgatter (was nicht ganz richtig ist), so entspricht die Laufzeit die Verschachtelungstiefe der Formeln (3.8), und sie beträgt einschließlich Auflösung der Terme k_i und d_i nur 3.

Die Schaltungen sind also sehr schnell aber für große Bitzahlen unpraktikabel aufwendig.

Die Addition ist nicht die einzige Rechenoperation, die man mit einem Computer durchführen will. Für die Subtraktion kann man den Subtrahenden als Zweierkomplement negieren und dann zum Minuenden addieren, so dass auch zur Implementation eines Subtrahierbefehles kein Subtrahierwerk erforderlich ist.

Die Multiplikation (und die Division) können auf verschiedene Arten implementiert werden. Einfache und billige CPUs haben oft kein Multiplikationsbefehl, sondern die Multiplikation wird in Software ausgeführt. Aber schon der IBM 7090 aus den frühen 60er Jahre hatte sogar Gleitkommamultiplikation und Gleitkommadivisionsbefehle, während der PDP-1 zur gleichen Zeit, oder später die frühen 8-Bit Mikroprozessoren wie der Intel 8080, keine eingebauten Multiplikations- oder Divisionsbefehle hatten. Heute hat sogar ein sehr einfacher und billiger Rechner wie der Mikrokontroller 8051, der hauptsächlich zur Echtzeitsteuerung von Alltagsgeräten eingesetzt wird, eingebaute Multiplikations- und Divisionsbefehle.

Wir wollen noch kurz schauen, wie man einen Multiplizierer bauen kann. Wir haben in Kapitel 1 gesehen, dass die binäre Multiplikation sehr einfach ist und mit einer kleinen Anzahl von Additionen ausgeführt werden kann, wobei nur für jedes Einsbit im Multiplikator ein Summand verarbeitet werden muss (deshalb ist die Anzahl der Summanden auch „klein“).

Trotzdem gibt es Ansätze, um diese einfache Operation noch effizienter zu machen, wie wir gleich sehen werden.

Wir erinnern kurz an die Multiplikationsmethode, jetzt formuliert für die Ausführung auf einem Rechner mit Registern einer festen Bitlänge. Die Faktoren seien gespeichert in zwei Registern

$$MD = [MD_{n-1}, \dots, MD_1, MD_0] \quad \text{und} \quad MQ = [MQ_{n-1}, \dots, MQ_1, MQ_0].$$

Für jedes Multiplikatorbit MQ_i bilden wir das bis zu $2n - 1$ Bit lange Wort

$$MQ_i \cdot MD = [MQ_i \cdot MD_{n-1}, \dots, MQ_i \cdot MD_1, MQ_i \cdot MD_0, \underbrace{0, \dots, 0}_i],$$

und diese Wörter müssen dann addiert werden, um das Produkt zu liefern. Ein Beispiel für die Produktberechnung auf diese Art (die so genannte „Schulmethode“) haben wir auf Seite 19 in Kapitel 1 vorgeführt.

Die n^2 Produktbits in diesen n Summanden (die Bits, die immer 0 sind, nicht mitgezählt) können sehr schnell und parallel durch n^2 AND-Gatter berechnet werden.

Die Addition von *zwei* Summanden kann zügig mit einem Carry Lookahead Addierer durchgeführt werden. Aber da wir ohnehin in der Regel mehr als zwei Summanden haben werden und deshalb ohnehin mehr als eine Addition durchführen müssen, gibt es auch andere Möglichkeiten und Ideen, um die Addition effizient und schnell durchzuführen ohne den Einsatz von aufwendigen Schaltungen wie ein CLA (ein CLA wird wegen der *Bitbreite* der Summanden auch bald impraktikabel).

Eine solche Idee ist der **Wallace Baum**, eine Erfindung des australischen Informatikers Professor Christopher Stewart Wallace (1933–2004). Ihm ist aufgefallen, dass man bei einer mehrstufigen Addition die Überträge nicht sofort erfassen und weiterverarbeiten muss, sondern in der *nächsten* Stufe einfach wie normale Summanden behandeln kann. Wenn man dann aber *trotzdem* Full Adder einsetzt, aber den dritten Eingang nicht für die Überträge braucht, kann man stattdessen drei normale Summanden auf einmal erfassen und zu nur zwei Ausgangssignalen „komprimieren“, die die Summe ohne Überträge und den Übertragsanteil der Summe darstellen. Hier wird ausgenutzt, dass die Summe von drei Einbitsummanden höchstens zwei Bit lang ist (und somit nicht länger, als die Summe von zwei Einbitsummanden).

Diese Idee kann man wie folgt anwenden, um Multiplizierer zu bauen, die nur unwesentlich langsamer sind als Addierer. Dazu fasst man eine ausreichende Anzahl von Full Addern zu FA Blöcken in der Breite der Summanden einer Multiplikation nach der Schulmethode zusammen. Man füttert die Full Adder Blöcke jeweils mit drei Summanden der für die Multiplikation auszuführenden Additionen und erhält zwei Ausgangswerte. Die Ausgangswerte aller Full Adder Blöcke leitet man, richtig ausgerichtet, wieder in Gruppen von drei an die Eingänge einer *zweiten* Stufe von FA Blöcken, deren Ausgänge dann an die Eingänge einer dritten Stufe weitergeleitet werden, und so weiter, jedesmal unter Verringerung der Anzahl der Summanden um den Faktor $\frac{2}{3}$, bis schließlich nur noch zwei Summanden übrig bleiben. Diese addiert man dann mit einem Carry Lookahead Addierer, um den Endwert des Produktes zu erhalten.

Diese Anordnung von Addiererblöcken in sich verzweigenden Stufen, wobei je drei Eingänge jeder Stufe zu zwei Eingängen der nächsten Stufe kombiniert werden, sieht ein bisschen aus wie ein Baum und wird deshalb nach dem Erfinder ein **Wallace Tree** oder **Wallace Baum** genannt.

Wir zeigen eine Skizze eines Wallace Baums zur Multiplikation von 8-Bit Zahlen. In der ersten Stufe sind dann acht Summanden zu bearbeiten, die aber versetzt ausgerichtet sind, so dass für jede Addition 10-Bit Addierer

eingesetzt werden müssen (in späteren Stufen *wächst* diese Breite). Diese stellen wir durch ein Rechteck dar und zeigen nicht die darin enthaltenen 1-Bit Full Adder.

Die Eingänge des obersten Blocks von Addierern werden belegt mit den Partialprodukten $MQ_i \cdot MD$, die wir mit „ p_i “ abkürzen, weil die volle Bezeichnung nicht in die Zeichnung hineinpasst.

Der Übertragsausgang jedes Blocks muss um eine Stelle nach links versetzt an die nächste Stufe geleitet werden. Das stellen wir in der Skizze durch eine gestrichelte Verbindung dar, während eine durchgehende Verbindung den Summenausgang kennzeichnet, der so ausgerichtet ist, wie der am weitesten rechts ausgerichtete Eingang des entsprechenden Addiererblocks, und der genau so auch weitergeleitet wird.

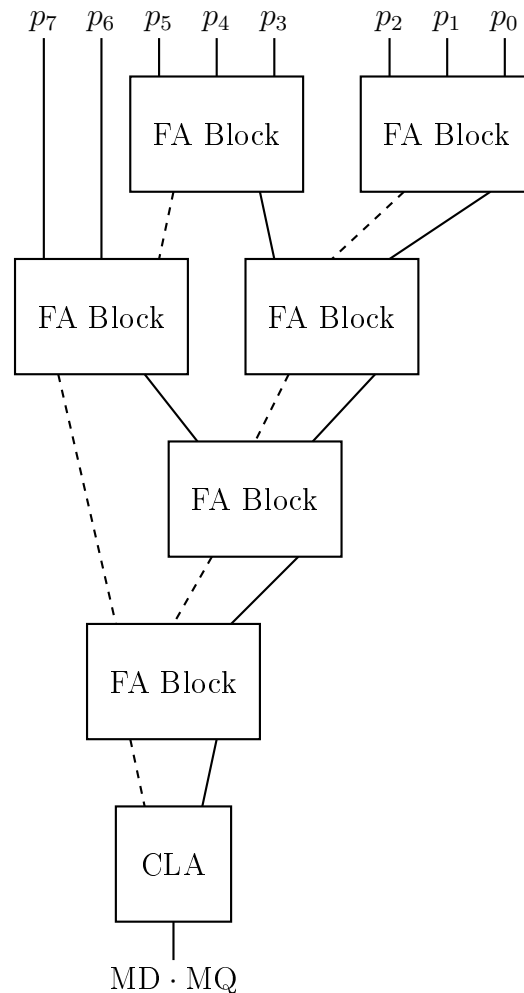


Abbildung 3.9: Ein Wallace Tree für 8 Summanden

Die Stufenanzahl oder Tiefe eines Wallace Baums für n Summanden beträgt etwa $2 \log_2 n$. Das kann man leicht einsehen: bei jeder Stufe reduziert sich die Gesamtanzahl der Summanden um den Faktor $\frac{3}{2}$, so dass die Anzahl der Stufen in etwa

$$\log_{3/2} \left(\frac{n}{2} \right) + 1$$

betragen sollte (der Logarithmusterm misst wieviele Stufen nötig sind, um auf zwei Summanden zu reduzieren; dazu addieren wir 1 für die letzte, CLA Stufe). Weil aber $\frac{3}{2}$ in etwa $\sqrt{2}$ beträgt, sind Logarithmen zur Basis $\frac{3}{2}$ etwa doppelt so groß, wie Logarithmen zur Basis 2. Deshalb können wir den Ausdruck für die Tiefe umschreiben als

$$2 \log_2 \left(\frac{n}{2} \right) + 1 = 2 \log_2 n - 2 \log_2 2 + 1 = 2 \log_2 n - 1.$$

Das passt genau für obiges Beispiel!

Bis auf die letzte Stufe werden in einem Wallace Baum einfache Full Adder durchlaufen. Die Tiefe des Baums ist deshalb auch ein Maß für die Laufzeit des Wallace Baum Multiplizierers, wo die „Maßeinheit“ die Durchlaufzeit eines Full Adder ist. Die Laufzeit des Wallace Baums wächst nur logarithmisch mit der Bitzahl. Das heißt, 32-Bit Zahlen zu multiplizieren braucht nur etwa die doppelte Zeit, die eine 8-Bit Multiplikation verlangt.

Übrigens, ein von Neumann Addierwerk braucht zur Addition von zwei n -Bit Zahlen durchschnittlich $\log_2 n - 1$ Durchgänge, so dass ein Wallace-Baum-Multiplizierer für die Berechnung eines Produkts in etwa die doppelte Zeit braucht, die ein von Neumann Addierer für die *Addition* der beiden Zahlen bräuchte (abgesehen, natürlich, von der Tatsache, dass die Halbaddierer, aus denen ein von Neumann Addierwerk zusammengesetzt ist, ein bisschen schneller als Volladdierer sind). Trotzdem: man kann sagen, dass Wallace-Baum-Multiplizierer recht schnell sind.

Abbildung 3.9 ist nur eine grobe Darstellung. In Wirklichkeit wird ein Wallace Tree *bitweise* aus Full Adder und Half Adder zusammengebaut. Die Verbindungen in einem bitweise zusammengesetzten Wallace Baum zu zeichnen ist natürlich viel umfangreicher und umständlicher, als das grobe Schema aus Abbildung 3.9 (weshalb wir keine Zeichnung zeigen wollen), aber es gibt eine einfache Regel, die beschreibt, wie man einen solchen bitweisen Wallace Baum entwerfen kann.

Die Bits, die für eine Multiplikation addiert werden müssen, sind die Bitprodukte $p_{ij} := MD_i \cdot MQ_j$ für alle Paare i und j mit $0 \leq i, j \leq n - 1$. Dieses Bitprodukt ist an der $i + j$ -ten Stelle im Endergebnis zu berücksichtigen, und wir nennen $i + j$ das **Gewicht** von p_{ij} .

Die Regel für die Konstruktion des Wallace Baums ist nun folgende. Wir beginnen mit dem Aufbau der obersten Stufe, dann mit der Stufe direkt

darunter, usw. Die Eingänge für die oberste Stufe sind die p_{ij} . Sie und die Ausgänge schon eingebauter Addierer werden nach folgendem Schema an die Eingänge weiterer Addierer geleitet:

- Wenn auf einer bestimmten Stufe *kein Gewicht* mehr als zwei Eingangssignale hat, ist dies die unterste Stufe. Unter diesen Umständen können die vorhandenen Eingangssignale zu zwei Summandenwörter zusammengefasst werden. Diese werden auf ein CLA geleitet und addiert; die Summe ist das zu berechnende Produkt.

Wenn aber diese Regel *nicht greift*, dann sind wir noch nicht auf der untersten Stufe, und wir bearbeiten die Stufe nach den folgenden Regeln.

- Wenn für ein bestimmtes Gewicht auf einer bestimmten Stufe (beginnend mit der oberen) mindestens drei unverarbeitete Eingangssignale vorhanden sind, werden drei davon auf ein Full Adder geleitet, und es entsteht auf der nächst tieferen Stufe ein neues Signal des gleichen Gewichts (der Summenausgang) und ein neues Signal des nächst höheren Gewichts (der Übertragsausgang).

Diesen Schritt wiederholen wir so lange wie möglich.

- Wenn für ein bestimmtes Gewicht auf einer bestimmten Stufe (beginnend mit der oberen) genau zwei unverarbeitete Eingangssignale vorhanden sind, werden sie auf ein Half Adder geleitet, und es entsteht auf der nächst tieferen Stufe ein neues Signal des gleichen Gewichts (der Summenausgang) und ein neues Signal des nächst höheren Gewichts (der Übertragsausgang).
- Wenn für ein bestimmtes Gewicht auf einer bestimmten Stufe nur noch ein oder gar kein unverarbeitetes Eingangssignal vorhanden ist, wird das Signal, falls es eins gibt, auf die nächste Stufe geleitet, und für dieses Gewicht ist die gegenwärtige Stufe fertig. Wir verarbeiten die anderen Gewichte auf dieser Stufe, bis alle fertig sind, und gehen dann zur nächst tieferen Stufe über (wieder mit der ersten Regel oben).

Ein bitweiser Wallace Baum für Produkte von zwei 8-Bit Zahlen hätte wie im größeren Schema 5 Stufen, wobei die letzte ein CLA ist. In den Stufen davor benötigt man insgesamt 24 Half Adder und 36 Full Adder (jeweils ein Bit breit), wenn man berücksichtigt, dass an manchen Positionen kein Übertrag eingehen kann (so dass man dort keinen Addierer braucht).

Im Wesentlichen unterscheidet sich dieser Wallace Baum vom größeren nur dadurch, dass wir Buch darüber führen, an welchen Stellen in den FA

Blocks Half Adder anstelle von Full Adder ausreichen und an welchen Stellen ein Adder entfallen kann, weil an der Bitposition nur ein Signal eingeht. Diese etwas genauer charakterisierten Bitadder kann man wie in der groben Zeichnung zu Blöcken zusammenfassen, und man würde dann genau das Schema von Abbildung 3.9 wieder erhalten.

Neben dem Wallace Baum gibt es noch andere Ideen, um Multiplikationen zu beschleunigen. Ein einfacher Gedanke besteht darin, Gruppen von mehreren Bits im Multiplikator zusammenzufassen und mit dem Multiplikand zu multiplizieren, um eine im Vergleich zur reinen binären Multiplikation kleinere Anzahl von Zwischenprodukten zu erzeugen, die man anschließend addieren muss, um das Gesamtprodukt zu erhalten.

Die Produkte von den kleinen Bitgruppen mit dem Multiplikand muss man nicht jedesmal neu berechnen; das kann man im Voraus einmal tun und die benötigten Produkte dann, wann immer man sie braucht, aus einer abgespeicherten „Multiplikationstabelle“ für die Bitgruppen ablesen, so dass dafür nicht viel Zeit aufgewendet werden muss.

Das beste Kosten-Nutzen-Verhältnis erzielt man mit 2-Bit Gruppen; hier halbiert sich schon die Anzahl der Summanden in der Schlussaddition und somit auch die Bearbeitungszeit, aber der im Gegenzug erforderliche zusätzliche Aufwand für die Vorausberechnung der Multiplikationstabelle ist minimal. Sie muss nur das 0, 1, 2 und 3-fache von MD aufführen, und diese Vielfache erhält man mit *insgesamt nur einer* Addition wie folgt:

n	$n \cdot MD$
0	0
1	MD
2	MD eine Stelle nach links geschiftet = $[MD]0$
3	die Summe der letzten beiden Einträge, also $MD + [MD]0$

Mit größeren Bitgruppen wächst die Größe der Multiplikationstabelle exponentiell, aber die Anzahl der Zwischenprodukte, die zu addieren sind, schrumpft nur invers linear. Das heißt, um die Anzahl der Zwischenprodukte noch einmal zu halbieren müsste man 4-Bit Gruppen bilden, und dann hätte die Multiplikationstabelle schon 16 Eintragungen, die auch umständlicher zu erzeugen sind, als die vier oben genannten.

Wir illustrieren dieses Verfahren (mit 2-Bit Gruppen) an einem Beispiel.

Beispiel 3.7 Wir wollen das Produkt von

$$MD = 181 = 10110101_2 \quad \text{und} \quad MQ = 230 = 11100110_2$$

berechnen.

Nach der klassischen Methode erhalten wir für jedes Einsbit in MQ einen Summanden, der eine passend verschobene Kopie von MD ist, wie folgt (die bei der Verschiebung rechts angefügten 0-Bits sind fett gedruckt):

$$\begin{array}{r}
 101101010 \\
 1011010100 \\
 1011010100000 \\
 10110101000000 \\
 101101010000000 \\
 \hline
 1010001010011110
 \end{array}$$

Jetzt berechnen wir das gleiche Produkt mit der beschleunigten Methode. Um je 2 Bits von MQ auf einmal verarbeiten zu können, müssen wir zuerst die Multiplikationstabelle erstellen, die so aussieht:

n	$n \cdot \text{MD}$
0	0
1	10110101
2	101101010
3	1000011111

Wir gruppieren die Bits von MQ in Paaren zu

$$11 \ 10 \ 01 \ 10$$

und müssen dann $[2 \cdot \text{MD}]$, $[1 \cdot \text{MD}]00$, $[2 \cdot \text{MD}]0000$ und $[3 \cdot \text{MD}]000000$ addieren, um das Produkt $\text{MD} \cdot \text{MQ}$ zu erhalten:

$$\begin{array}{r}
 101101010 \\
 1011010100 \\
 1011010100000 \\
 1000011111000000 \\
 \hline
 1010001010011110
 \end{array}$$

Natürlich haben wir wieder das gleiche Ergebnis, und es entspricht der Dezimalzahl $41630 = 181 \cdot 230$.

Die Zeitersparnis *in diesem Beispiel* ist nicht sehr groß, weil der Multiplikator nur eine mittlere Anzahl von Einsbits hatte. Die oben genannte Halbierung der Schrittzahl tritt aber bei einem Multiplizierwerk tatsächlich auf, weil das Multiplizierwerk ja beliebige Faktoren verarbeiten können muss und deshalb auch Nullsummanden durch seine Addierer schicken muss (eine Abfrage, ob ein Summand 0 ist und deshalb nicht verarbeitet werden muss, würde kaum weniger Zeit kosten, als das Addieren des Nullsummanden).

3.1 Nachtrag über Binärbrüche

Ein Thema, dass wir beim Rechnen mit Binärzahlen bisher nicht besprochen haben, ist wie man die Nachkommastellen von binären Bruchzahlen am bequemsten berechnen kann. Die Umwandlung von ganzen Zahlen in ihre Binärdarstellung wurde in Kapitel 1 erläutert, aber wir haben noch keine Methode genannt, wie man Dezimalbrüche oder rationale Zahlen in Binärbrüche umwandeln kann, obwohl man, oder der Rechner, das oft tun muss, zum Beispiel bei der Umwandlung von Zahlen in ein Gleitkommaformat.

Es gibt eine Methode, die immer funktioniert, nämlich bei rationalen Zahlen den Zähler und den Nenner getrennt in das Binärsystem zu wandeln und dann binär rechnend auszudividieren, mit der benötigten Anzahl von Nachkommastellen.

Dezimalbrüche kann man auf die gleiche Weise umwandeln, indem man den Nachkommateil (sagen wir k Nachkommastellen) als Bruch mit Nenner 10^k schreibt und dann ausdividiert. Zum Beispiel würde man 1,7 als $1 + \frac{7}{10}$ schreiben und den Bruchteil wie folgt ausdividieren:

$$\begin{array}{r}
 111,00000000 : 1010 = 0,10110\dots \\
 \underline{101\ 0} \\
 10\ 000 \\
 \underline{1\ 010} \\
 11\bar{0}0 \\
 \underline{1010} \\
 100
 \end{array}$$

Ab diesem Stadium wiederholt sich die Situation, die schon beim zweiten Nachkommabit gegeben war, so dass sich ab jetzt auch die Quotientenbits, ab der zweiten Nachkommastelle, wiederholen werden.

Der Quotient lautet also $0,10\overline{110}$, wo der Querstrich einen periodischen Bruch notiert, in dem die überstrichene Zifferngruppe sich unendlich oft wiederholt.

Die Zahl 1,7 hat also die Binärdarstellung $1,10\overline{110}$.

Will man 1,7 als IEEE-Gleitkommazahl einfacher Genauigkeit schreiben, so ist das Vorzeichenbit 0, das Exponentenfeld lautet $127 = 01111111$, weil 1,7 schon im IEEE Sinn normiert ist, und das Mantissenfeld enthält die ersten 23 oben berechneten Nachkommabits 10110011001100110011010 (wo die letzten Stellen aufgerundet wurden, weil das nächste Bit 1 wäre). Wenn

wir alles zusammenpacken hat 1,7 die IEEE Gleitkommadarstellung

$$\begin{aligned} &0\ 01111111\ 10110011001100110011010 \\ &= 0011\ 1111\ 1101\ 1001\ 1001\ 1001\ 1001\ 1010 \\ &= 3FD9999A_{16}. \end{aligned}$$

Das Unangenehme an der obigen Umwandlung von 1,7 in das Binärsystem war die Notwendigkeit, im ungewohnten Binärsystem eine Division auszurechnen. Zum Glück gibt es eine andere, bequemere Umwandlungsmethode, die diese Unannehmlichkeit vermeidet.

Die Idee ist ganz einfach. Gegeben sei eine beliebige reelle Zahl c , die eine Binärdarstellung

$$c = a_k a_{k-1} \dots a_2 a_1 a_0, b_1 b_2 b_3 \dots$$

hat, mit noch zu bestimmenden Bits a_i und b_j vor und hinter dem Komma. Hier bezeichnet b_j den j -ten Nachkommabit, der an der 2^{-j} -Stelle steht.

Die Vorkommabits kann man nach den Methoden aus Kapitel 1 aus dem ganzzahligen Anteil von c berechnen. Uns geht es hier nur darum, wie man die Nachkommabits bestimmt, und wir können deshalb vereinfachend annehmen, dass es keine Vorkommabits gibt, also dass $0 < c < 1$ und dass c die Form

$$c = 0, b_1 b_2 b_3 \dots$$

hat.

Die b_j können wir eins nach dem anderen „ablesen“, indem wir sie am Komma vorbeischieben. Das ist ganz einfach zu machen: wenn wir c verdoppeln, rutscht das Komma einfach eine Stelle nach rechts.

Die Bits, die so über das Komma hinwegrutschen, lesen wir ab und schmeißen sie anschließend weg, so dass sie bei weiteren Verdoppelungen nicht im Weg stehen. Nach jedem Verdoppeln kann dann nur 0 oder 1 links vom Komma zu stehen kommen, und das ist der Wert des Bits, der vorher direkt hinter dem Komma stand.

Diese Überlegung ergibt folgenden Algorithmus für die Bestimmung der Nachkommabits einer Zahl c , die positiv aber < 1 ist:

1. Verdoppele c zu $2c$.
2. Wenn $2c < 1$, dann ist das nächste Bit des Bruchs 0. Wenn weitere Bruchbits gewünscht werden, ersetze c durch $2c$ und kehre zu Schritt 1 zurück.
3. Wenn $2c \geq 1$, dann ist das nächste Bit des Bruchs 1. Wenn weitere Bruchbits gewünscht werden, ersetze c durch $2c - 1$ und kehre zu Schritt 1 zurück.

Das Schöne an diesem Algorithmus ist, dass die Berechnungen auch im bequemeren Dezimalsystem durchgeführt werden können; man muss hier also nicht binär rechnen!

Als Beispiel bestimmen wir wieder die Binärdarstellung der Dezimalzahl 1,7. Natürlich haben wir eine 1 als Vorkommaziffer, und wir bestimmen nur die Nachkommaziffern, anhand von $c = 0,7$, nach folgendem Schema:

j	$2c$	b_j	c_{neu}
1	1,4	1	0,4
2	0,8	0	0,8
3	1,6	1	0,6
4	1,2	1	0,2
5	0,4	0	0,4

Da c_{neu} schon nach dem ersten Schritt den Wert 0,4 hatte, wiederholt sich die Berechnung ab dem zweiten Schritt, und deshalb wiederholen sich auch die ab dem zweiten Schritt bestimmten Bitwerte immer wieder.

Wir finden also, wie bei der ersten Berechnung, für 1,7 den Binärbruch $1,10110$.

Die gleiche Methode kann man auch benutzen, um rationale Brüche p/q als Binärkommabrüche zu schreiben. Man rechnet dann einfach mit rationalen Brüchen statt mit Kommazahlen, aber die Schritte sind die gleichen.

Zum Beispiel ist hier die Umwandlung von $c = \frac{1}{3}$:

j	$2c$	b_j	c_{neu}
1	$\frac{2}{3}$	0	$\frac{2}{3}$
2	$\frac{4}{3}$	1	$\frac{1}{3}$

Da schon hier c_{neu} mit dem ursprünglichen Wert von c übereinstimmt, wiederholen sich die bisher bestimmten Bits immer wieder, und wir haben $\frac{1}{3} = 0,0\overline{1}$.

Auch bei dieser Berechnung waren keine binären Rechenschritte erforderlich.

Kapitel 4

Eine einfache CPU

In Kapiteln 2 und 3 haben wir den Aufbau der elektronischen Schaltungen kennen gelernt, die Grunddienste in einem Rechner absolvieren und Rechenoperationen ausführen, Daten bewegen und die Abläufe im Rechner koordinieren. Jetzt wollen wir die dort entworfenen Bauteile miteinander verbinden, um eine einfache zentrale Recheneinheit oder CPU („*Central Processing Unit*“) aufzubauen, die man tatsächlich für die Ausführung von Nutzprogrammen verwenden könnte.

Wir werden die Architektur dieser CPU beschreiben, und zwar sowohl die Mikroarchitekturebene (das heißt, wie die elementaren Schaltungen zu einer CPU miteinander verbunden werden) wie auch die ISA Ebene (welche Befehle implementiert sind und wie sie auf der Mikroarchitekturebene ausgeführt werden). Außerdem werden wir einige einfache Programme mit dem implementierten Befehlssatz schreiben, als Illustration für die Verwendung eines Rechners auf der einfachsten Ebene und um einige elementare Programmiertechniken vorzustellen.

Wegen der großen Vielfalt und der beschränkten uns zur Verfügung stehenden Zeit können wir leider nicht detailliert auf die Architektur von wirklichen häufig vorkommenden CPUs eingehen, wie die Pentium Prozessoren oder Mikrokontroller wie der Intel 8051 Prozessor.

Unsere fiktive einfache CPU wird ein 16-Bit Rechner sein, der mit 16-Bit Daten umgehen kann und der entsprechend auch 16-Bit lange Befehlswörter hat. Die Befehlswörter sind unterteilt in ein 4-Bit großes Befehlscode- oder *Operationcodefeld* am oberen Ende und ein 12-Bit langes *Adressfeld* in den niederen 12 Bits (Abbildung 4.1 auf der nächsten Seite).

Das Adressfeld ist groß genug, um $2^{12} = 4096$ Adressen unterscheiden zu können, und wir gehen davon aus, dass unser Rechner mit einem Hauptspeicher aus 4096 Sechzehnbitwörtern ausgestattet ist, und dass dieser Speicher *wortweise* adressiert wird (und nicht etwa *byteweise*, wie heute üblich ist).

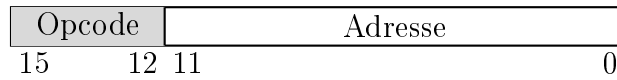


Abbildung 4.1: Ein Befehlswort unseres fiktiven Rechners

Neben dem Hauptspeicher werden noch einige Register benötigt, als Zwischenspeicher für das Rechnen, für das Entziffern und die Ausführung der Befehle, und für die Durchführung der anfallenden Hauptspeicherzugriffe. Hier ist die Liste der CPU-internen Register, die wir in unserem Entwurf implementieren wollen:

Registernamen	Bezeichnung	Bitbreite (Anzahl von Flipflops)
Datenregister	DR	16
Akkumulator	AC	16
Recheneinheit	ALU	16
Adressregister	AR	12
Programmzähler	PC	12
Indexregister	IX	12
Instruktionsregister	IR	4

Tabelle 4.1: CPU Register

Das **Datenregister** wird als Datenzwischenspeicher bei *allen* Datenbewegungen zwischen der CPU und dem Hauptspeicher verwendet, während das **Adressregister** dabei angibt, welches Hauptspeicherwort an dem Datentransfer beteiligt sein soll.

Der **Akkumulator** ist ein für Programme direkt ansprechbares Arbeitsregister, in dem Daten bearbeitet werden können und in dem gerechnet werden kann. Er kann auch für Datenbewegungen direkt durch Programmbefehle angesprochen werden (während das Datenregister nur intern verwendet wird und für den Programmierer *nicht* unmittelbar ansprechbar ist).

Die **ALU** ist das eigentliche Rechenwerk mit Addierern, einem Shiftregister und einem Komplementierer, das alle Rechenbefehle ausführt. Für den Programmierer sieht es allerdings so aus, als würden diese Operationen im Akkumulator stattfinden, weil auch die ALU zu den nur intern zugänglichen Bestandteilen der CPU gehört und nicht unmittelbar für den Programmierer zugänglich ist.

Programme werden wie Daten im Hauptspeicher abgelegt, und der **Programmzähler** regelt die Befehlsabfolge im Programm, indem er immer auf den nächst auszuführenden Befehl zeigt. Er hat also so etwa die Funktion

eines Lesezeichens, mit dessen Hilfe die CPU weiß, an welcher Stelle des Programms sie sich gerade in der Bearbeitung befindet.

Das ***Indexregister*** ist ein Hilfsregister in Adressbreite, das nützlich ist als „Lesezeichen“ bei der Bearbeitung von Datenblöcken oder als „Leitfaden“ für die Rückkehr aus einem ***Unterprogramm***. Ein Unterprogramm ist ein Programmsegment außerhalb der Hauptbefehlsfolge, das eine häufig wiederkehrende Zwischenoperation ausführt und nur einmal im Speicher vorhanden ist, aber aus verschiedenen Stellen innerhalb des Hauptprogramms angesprungen werden kann, um die Operation mehrmals durchzuführen.

Unterprogramme sparen Speicherplatz, aber erfordern eine sichere Methode, um nach jeder Verwendung des Unterprogramms das Hauptprogramm an der *jeweils richtigen Stelle* fortzusetzen. Diese ist jedesmal verschieden und muss dem Unterprogramm deshalb jedesmal mitgeteilt werden. Dazu kann ein Indexregister verwendet werden (obwohl heutzutage meistens eine andere Methode benutzt wird).

Im 4 Bit breiten Operationscodefeld unserer Befehlswörter können $2^4 = 16$ verschiedene CPU Befehle kodiert werden, die die Register ansprechen und verwenden. Obwohl nicht alle Befehle tatsächlich eine Hauptspeicherstelle ansprechen (und deshalb nicht alle Befehle eine Adressangabe erfordern), werden wir nur die genannten 4 Bits für die Kodierung von Befehlen verwenden, und darauf verzichten, das Adressfeld, wo es möglich wäre, zur Unterscheidung einer größeren Anzahl von nicht den Speicher ansprechenden Befehlen zu „missbrauchen“.

Wir wollen die 16 Befehle implementieren, die in Tabelle 4.2 auf der nächsten Seite aufgelistet sind.

Die Notation $M[Y]$ in dieser Tabelle bezeichnet den Inhalt des Hauptspeicherwortes mit Speicheradresse Y , und der doppelte Rückwärtspfeil \Leftarrow bedeutet, dass das Register oder die Speicherstelle links vom Pfeil den Wert des Ausdrucks rechts vom Pfeil als neuen Inhalt erhält.

Wir werden *nicht* genau festlegen, welche 4-Bit Folge genau welchen Befehl in der Tabelle kodiert, da wir diese Information nirgendwo ausnutzen werden. Theoretisch könnte die Kenntnis der Bitkodierung manche Tricks ermöglichen (zum Beispiel, einen Programmbefehl gleichzeitig als eine Datenkonstante mit einem bestimmten Wert zu benutzen, weil zufällig der Befehlscode diesen konstanten Wert ergibt), aber solche Besonderheiten auszunutzen wäre eine ganz schlechte Programmiertechnik, weil solche Programme nicht auf andere Prozessoren (oder neuere Versionen des gleichen Prozessors) portierbar wären.

Bevor wir mit diesen Befehlen Programme erstellen, wollen wir sehen, wie die Befehle von der CPU ausgeführt werden, d. h., welche Daten in welcher Reihenfolge zwischen den Registern, der ALU und dem Hauptspeicher bewegt werden, um die Wirkung eines Befehls zu erzeugen.

Befehl	Y	Wirkung
LOAD	Y	$AC \Leftarrow M[Y]$
STORE	Y	$M[Y] \Leftarrow AC$
ADD	Y	$AC \Leftarrow AC + M[Y] \pmod{2^{16}}$
AND	Y	$AC \Leftarrow AC \wedge M[Y]$ (bitweise)
JUMP	Y	Setze Programm ab Befehl in Speicherstelle Y fort.
JUMPZ	Y	Wenn $AC = 0$, setze Programm ab Befehl in Speicherstelle Y fort, aber wenn $AC \neq 0$, setze Programm mit dem direkt folgenden Befehl fort.
COMP		$AC \Leftarrow \overline{AC}$ (bitweise)
RSHIFT		rotiere die Bits von AC um eine Stelle nach rechts (zyklisch, d.h., Bit 0 wird an Stelle 15 bewegt).
LOAD*	Y	$AC \Leftarrow M[Y+IX]$
STORE*	Y	$M[Y+IX] \Leftarrow AC$
ADD*	Y	$AC \Leftarrow AC + M[Y+IX] \pmod{2^{16}}$
JUMP*	Y	Addiere Y zum Inhalt von IX ($\pmod{2^{12}}$) und setze Programm ab der resultierenden Adresse fort.
DXJNZ	Y	Dekrementiere IX (also ziehe 1 von IX ab) und wenn das Ergebnis $\neq 0$ ist, setze Programm ab Adresse Y fort. Wenn das Ergebnis 0 ist, setze Programm mit dem direkt folgenden Befehl fort.
JSETX	Y	Lade die Adresse des direkt folgenden Befehls in IX und dann setze Programm ab Speicherstelle Y fort.
LDIX	Y	$IX \Leftarrow M[Y]_{0-11}$
STIX	Y	$M[Y]_{0-11} \Leftarrow IX, M[Y]_{12-15} \Leftarrow 0.$

Tabelle 4.2: Die 16 CPU-Befehle

Für die Datenbewegungen müssen gewisse Datenleitungen zwischen den Registern bereitgestellt werden. Die benötigten CPU-Komponenten und die Datenleitungen zwischen ihnen werden im schematischen Diagramm 4.2 auf der nächsten Seite gezeigt.

Der Multiplexer-Demultiplexer ermöglicht den Zugriff auf Hauptspeicherstellen, indem er das Datenregister DR mit dem vom Adressregister AR *selektierten* Speicherwort verbindet. Deshalb ist AR mit dem Kontrolleingang des Multiplexers-Demultiplexers verbunden und DR mit dem Dateneingang oder Ausgang.

Ferner gibt es Datenleitungen vom Datenregister zu allen anderen Registern und zur arithmetischen und logischen Recheneinheit ALU, sowie Datenleitungen von der ALU, vom Indexregister und vom Akkumulator zum Datenregister.

Die ALU ist mit dem Akkumulator, dem Indexregister und dem Datenregister mit Datenleitungen in beiden Richtungen verbunden, weil die Inhalte genau dieser Register an arithmetischen und logischen Operationen beteiligt sein können. Rechenoperationen, die zwei Operanden haben, entnehmen diese aus dem Akkumulator und aus einer Hauptspeicherstelle, aber der Operand aus dem Hauptspeicher wird nur über das Datenregister als Übertragungsstation an die ALU geleitet. Weil das Indexregister an Adressberechnungen beteiligt ist, hat auch dieses Register Verbindungen in beiden Richtungen zur ALU.

Das Adressregister selektiert Hauptspeicherstellen im Multiplexer und Demultiplexer und ist auf diese Weise unmittelbar an Speicherzugriffen beteiligt, aber auch der Programmzähler PC enthält Speicheradressen (von Befehlen und nicht von Daten) und benötigt deshalb eine Leitung zum Adressregister. Da in unserer CPU das Indexregister als Platzmarkierer bei Unterprogrammaufrufen verwendet wird, gibt es auch eine Datenleitung vom Programmzähler zum Indexregister.

Sprungbefehle funktionieren durch Setzen des Wertes des Programmzählers; das bewirkt automatisch, dass der nächste Programmbefehl von der gewünschten Speicheradresse genommen wird. Damit man einen Wert in den Programmzähler laden kann, gibt es eine Datenleitung vom Datenregister zum Programmzähler.

Diagramm 4.2 ist eine etwas vereinfachte Darstellung, denn in Wirklichkeit bestehen die Datenleitungen aus mehr als den gezeigten Verbindungen durch Leiterbahnen. In jeder Leitung liegt noch ein Puffer mit einem Steuereingang, der die Leitung nur bei Anliegen eines Freigabesignals (*enable*) und nur in eine bestimmte Richtung freigibt. Die erlaubte Richtung ist durch die Pfeile gekennzeichnet, aber die Puffer und ihre Freigabeeingänge werden nicht gezeigt.

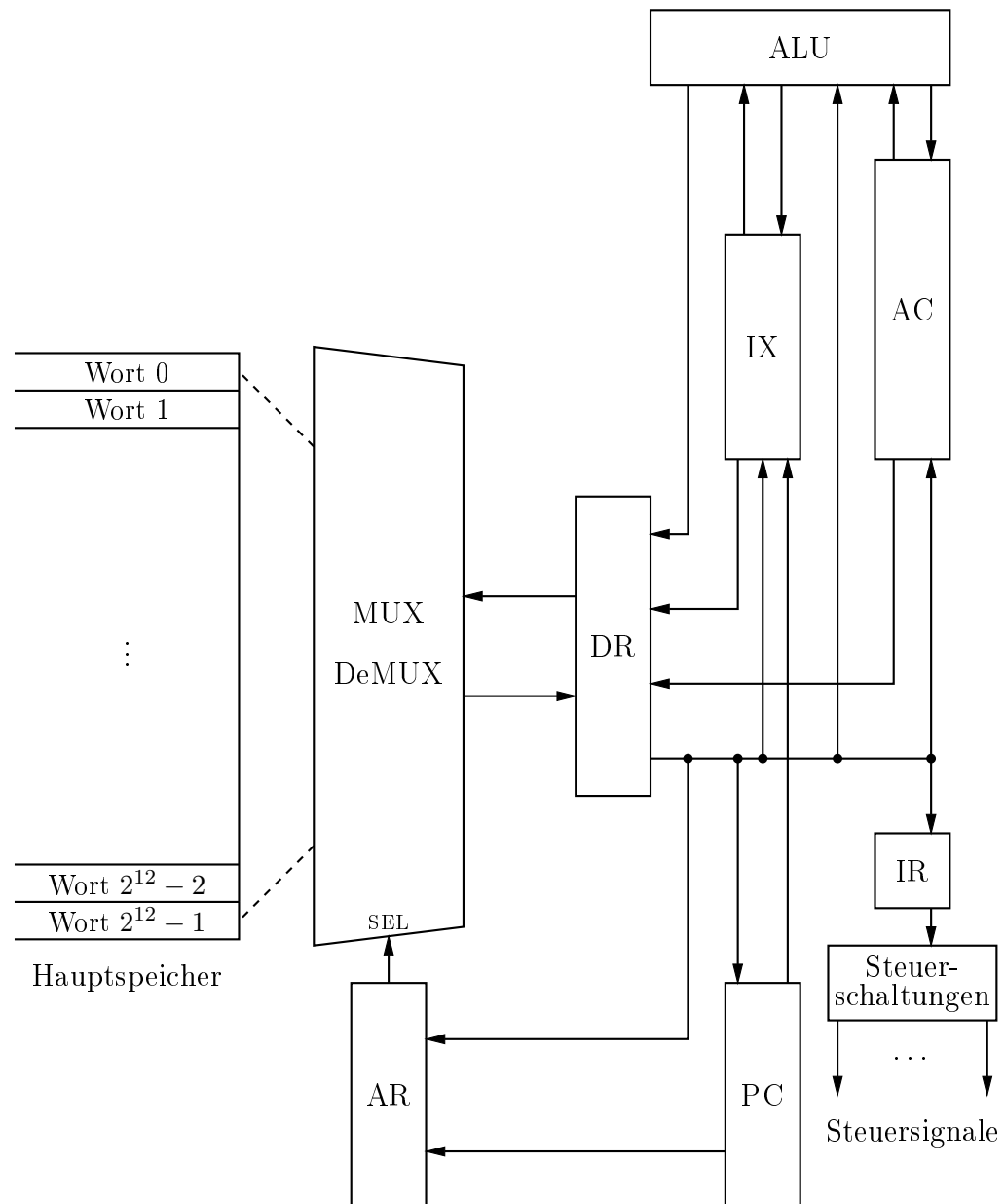


Abbildung 4.2: Die Datenwege in unserer CPU

Die Existenz solcher Freigabesignale bedeutet, dass die Bahnen in der Zeichnung nur den potentiellen Datenfluss beschreiben. Die Daten können nicht zu jeder Zeit und nicht frei fließen, sondern werden zu *bestimmten* Zeiten über *bestimmte* Leitungen bewegt, um eine spezielle Folge von Schritten durchzuführen, die die Maschinenbefehle der CPU ausführen.

Die zeitliche Reihenfolge des Datenflusses und die Freigabe zu bestimmten Zeiten werden durch einen Mikrozähler geregelt, der auch in der Zeichnung nicht erscheint.

Wenn ein Befehl ausgeführt werden soll, muss er zunächst aus dem Hauptspeicher geholt werden und dann *entschlüsselt* werden, d. h., die CPU muss bestimmen, auf welche Hauptspeicherstelle der Befehl wirkt (wenn er den Hauptspeicher anspricht, was die meisten aber nicht alle Befehle tun), und genau welche Wirkung er haben soll. Unmittelbar nach dem Holen aus dem Hauptspeicher landet der Befehl, wie alle von außen kommenden Daten, im Datenregister, und deshalb wirkt die Entschlüsselungsphase auf den Inhalt des Datenregisters. Das Adressfeld, wenn benötigt, wird „gelesen“, indem sein Inhalt an das Adressregister AR geleitet wird. Das Operationscodefeld, 4 Bits breit, wird an das Instruktionsregister IR geleitet und von dort aus weiter an eine Steuereinheit.

Die Steuereinheit besteht aus digitaler Logik, die mit Schaltungen ähnlich zu denen, die wir in den vergangenen Kapiteln besprochen haben, aus dem Stand des schon erwähnten Mikrozählers, aus den einzelnen Bits des im IR gespeicherten Operationscodes und aus gewissen von den vorangegangenen Befehlen gesetzten „Zustandsbits“ die oben genannten Steuer- und Freigabesignale für die Datenleitungen errechnet. Die Freigabesignale sind als *Ausgänge* der Steuereinheit angedeutet; wir haben nur nicht gezeigt, wo sie hingeleitet werden.

Wir werden aber später die Freigabesignale wenigstens in einer Tabelle genau beschreiben und auch erläutern, wie sie gesetzt werden, um die Wirkung eines Maschinenbefehls zu erzeugen.

Die gerade erwähnten Zustandsbits ermöglichen der CPU, in ihrem weiteren Vorgehen auf die Ergebnisse der vorher ausgeführten Operationen oder auf externe Einflüsse zu reagieren. Typische Beispiele sind: ein Overflow Bit, das gesetzt wird, wenn ein Befehl zu einem Überlauf führt; das Vorzeichenbit des Akkumulators; ein Zero Bit, das gesetzt wird, wenn alle Bits des Akkumulators 0 sind, und dergleichen.

Durch Abfrage der Zustände können Rechenfehler abgefangen werden (Fälle, wo die Arithmetik in der beschränkten Welt der Computerregister von den wirklichen Ergebnissen in der unbeschränkten Welt der Zahlen abweichen, oder wo unzulässige Operanden versehentlich verarbeitet wurden), und der Computer kann bedingte Entscheidungen treffen, d. h., den Arbeitsvor-

gang verschiedenen Umständen anpassen und den Umständen entsprechend variieren. Zu den bedingten Entscheidungen gehören nicht nur einfache Abfragen der Sorte „Wenn Bedingung A zutrifft, tue dies und sonst tue das,“ sondern auch die Wiederholung von Programmabschnitten eine begrenzte Anzahl von Malen unter Kontrolle eines Zählers oder einer Bedingung.

Zur Implementierung der Befehle aus unserer Liste benötigen wir zwei Zustandssignale: ein Zerobit für den Akkumulator und ein Nonzerobit für das Indexregister.

Die interne Schrittreihenfolge zur Ausführung der CPU Befehle aus Liste 4.2 wollen wir jetzt beschreiben. Am verständlichsten lässt sich das mit Flussdiagrammen machen, die wir auf den nächsten Seiten für unsere CPU-Architektur präsentieren.

Vor der Ausführung jedes Befehls muss eine Vorstufe ablaufen, in der der Befehl aus dem Hauptspeicher in die CPU geholt wird, und in der dafür gesorgt wird, dass die Befehle einer nach dem anderen in fortlaufender Reihenfolge ausgeführt werden. Diese Vorstufe zeigen wir in Diagramm 4.3.

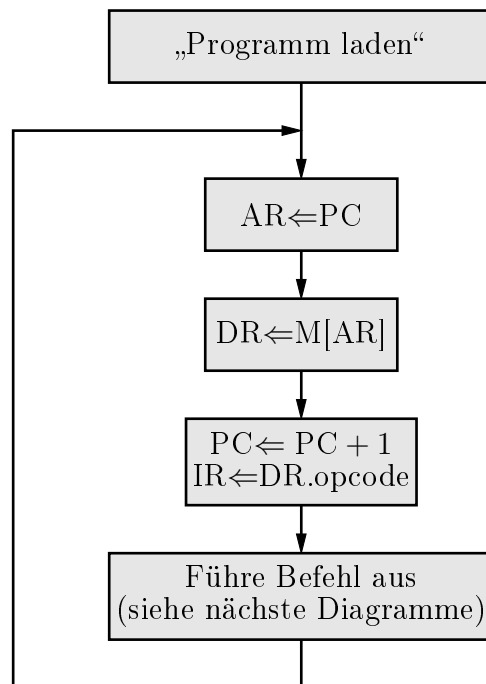


Abbildung 4.3: Vorstufe: Holen des Befehls

Im ersten Kasten in der Schleife wird die Adresse des nächsten Programmbefehls, die ja im Programmzähler PC steht, in das Adressregister geladen.

Der folgende Kasten selektiert mit dieser Adresse das Speicherwort, in dem der nächste Befehl steht, und lädt dessen Inhalt in das Datenregister (wo

alle Einzelheiten des Befehls für die Ausführung ausgelesen werden können, insbesondere auch das Adressfeld des Befehls, wenn der Befehl es auswerten muss).

Im nächsten Kasten wird als erstes der Programmzähler um 1 erhöht, damit er auf jeden Fall am Ende der Schleife auf den folgenden Programmbefehl zeigt, der dann automatisch ausgeführt wird, wenn der vorangegangene Befehl den Programmzähler nicht weiter ändert. Der Operationscodeteil vom Befehl wird in das Instruktionsregister bewegt, damit er im letzten Kasten von der Steuereinheit entziffert werden kann. Die Steuereinheit sorgt mit den von ihr produzierten Freigabesignalen dafür, dass der Befehl ausgeführt wird.

Anschließend kehrt der Ablauf an den Anfang der Schleife zurück, wo der nächste Programmbefehl geholt wird.

In Diagramm 4.4 auf der nächsten Seite zeigen wir den Ablauf der Befehle, die zu ihrer Ausführung nur Daten benötigen, die jetzt schon in der CPU verfügbar sind. Diagramm 4.5 auf Seite 175 behandelt die Befehle, die zu ihrer Ausführung noch ein zweites Mal den Speicher ansprechen müssen.

Die Befehlsverarbeitung beginnt mit einer Abfrage, ob es sich um einen der Befehle mit einem * im Namen handelt, die zu ihrem Adressfeld den Inhalt des Indexregisters addieren. Wenn ja, wird diese Modifikation ausgeführt und das Adressfeld des Datenregisters wird durch den angepassten Wert ersetzt. Danach ist keine weitere Anpassung erforderlich, d. h., diese Befehle können anschließend genau so verarbeitet werden, wie ihre ungesternten Varianten.

Jetzt wird der Operationscode im IR mit den Operationscodes der im großen Abfragekasten aufgelisteten Befehle verglichen. Gibt es eine Übereinstimmung, so folgt die Bearbeitung dem entsprechenden nach unten weisenden Ausgangspfeil.

Die Befehle RSHIFT und COMP bewirken in einem letzten Bearbeitungsschritt die von ihnen bezweckte Veränderung des AC Inhalts.

Die anderen vier Befehle in dem Abfragekasten sind Sprungbefehle und ihr letzter Akt, im untersten Bearbeitungskasten, besteht darin, das Adressfeld des Datenregisters in den Programmzähler zu laden. Beim nächsten Holzyklus (die Vorstufe aus Diagramm 4.3 auf der vorherigen Seite) wird dieser Wert des Programmzählers benutzt, um den nächsten Befehl im Speicher zu finden, so dass effektiv ein Sprung an diese Adresse im Programm stattfindet.

Die einfachen Sprungbefehle JUMP und JUMP* machen nur die gerade beschriebene Operation. JSETX führt sie auch aus, aber vorher speichert er den *alten* Wert des PC, der auf den im Speicher auf JSETX folgenden Befehl zeigt, in das Indexregister, wo er später benutzt werden kann, um mit einem JUMP* 0 Befehl die Programmausführung an der jetzigen Unterbrechungsstelle fortzusetzen.

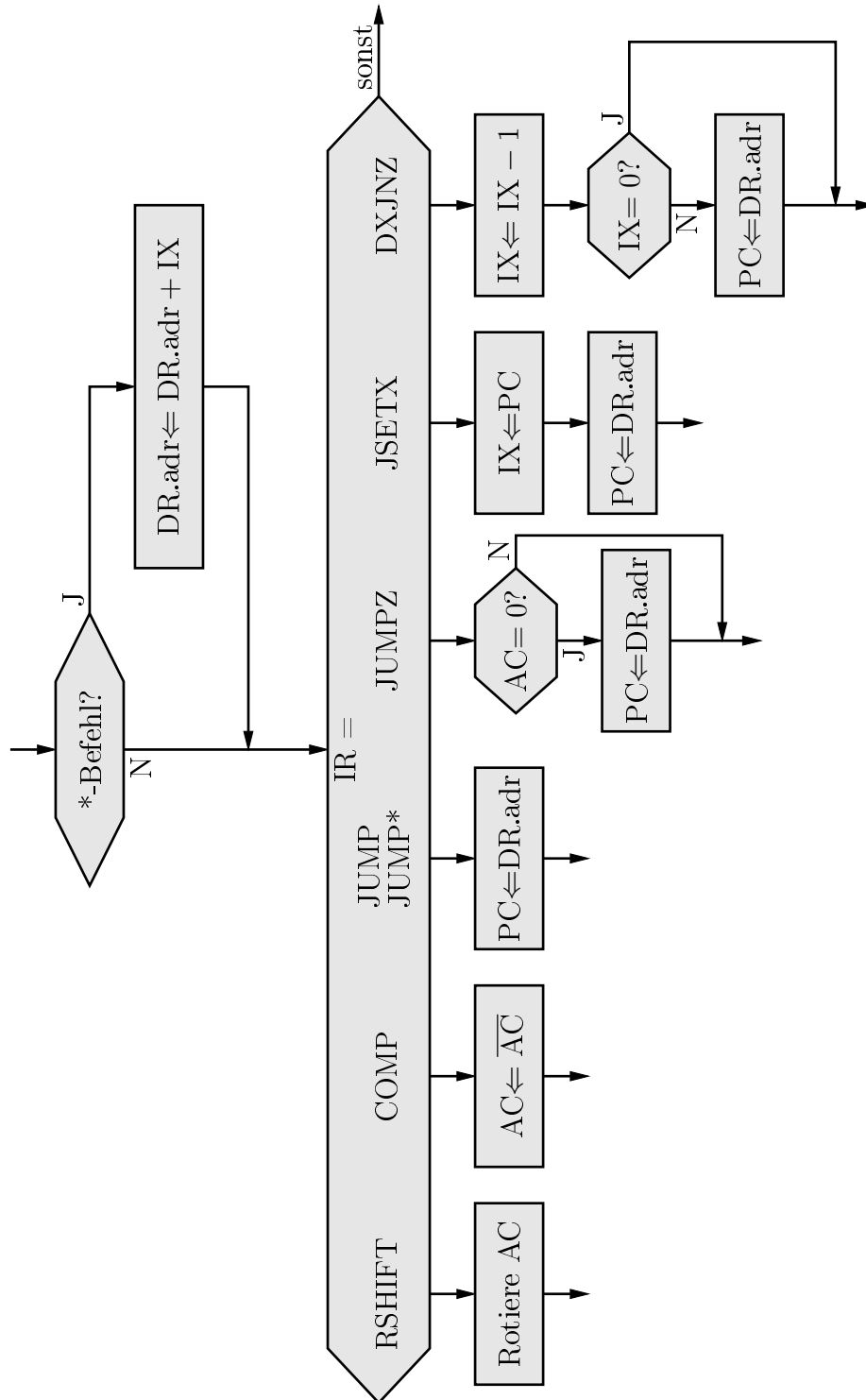


Abbildung 4.4: Befehle ohne weiteren Speicherzugriff

Die letzten beiden Befehle sind bedingte Sprungbefehle; sie testen zuerst den Zustand von AC oder IX und ändern den PC, um einen Sprung zu bewirken, nur dann, wenn der richtige Zustand gegeben ist ($AC = 0$ für JUMPX und $IX \neq 0$ für DXJNZ); andernfalls bleibt der PC auf seinem vorherigen Wert und die Programmausführung wird normal mit dem im Speicher folgenden Befehl fortgesetzt.

Da DXJNZ auch zur Zählung von Schleifendurchläufen genutzt werden soll, dekrementiert er das Indexregister, bevor er den Status prüft.

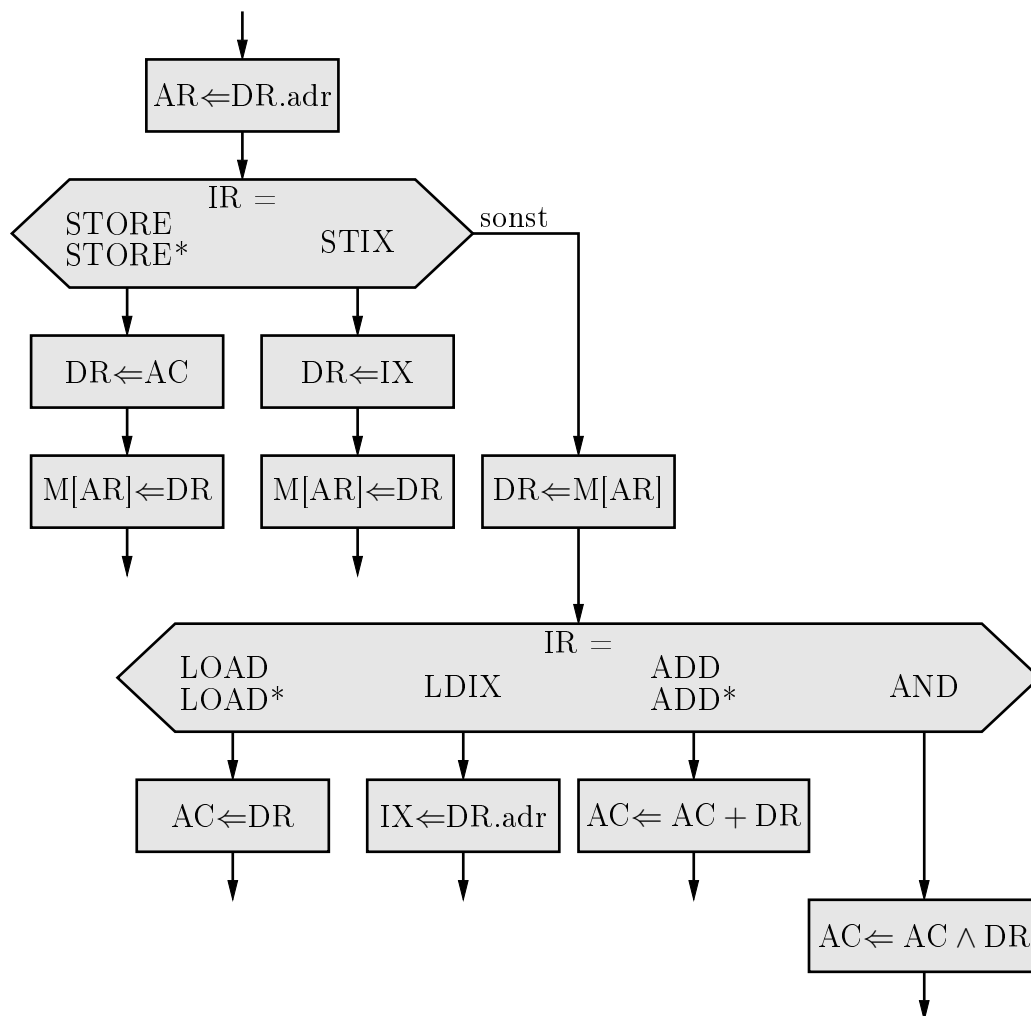


Abbildung 4.5: Befehle mit einem zweiten Speicherzugriff

Die verbleibenden Befehle benötigen einen Datenzugriff auf eine Hauptspeicherstelle, deren Adresse im Adressfeld des Befehls und deshalb jetzt im

Adressfeld des Datenregisters steht. Die Bearbeitung dieser Befehle wird in Diagramm 4.5 auf der vorherigen Seite gezeigt und beginnt damit, dass das Adressfeld des Datenregisters in das Adressregister geladen wird, so dass der Multiplexer-Demultiplexer die im Befehl bezweckte Speicherstelle selektieren wird.

Im ersten Abfragekasten werden die Speicherbefehle, also STORE, STORE* und STIX abgefangen. Sie laden die zu speichernden Daten aus dem AC oder für STIX aus dem Indexregister in das Datenregister und bewegen dann dessen Inhalt mit Hilfe des Demultiplexers an die richtige Speicherstelle.

Alle verbleibenden Befehle holen Daten aus dem Speicher. Im Bearbeitungskasten, der auf dem Ausgang „sonst“ des ersten Abfragekastens folgt, werden diese Daten schon in das Datenregister geholt.

Dann erfolgt eine Abfrage über die Befehle. LOAD und LOAD* bewegen den Inhalt des Datenregisters in den Akkumulator und LDIX bewegt das Adressfeld des Datenregisters in das Indexregister. Die Befehle ADD, ADD* und AND* führen die richtige Rechenoperation mit den Inhalten des Akkumulators und des Datenregisters in der ALU aus und hinterlassen das Ergebnis im AC.

Jetzt wo die Schrittfolge im Mikroprogramm für die Ausführung der einzelnen Makrobefehle bekannt ist, wollen wir noch sehen, wie diese Schrittfolge in der CPU gesteuert und bewirkt wird. Wir haben in Abbildung 4.2 die Datenwege in der CPU gezeichnet und erwähnt, dass diese (mehrbit breiten) Leitungen durch Steuersignale freigeschaltet werden müssen, bevor über sie Daten fließen.

Für diese Leitungen und für die in den Flussdiagrammen vorkommenden Datenflüssen benötigen wir die folgenden neun Steuersignale:

Signal steuert Datenrichtung	
c_1	$DR \leftarrow AC$
c_2	$AC \leftarrow DR$
c_3	$AR \leftarrow DR.adr$
c_4	$PC \leftarrow DR.adr$
c_5	$AR \leftarrow PC$
c_6	$IR \leftarrow DR.opcode$
c_7	$DR \leftarrow IX$
c_8	$IX \leftarrow DR.adr$
c_9	$IX \leftarrow PC$

Tabelle 4.3: Die Steuersignale unserer CPU

Zwei weitere Steuersignale schalten den Multiplexer und den Demultiplexer frei.

Als Bedingungen für das Setzen der Signale, also als Eingänge zu den Schaltungen, die die Signale ausgeben, haben wir die Operationscodes der Befehle, die Ausgänge des Mikrozählers μZ , und zwei Zustandsbits

$$Z = \text{NOR}(AC_i) \quad \text{und} \quad NZ = \text{OR}(IX_i).$$

Der Zustand Z ist genau dann 1, wenn alle AC Bits 0 sind, also wenn der Akkumulator eine Null enthält. Dieses Bit kann als das NOR aller AC Bits berechnet werden.

Der Zustand NZ ist genau dann 1, wenn der Inhalt des Indexregisters *nicht* Null ist; NZ kann als das OR aller IX Bits berechnet werden.

Das Setzen der Steuersignale ist nichts anderes als die Berechnung einer komplizierten booleschen Funktion. Deren Wertetabelle, oder genauer gesagt die relevanten Zeilen der Wertetabelle, präsentieren wir in Tabelle 4.4 auf der nächsten Seite. Damit die Tabelle lesbar bleibt, sind nur die Einswerte angegeben; die Zellen, in denen eine 0 steht, haben wir leer gelassen.

Eingangswertkombinationen, die in der Tabelle nicht aufgeführt sind, können beim Lauf der CPU nicht vorkommen oder sie führen keine Aktion aus: alle Ausgangsbits sind 0, so dass keine Steuersignale gesetzt werden. Deshalb müssen diese Zeilen auch nicht angegeben werden.

Nicht berücksichtigt in dieser Beschreibung und in der Tabelle sind alle Operationen der ALU, obwohl sie in Wirklichkeit auch durch Steuersignale ausgelöst werden müssen und in der Zeitplanung berücksichtigt werden müssen. Diese Details haben wir zur Vereinfachung weggelassen. Auch ist für das Setzen der Zustandsbits Z und NZ kein Zeitbedarf angesetzt worden; wir tun so, als würden diese Zustandssignale immer ohne Verzögerung vorliegen.

Unter anderem zeigt die Tabelle den für jeden Befehl auszuführenden Schritt nicht an, in dem der Programmzähler nach Holen des Befehls inkrementiert wird. Die dafür (eventuell) benötigten Datenleitungen zwischen dem Programmzähler und der ALU sind in Abbildung 4.2 auch nicht angegeben, damit die Zeichnung überblickbar bleibt.

In der vierten Zeile in der Tabelle werden keine (der aufgeführten) Steuersignale gesetzt. Diese Zeile berücksichtigt einen Mikrozählertakt für das Ausführen von arithmetischen Operationen, und zwar für die Addition des Indexregisterinhalts zum Adressfeld des Datenregisters bei den *-Befehlen, und für die Ausführung der akkumulatorinternen Rechenbefehlen RSHIFT und COMP.

Ein entsprechender Zeitbedarf für die Rechenoperationen der Befehle AND, ADD und AND* ist in der Tabelle nicht aufgeführt; man könnte dafür zwei zusätzliche Zeilen vorsehen ohne gesetzte Steuersignale und mit einem Mikrozählerstand um Eins höher als der letzte, der für diese Befehle in der Tabelle steht. Diese „Leerzeile“ haben wir aus Vereinfachungsgründen weggelassen.

μZ	IR=	Z	NZ	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	MUX	de MUX
0								1						
1													1	
2									1					
3	*-Befehle ändern DR.adr, RSHIFT, COMP													
3	JUMP						1							
3	DXJNZ		1				1							
3	JUMPZ	1					1							
3	LDIX, STIX, STORE, LOAD, ADD, AND					1								
3	JSETX											1		
4	JUMP*						1							
4	STORE*, LOAD*, ADD*					1								
4	STORE			1										
4	STIX									1				
4	LDIX, LOAD, ADD, AND												1	
4	JSETX						1							
5	STORE*			1										
5	LOAD*, ADD*												1	
5	STORE, STIX													1
5	LOAD				1									
5	LDIX										1			
6	STORE*													1
6	LOAD*				1									

Tabelle 4.4: Der Zeitplan für das Setzen der Steuersignale

Obwohl die CPU, die wir gerade beschrieben haben, sehr einfach ist und einen sehr einfachen und begrenzten Befehlssatz hat, kann man mit ihr jede rechnerische Aufgabe bewältigen, oder zumindest jede, deren Programm in den sehr kleinen Hauptspeicher passt. Wirkliche moderne Computer haben in der Regel sehr viel mehr Speicher, mehr Register, mehr Befehle und mehr Adressierungsarten (unsere CPU hat nur zwei: direkt und indiziert). Dadurch sind sie viel leistungsfähiger und die gleiche Aufgabe kann man auf ihnen mit kürzeren und weniger umständlichen Programmen erledigen, wie auf unserer CPU.

Will wollen hier keine moderne CPU wie etwa die Pentium 4 vorstellen, weil das wegen der Kompliziertheit ihrer Befehlssätze zu viel Zeit kosten würde. Aber für unsere einfache CPU wollen wir noch ein paar kurze Beispielprogramme schreiben, um die Programmierung auf Maschinensprache oder Assemblerebene zu illustrieren und um ein paar wichtige (oder historisch wichtige) Programmiertechniken vorzuführen, damit man eine Ahnung davon bekommt, wie einige häufig vorkommende Aufgaben in Maschinensprache gelöst werden können.

Ein Programm ist nichts anderes als ein Block von Wörtern an festen Adressen im Hauptspeicher, die wie oben beschrieben eines nach dem anderen in die CPU geholt werden (wobei der Programmzähler immer angibt, welches Wort zu holen ist) und dann ausgeführt werden. So lange keine Adresse aus dem Block im Programmzähler steht, sieht der Block aus wie Daten — erst der Programmzähler bewirkt, dass diese „Daten“ als Befehle betrachtet und ausgeführt werden.

Wir notieren Programmbefehle durch ihre für die Befehlsausführung wesentlichen Merkmale, im vierspaltigen Format

xxxx BEFEHL AAAA Kommentar.

Nur die groß geschriebenen Angaben **BEFEHL AAAA** haben etwas mit dem Inhalt des Befehlswortes zu tun; die Angabe **xxxx** ist eine Zahl und gibt die Adresse im Speicher an, an der das Befehlswort steht. Das Kommentarfeld rechts ist nur für menschliche Leser gedacht und kann zum Beispiel von einem Assembler, der die richtigen Zahlencodes der Befehlswörter an die richtigen Stellen im Speicher schreiben soll, völlig ignoriert werden.

BEFEHL ist der Name des Befehls, also eine symbolische Bezeichnung für den Inhalt des Operationscodefeldes im Befehlswort, ausgewählt aus der Liste in Tabelle 4.2.

AAAA ist wieder eine Zahl (oder leer) und gibt für die Befehle, die ihr Adressfeld auswerten (also alle bis auf **COMP** und **RSHIFT**) den Inhalt des Adressfelds an. Für **COMP** und **RSHIFT** ist **AAAA** leer, aber man kann

davon ausgehen, dass im Befehlswort selber das Adressfeld auf Nullen gesetzt ist.

Die rechte Spalte der Befehlszeile enthält einen **Kommentar**, der beschreibt, was der jeweilige Befehl mit den abstrakten Daten macht, d. h., welchen Zweck er für die Erfüllung der Aufgabe des Programms hat (und nicht welche Wirkung er auf die Registerinhalte und den Speicher hat, denn das ist klar aus dem Rest der Zeile). Sinn des Kommentars ist es, das Programm für Menschen verfolgbar zu machen und dem Programmierer als Gedächtnisstütze zu dienen, falls das Programm wegen einem Fehler oder wegen einer Änderung der Aufgabe modifiziert werden muss.

Die Kommentare sind nicht ein Teil des Programms und nicht zu seinem Funktionieren erforderlich, aber unkommentierte Programme verlieren schnell ihre Nützlichkeit, weil niemand mehr entziffern kann, wozu sie geschrieben wurden oder was sie machen. Es ist also guter Programmierstil, Programme weitgehend zu kommentieren.

Beispiele 4.1 a) Unsere CPU hat einen sehr beschränkten Befehlssatz, in dem nur zwei logische Operationen auf Wörter als Bitfelder vorkommen, nämlich COMP, der bitweise NOT-Operator, und das bitweise ausgeführte AND. Andere logische Operationen muss man durch ein kurzes Programm realisieren, und wir wollen hier ein Programm vorstellen, dass das bitweise OR der Inhalte der Speicherstellen M[0] und M[1] ausführt und das Ergebnis in M[2] speichert.

Unser Programm nutzt die Beziehung

$$a \vee b = \overline{\overline{a} \wedge \overline{b}}$$

aus, um diese Berechnung durchzuführen. Im Kommentar in der rechten Spalte werden wir die Variablennamen a für den ersten Operanden, den Inhalt von Speicherstelle 0, und b für den zweiten Operanden, den Inhalt von Speicherstelle 1 verwenden.

Da wir so wenige Register haben, werden wir Zwischenergebnisse im Hauptspeicher ablegen und aufbewahren müssen, und wir gehen deshalb bei allen Beispielprogrammen davon aus, dass die Speicherzellen 0–15 als ein Arbeitsspeicher benutzt werden, in dem Daten, Konstanten und Zwischenergebnisse der Berechnung abgelegt werden. Unser Programm beginnt deshalb erst bei Speicherstelle 16.

Hier nun endlich unser kurzes Beispielprogramm für die OR Operation:

16	LOAD	0	$AC \leftarrow a$
17	COMP		\bar{a} in AC
18	STORE	2	\bar{a} in M[2] zwischenspeichern
19	LOAD	1	$AC \leftarrow b$
20	COMP		\bar{b} in AC
22	AND	2	$\bar{a} \wedge \bar{b}$ in AC
23	COMP		$a \vee b$ in AC
24	STORE	2	Ergebnis hinterlassen

- b) Jetzt gehen wir davon aus, dass nicht nur zwei Datenwörter geodert werden sollen, sondern dass ab Speicherstelle 100 ein ganzer Block von Daten steht, und dass das bitweise OR *aller* Wörter des Blocks zu bilden ist und im AC zu hinterlassen ist.

Die Anzahl n der Wörter im zu bearbeitenden Datenblock ist variabel und muss dem Programm deshalb mitgeteilt werden. Das Programm soll davon ausgehen, dass diese Zahl bei Programmbeginn im AC Register steht.

Um diese Aufgabe zu lösen, werden wir einen Programmabschnitt, der das nächste Datenwort mit dem bisherigen Zwischenergebnis odert, in einer Schleife mehrmals durchlaufen. Zum Zählen der Durchläufe können wir das Indexregister und den DXJNZ Befehl verwenden.

Die einzelnen Oderoperationen können wir mit dem Programm aus Teil a) durchführen und es wäre möglich, diese Befehlsfolge in den zu wiederholenden Programmabschnitt zu schreiben. Wir wollen aber eine andere Lösung wählen, die zwar für diese spezielle Aufgabe keinen großen Vorteil bringt, aber für das Programmieren allgemein auf diesem Rechner nützlich ist.

Die Oderoperation ist eine Standardoperation, die vermutlich sehr oft in verschiedenen Programmen benötigt wird aber leider nicht im Befehlssatz implementiert ist. Dann macht es Sinn, das OR-Programm nicht direkt in unser neues Programm einzubauen, sondern es irgendwo im Speicher abzulegen, wo es von *jedem* Programm verwendet werden kann, das es benötigt, ohne dass für jede Verwendung eine weitere Kopie des OR-Programms an der verwendenden Stelle stehen muss. Eine einzige Kopie, an einer bekannten Stelle im Speicher, reicht dann aus, sogar wenn die Operation an mehreren verschiedenen Stellen in einem Programm ausgeführt werden soll.

Damit eine einzige Standardkopie des OR Programms für diese Art der Verwendung an mehreren Stellen in externen Programmen eingesetzt

werden kann, muss dafür gesorgt werden, dass das OR Programm nach seiner Ausführung nicht einfach stoppt, sondern an das aufrufende Programm zurückspringt, damit dieses Programm nach der Berechnung des ORs seinen Lauf *mit dem nächsten zu verarbeitenden Befehl* fortsetzen kann. Und dazu muss dem OR Unterprogramm mitgeteilt werden, wo bei jedem Aufruf die jeweils richtige (und sich immer verändernde) Rücksprungstelle sich befindet.

Unsere CPU bietet für diesen Zweck den JSETX Befehl für die Mitteilung des aktuellen Programmzählerstandes vor dem Aufruf an, und ermöglicht mit dem indizierten Sprungbefehl einen Rücksprung an die variierende aufrufende Adresse nach Beendung des OR Unterprogrammes.

Für die Anwendung dieser Techniken muss das OR Programm aus Teil a) geringfügig angepasst werden. Wir werden Speicherstelle 0 nicht mehr für den ersten Operanden verwenden sondern freigeben, und stattdessen den ersten Operanden im AC erwarten. Den zweiten Operanden holen wir nach wie vor aus Speicherstelle 1 (und das Hauptprogramm wird dort sinnvollerweise die bisherigen Zwischenergebnisse des Block-Orders aufbewahren). Auch das Ergebnis des ORs wird nicht mehr in Speicherstelle 2 abgelegt, sondern einfach im AC hinterlassen (aber Speicherstelle 2 wird noch als Zwischenspeicher genutzt).

Durch diese Änderungen (Daten jetzt im AC) können der erste und letzte Befehl des OR Unterprogramms entfallen, und die mittleren Befehle können so bleiben, wie sie waren. Wir müssen noch einen Rücksprung an die aufrufende Stelle in das Unterprogramm einbauen; er ersetzt den überflüssig gewordenen letzten Befehl.

Den ersten Befehl des Unterprogramms löschen wir nicht aus dem Speicher; wir springen ihn nur nicht mehr an. Den letzten Befehl überschreiben wir mit dem benötigten Sprungbefehl. Unser Hauptprogramm braucht eine 0 als Datenkonstante; diese wird in Speicherstelle 3 zur Verfügung gestellt. Das Hauptprogramm lassen wir bei Speicherstelle 30 beginnen und wir haben dann folgende neuen Speicherbelegungen:

3	0	Konstante 0
:	:	
24	JUMP*	0 Rücksprung an die Adresse in IX!
:	:	

30	STORE	0	Diese zwei Befehle speichern Blockgröße
31	LDIX	0	n in IX (als Zeiger und Zähler)
32	JMPZ	40	Wenn keine Daten ($n=0$), nichts tun!
33	LOAD	3	$AC \leftarrow 0$, initialisiere das Oder mit 0
Jetzt beginnt die Ausführungsschleife:			
34	STORE	1	Bewahre Zwischenergebnis in M[1] auf
35	LOAD*	99	Effektive Adresse $99 + n$ ist Adresse der
			letzten unverarbeiteten Stelle im Block!
36	STIX	0	Rette Zählerwert in M[0]
37	JSETX	17	Springe OR Unterprogramm an,
			hinterlasse Rücksprungadresse 38 in IX
38	LDIX	0	Nach Rücksprung Zähler wiederherstellen
39	DXJNZ	34	Zähler herunterzählen, nochmal durch
			die Schleife wenn nicht 0
40	Programm fertig

Die Art, wie hier die Schleife realisiert wurde, ist durchaus üblich und realistisch. Die Methode für den Unterprogrammaufruf durch Hinterlassen der Rücksprungadresse in einem Indexregister ist von historischem Interesse (zum Beispiel hat die IBM 7094 Unterprogrammaufrufe auf diese Weise ermöglicht), aber heute wird meistens eine andere Methode verwendet, die nicht darunter leidet, dass die Anzahl der Indexregister begrenzt ist.

Heute werden Unterprogrammaufrufe mittels eines ***Stapelspeichers*** realisiert. Ein ***Stapel*** oder ***Stack*** ist nichts anderes als ein großer Speicherblock, der Daten aufnehmen kann. Zu jedem Stapel gibt es einen ***Stapelzeiger*** (***stack pointer***), ähnlich zum Programmzeiger, der auf die nächste freie Stelle im Stapel zeigt.

Für den Stapel gibt es spezielle Speicher- und Ladebefehle. Der Speicherbefehl heißt **PUSH** und speichert das angesprochene Datenwort an die vom Stapelspeicher adressierte freie Stelle; dann wird der Stapelspeicher inkrementiert (oder dekrementiert, je nachdem in welcher Richtung der Stapel wächst) um erneut auf die nächste *noch* freie Stelle zu zeigen.

Der Befehl, um Daten vom Stapel zu holen, heißt **POP**. Er dekrementiert (oder inkrementiert) den Stapelzeiger, um auf die letzte belegte Stelle zu zeigen, und liefert dann den Inhalt dieser Stelle. Wegen der Stapelzeigerveränderung wird durch das Holen die angesprochene Stapelstelle

wieder frei.

Der Name Stapelspeicher rührt daher, dass das Prinzip so funktioniert wie ein Tellerstapel in einer Kantine oder einer Großküche, wo eine Feder dafür sorgt, dass der oberste Teller immer auf gleicher Höhe ist. Kommen neue saubere Teller auf den Stapel, wird der bisherige Inhalt nach unten gedrückt; wird der oberste Teller entnommen (und andere kann man nicht entnehmen!), kommt der Stapel wieder hoch und es gibt einen neuen obersten Teller.

Die Speicher- und Holreihenfolge eines Stapels funktioniert nach dem LIFO-Prinzip („last in first out“): Daten werden in der umgekehrten Reihenfolge entnommen, in der sie heraufgepackt wurden, und das zuletzt gespeicherte Datum wird als Erstes wieder herausgegeben.

Für Unterprogrammaufrufe gibt es einen **Call Stack**, der nur zu diesem Zweck genutzt wird. Ein Unterprogrammaufruf rettet vor dem Sprung ins Unterprogramm den Programmzählerstand wie mit einem PUSH auf den Stapel, und der Rückkehrsprung aus dem Unterprogramm holt seine Sprungadresse wie mit einem POP vom Stapel. Das LIFO Prinzip sorgt wie eine verschachtelte Klammerung dafür, dass die Rücksprünge nicht durcheinander kommen und dass auch für verschachtelte Unterprogrammaufrufe jedes Unterprogramm zur zu seinem Aufruf passenden Rücksprungadresse zurückspringt.

Auf unserer kleinen CPU reichte aber weder die Anzahl der verfügbaren Operationscodes noch der Speicher dazu, einen sinnvollen Stapel zu implementieren, so dass wir die ältere Technik gewählt haben. Das war gerechtfertigt, weil die Mittel dazu auch andere Zwecke erfüllen.

- c) Als letztes Beispiel schreiben wir ein neues Programm, dass die Anzahl der Einsbits im Akkumulator zählt und in M[0] speichert.

Neben dieser Verwendung von Speicherstelle 0 benutzen wir Speicherstelle 1 als sicheren Aufbewahrungsort für den AC, und wir brauchen drei Konstanten: eine 0, abgelegt in Speicherstelle 3, eine 16 (Anzahl der Bits im AC Register), abgelegt in Speicherstelle 4, und eine 1, abgelegt in M[5].

Das Programm lassen wir an der Adresse 6 beginnen.

Die Konstanten:		
3	0	Konstante 0
4	16	Konstante 16
5	1	Konstante 1

		Programmbeginn mit Vorbereitungen:
6	STORE 1	Rette AC in M[1]
7	LOAD 3	Diese beiden Befehle initialisieren
8	STORE 0	die Zählung in M[0] auf 0
9	LDIX 4	Initialisiere Schleifenzähler auf 16
		Jetzt beginnt die Ausführungsschleife:
10	LOAD 1	Hole letzten AC Zustand
11	RSHIFT	Rotiere um eine Bitstelle
12	STORE 1	Rette AC in M[1]
13	AND 5	schneide rechtes Bit heraus
14	ADD 0	und addiere zur Zählung
15	STORE 0	speichere neuen Zählwert
16	DXJNZ 10	Zähler herunterzählen, nochmal durch
		die Schleife wenn nicht 0
17	Programm fertig

Kapitel 5

Realisierung und Regelung von Vorgängen und Prozessen

In diesem Kapitel wollen wir, ein wenig abstrakter als bisher, einige Designaspekte für elektronische Rechner und andere automatisierte Maschinen besprechen.

Zunächst wird es darum gehen, für einen einfachen automatischen Vorgang eine Maschine zu entwerfen, die diesen Vorgang mit Hilfe von digitaler Logik ausführen kann. Unser Augenmerk dabei wird nicht so sehr auf *universelle* Rechner gerichtet sein, wie die Computer, auf die wir uns bisher konzentriert haben und die viele verschiedene Aufgaben mit allgemeinen Techniken erfüllen können, sondern mehr auf Automaten mit einer einzelnen, speziellen und einigermaßen eingeschränkten Aufgabe.

Beispiele für solche Maschinen sind Getränkeautomaten, Fahrkartenautomaten und andere Verkaufsautomaten, Haushaltsmaschinen wie Waschmaschinen, Wäschetrockner, Herde, Fernseher oder DVD-Rekorder, die Computer gesteuert sind, oder etwas näher an unserem klassischen Thema wichtige Peripheriegeräte wie Drucker, Modems usw. Diese Geräte sind alle darauf spezialisiert, eine kleine Anzahl von Operationen auszuführen unter Kontrolle von eingehenden Signalen von externen Quellen, darunter auch die von menschlicher Hand eingegebenen Stellungen verschiedener Knöpfe und Einstellräder.

Wenn wir die Aufgaben ein wenig vereinfachen, können wir lernen, wie man effiziente digitale Logikschaltungen entwerfen kann, die genau auf die gewünschte Weise auf die Eingaben reagieren.

Anschließend werden wir kurz auf das (im allgemeinen sehr schwierige) Thema eingehen, wie man kontrollieren und beweisen kann, dass ein Computerprogramm oder ein Regelungsmechanismus so funktioniert, wie vorgesehen, d. h., dass unter *allen möglichen Einstellungen und Eingaben* die vorge-

sehene Wirkung eintritt. Ein solcher Nachweis kann lebenswichtig sein, wenn die programmierte Maschine ein Verkehrsmittel oder ein medizinisches Gerät ist, um nur einige erdenkliche Gefahrensituationen zu nennen. Und auch ein Geldautomat, der die falsche Anzahl von Scheinen herausgibt, kann sehr unangenehme Konsequenzen nach sich ziehen.

Ein weiteres Thema dieses Kapitels werden parallele Prozesse sein, die in der Computerwelt eine immer wichtigere Rolle spielen, weil sie die effektivste Methode bilden, Rechenabläufe zu beschleunigen. Gleichzeitig ablaufende parallele Vorgänge, die miteinander zusammenspielen, erfordern eine aufwendige Organisation, um sicher zu gehen, dass jeder Prozess gültige Daten erhält und den Ablauf der anderen Prozesse nicht stört.

Wir beginnen mit dem Problem des Entwurfs von effizienten automatischen Maschinen. David Huffman hat 1954 ein sehr allgemeines Modell für **sequentielle Automaten** vorgeschlagen, d. h., für Maschinen, die schrittweise Eingangsdaten verarbeiten und Ausgangsdaten herausgeben *in Abhängigkeit von den Eingängen und dem bisherigen Ablauf*.

Der bisherige Ablauf wird von der Maschine in der Gestalt von Zustandswerten in einem Speicher aufbewahrt, und diese Zustandswerte werden nach jedem Durchlauf neu geschrieben.

Dieses Modell wird umrissen in Abbildung 5.1 auf der nächsten Seite. Der große Kasten enthält kombinatorische Schaltkreise, die aus den Eingangssignalen e_1, \dots, e_n und den bisherigen Speicherwerten z_1, \dots, z_p die Ausgangssignale a_1, \dots, a_m und neue Speicherwerte Z_1, \dots, Z_p berechnen. Diese Schaltkreise haben verschiedene Durchlaufzeiten und werden deshalb von einem Taktsignal synchronisiert, das auch das Setzen der Speicherwerte steuert, so dass die ganze Arbeit der Maschine in geregelter Weise sequentiell, also in aufeinander folgenden Schritten, vor sich geht.

Die Speicherinhalte kann man als gewisse interne *Zustände* der Maschine auffassen, die ihre Funktion steuern. Die praktische Wirkung ist so, als würde die Maschine einen Satz von Regeln implementieren der Form: „Wenn du in Zustand z bist und Eingabe e erhältst, dann gebe a aus und gehe in Zustand Z über.“

Diese Regeln kann man mit verschiedenen Mitteln mathematisch beschreiben.

Eine Art der Beschreibung geschieht durch eine Funktion. Man hat einen endlichen Satz I von 2^n möglichen verschiedenen Eingaben, einen endlichen Satz A von 2^m möglichen Ausgaben, und einen endlichen Satz Z von 2^p möglichen inneren Zuständen, und in jedem Schritt, den der Rechner ausführt, werden die Eingaben und Zustände ersetzt durch Ausgaben und neue Zustände, die sich aus den Eingangsdaten ergeben als die Werte einer festen

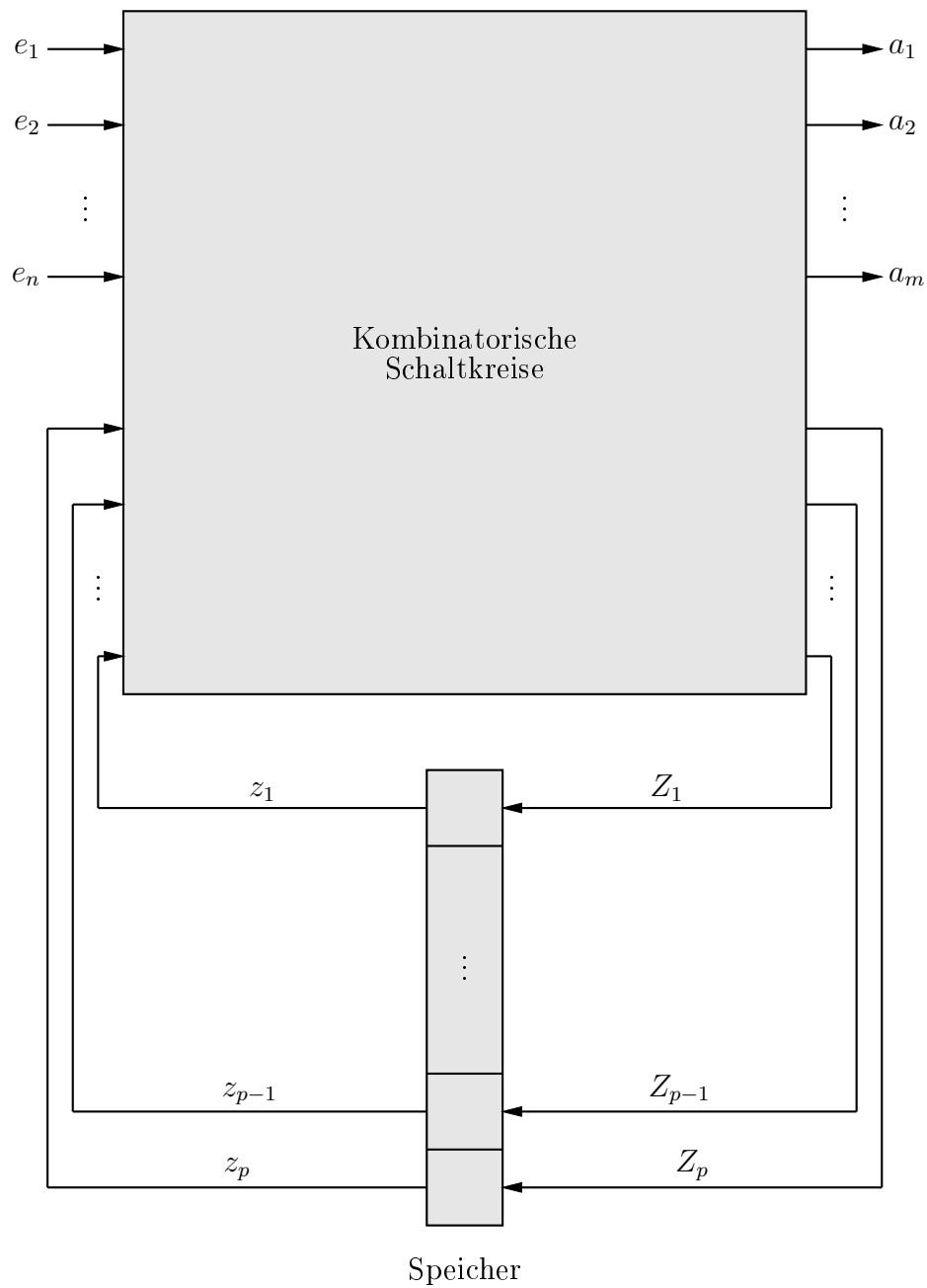


Abbildung 5.1: Das Huffman Modell eines sequentiellen Automaten

Funktion

$$f: I \times Z \longrightarrow A \times Z.$$

Obwohl diese Funktion die Arbeitsweise des Rechners genau beschreibt, ist sie nicht sehr praktisch, denn die Details des Huffman Modells, und insbesondere die Zustandsmenge, müssen bei einer Anwendung nicht eindeutig feststehen.

In Anwendungen ist es sogar oft so, dass die Eingaben und Ausgaben nicht durch binäre Signale gegeben sein müssen, oder dass manche Eingaben oder Zustände nur unter bestimmten Umständen vorkommen können, so dass die Gesamtanzahl der möglichen Eingaben oder der zulässigen Zustände nicht unbedingt eine Zweierpotenz sein muss. Solche Situationen können wir trotzdem binär kodieren, aber aus verschiedenen Gründen müssen nicht alle Eingaben tatsächlich vorkommen können. Das bedeutet, dass die Funktion f eventuell nur auf einer *Teilmenge* von $I \times Z$ definiert ist.

Das ist nicht weiter schlimm; wenn es nötig ist, können wir so tun, als wäre die Funktion für alle Eingangswerte definiert, weil ihr „fingierter“ Wert an den Stellen, wo sie nicht wirklich definiert ist, keine praktische Auswirkung hat und die Wirkungsweise der beschriebenen realen Maschine nicht beeinflusst.

Wir haben die Schritte der Huffman Maschine als die Anwendung eines Satzes von Regeln beschrieben. Im gleichen Sinn ist es oft hilfreich, sich die einzelnen Wertezuordnungen

$$f(e, w) = (a, z)$$

von f als anzuwendende „Regeln“ vorzustellen für den Übergang von Eingangswerten zu Ausgangswerten und von alten Zuständen zu neuen Zuständen.

Die praktische Aufgabe, die sich uns stellt, ist folgende: wir wollen für einen speziellen Zweck einen Automaten bauen, der mit gewissen Eingaben konfrontiert sein soll und darauf unter verschiedenen möglichen Umständen auf eine bestimmte passende Art reagieren soll. Wie kann man einen effizienten Schaltkreis entwerfen, der das gewünschte Verhalten zeigt?

Eine direkte Lösung dieser Aufgabe ist nicht ganz sichtbar, aber die Aufgabe wird leichter, wenn wir zwischen der Startposition (gewünschtes Verhalten des Automaten) und der Zielposition (effizienter Schaltkreis) eine weitere Stufe einbauen. Als diese mittlere Stufe könnte sich das Huffman Modell eignen, wenn wir eine gute Möglichkeit hätten, die Details eines passenden Huffman Modells aus den Vorgaben herzuleiten, und wenn wir eine gute Möglichkeit hätten, aus den Details des Huffman Modells einen Schaltkreisentwurf zu gewinnen.

Die erste Teilaufgabe lässt sich mit Hilfe einer bequemer Darstellung des Huffman Modells bequem lösen. Diese bequemere Darstellung nennt sich ein **Zustandsgraph**.

Zunächst ein bisschen mathematische Terminologie. Ein **Graph** (genauer: ein **markierter gerichteter Graph**)

$$G = (V, E, M, \mu),$$

besteht aus einer Menge V , deren Elemente auf englisch **vertices** und auf deutsch **Knoten** heißen, aus einer Teilmenge

$$E \subseteq V \times V,$$

deren Elemente auf englisch **directed edges** und auf deutsch **gerichtete Kanten** heißen, aus einer Menge M von **Markierungen** (oder englisch **markers**), und aus einer Zuordnung

$$\mu: E \longrightarrow \mathcal{P}(M) \setminus \emptyset,$$

der **Markierungsfunktion**, die jeder Kante eine nichtleere Menge von Markierungen zuordnet.

Wir sagen, dass eine Kante $e = (v_1, v_2)$ die Knoten v_1 und v_2 **verbindet**, oder dass e eine Kante **von** v_1 **nach** v_2 ist. Wir nennen v_1 den **Anfangspunkt** und v_2 den **Endpunkt** oder **Schlusspunkt** von e , oder manchmal nennen wir v_1 und v_2 zusammen die **Endpunkte** von e .

Mathematisch haben die genannten Mengen genau die hier genannte sehr einfache Struktur und ihre Elemente sind trotz der suggestiven Namen keine geometrischen Objekte, aber beim Umgang mit Graphen stellt man sie dann doch durch Zeichnungen dar, in denen die Knoten durch Punkte repräsentiert werden und die Kanten (v_1, v_2) durch einen Pfeil dargestellt werden, der vom Punkt v_1 zum Punkt v_2 führt. Jede Kante (v_1, v_2) wird in der Zeichnung beschriftet mit allen Elementen der Teilmenge $\mu(v_1, v_2)$ von M .

Beachten Sie, dass man auf Papier zwischen zwei Punkten mehrere verschiedene Pfeile zeichnen kann, aber dass in einem Graphen die Kanten durch ihren Anfangs- und Schlusspunkt eindeutig bestimmt sind, und deshalb kann es in der Zeichnung eines Graphen zwischen zwei Knotenpunkten in jeder Richtung höchstens einen Pfeil geben.

Aus der Funktion $f: I \times Z \longrightarrow A \times Z$ eines Huffman Modells kann man wie folgt einen Zustandsgraphen G konstruieren, der die Funktion beschreibt und aus dem man die Funktion wiedergewinnen kann. Wir haben schon bemerkt, dass diese Funktion nicht überall definiert sein muss, und dass für praktische Anwendungen es fast die Regel ist, dass sie nicht für alle Paare $(e, z) \in I \times Z$

definiert ist, und deshalb gehen wir davon aus, dass f vielleicht nur auf einer geeigneten Teilmenge

$$R \subseteq I \times Z$$

definiert ist. Bei praktischen Anwendungen besteht R aus den für die jeweilige Aufgaben „relevanten“ Kombinationen von Eingaben und Zuständen. Nichtrelevante Kombinationen sind entweder physikalisch oder logisch ausgeschlossen oder es spielt für die Lösung der Aufgabe keine Rolle, welche Werte dort angenommen werden, so dass wir für diese Kombinationen auch keine speziellen Vorkehrungen treffen müssen.

Als *Knoten* von G wählen wir alle Zustände $z \in Z$, die entweder in R genannt werden oder in den Werten, die f auf R annimmt:

$$V := \{z \in Z \mid \text{es gibt } e \in I \text{ mit } (e, z) \in R \text{ oder es gibt } e \in I, w \in Z \text{ und } a \in A \text{ mit } (e, w) \in R \text{ und } f(e, w) = (a, z)\}$$

Insbesondere ist $V \subseteq Z$.

Als *Kanten* von G wählen wir die *Zustandsübergänge*, die unter den „Regeln“ von f vorkommen, also die durch die Anwendung von f bewirkt werden können, wenn die Eingaben passend sind. Etwas genauer und mathematischer ausgedrückt:

$$E := \{(w, z) \in V \times V \mid \text{es gibt } e \in I \text{ und } a \in A \text{ mit } (e, w) \in R \text{ und } f(e, w) = (a, z)\}.$$

Jede Kante *markieren* wir mit den *Wertübergängen*, die in den Regeln vorkommen, die diese Kante als ihr Zustandsübergang bestimmen. In anderen Worten,

$$M := \{(e, a) \in I \times A \mid \text{es gibt } w \text{ und } z \in Z \text{ mit } (e, w) \in R \text{ und } f(e, w) = (a, z)\}$$

und für jede einzelne Kante (w_0, z_0) ist

$$\mu(w_0, z_0) = \{(e, a) \in I \times A \mid (e, w_0) \in R \text{ und } f(e, w_0) = (a, z_0)\}$$

Damit sind alle Bestandteile des f zugeordneten Zustandsgraphen genau festgelegt. Zum besseren Verständnis fassen wir die Definition zusammen und wiederholen sie etwas informeller:

Die Funktion f beschreibt wie aus gewissen Eingaben in Kombination mit gewissen Zuständen eine Ausgabe in Kombination mit einem neuen Zustand entsteht. Jede Zustandsveränderung, die so entsteht, beschreiben wir

mit einer gerichteten Kante eines Graphen, die vom alten Zustand zum neuen Zustand weist; diese Zustände müssen somit automatisch Knoten des Graphen sein. Bei dem Zustandsübergang wird auch ein Eingangswert in ein Ausgangswert verwandelt, und wir markieren die Kante mit diesem Wertepaar; gibt es mehrere Wertezuordnungen, die zu der gleichen Kante gehören, so markieren wir die Kante mit *allen*.

Genau so einfach wie die Konstruktion des Graphen aus f ist es, aus dem fertigen Graphen G die Funktion f wieder abzulesen.

Jede Kanten*markierung* von G (also jede Kombination einer Kante mit einer Markierung, die sie trägt) gehört zu einer Regel von f . Und zwar: Wenn eine Kante (w, z) eine Markierung $(e, a) \in I \times A$ trägt, dann hat f eine Regel

$$f(e, w) = (a, z).$$

Jede weitere Markierung der gleichen Kante liefert eine weitere Regel von f , und wenn wir diese Regeln für alle Kanten von G zusammentragen, dann haben wir alle Regeln von f gesammelt, und f und ihr Definitionsbereich sind genau bestimmt.

Wir illustrieren die Zuordnung eines Graphen zu einer Funktion und umgekehrt an einem Beispiel.

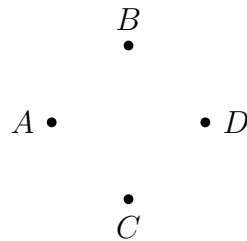
In diesem Beispiel haben wir vier Zustände A, B, C und D , vier mögliche Eingaben a, b, c und d und vier mögliche Ausgaben w, x, y und z . Wir haben es vorgezogen, symbolische Namen (also Buchstaben) zu verwenden, aber die Anzahl der Elemente in jeder Klasse ist eine Zweierpotenz $4 = 2^2$ und wir könnten genau so gut diese Größen binär kodieren, jeweils mit 2 Bits. Später brauchen wir die binäre Kodierung, aber im Moment behalten wir die symbolischen Bezeichnungen bei, weil das die Lesbarkeit und Verständlichkeit erhöht.

Die Funktion f soll folgende Werte annehmen, wobei wir nur die „relevanten“ Werte in der Tabelle aufführen. Andere Eingabe-Zustandskombinationen sind entweder nicht möglich oder es spielt keine Rolle, wie der Automat darauf reagiert (wir sagen dazu: die anderen Fälle sind „don’t care“ Fälle).

e	z^{alt}	a	z^{neu}
a	B	y	D
b	D	y	C
c	A	w	B
c	C	z	A
d	B	x	D

Die Knoten des zu f gehörenden Zustandsgraphen G sind alle Zustände, die in dieser Tabelle in der zweiten oder in der vierten Spalte erscheinen,

und das sind in diesem Fall alle Zustände in unserem Vorrat. Hier also die Knoten:



Jede Regel, also jede Zeile der Wertetabelle von f , wie wir sie angegeben haben, produziert als Kante einen Pfeil vom Zustand in der zweiten Spalte zum Zustand in der vierten Spalte, markiert mit dem Paar von Werten aus der ersten und dritten Spalte. Gerichtete Kanten, die mehrmals vorkommen, werden nur einmal gezeichnet, aber mit allen Markierungen versehen, die man aus der Wertetabelle für diese Kante erhält.

Das liefert dann folgenden Zustandsgraphen:

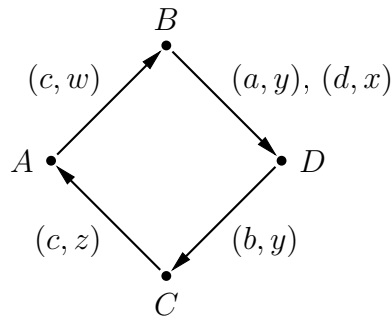


Abbildung 5.2: Der Zustandsgraph von f

Wie auf der vorherigen Seite erklärt, kann man aus diesem markierten Graphen direkt wieder die Wertetabelle von f ablesen.

Solche Graphen haben verschiedene Anwendungen. Wir werden sie gleich aus Vorgaben für das gewünschte Verhalten eines Automaten gewinnen, aus ihnen die zugehörige Funktion ablesen und zeigen, wie man diese Funktion mit digitaler Logik implementieren kann, aber das ist nicht alles, was man mit einem Zustandsgraphen machen kann.

Wenn man sich durch einen solchen Graphen, den Pfeilen folgend, von Knoten zu Knoten bewegt, so sind nur gewisse Wege möglich, die bildlich in dem Graphen gut sichtbar und erkennbar sind. Diese Wege lassen dann nur gewisse Folgen von Eingaben zu, und produzieren aus ihnen gewisse Folgen von Ausgaben.

Dieses „Verhalten“ eignet die Graphen dazu, Sprachen und Sprachprozesse zu beschreiben. Die Pfeilwege durch den Graphen bestimmen die Syntax der Sprache, also welche Folgen von Buchstaben oder anderen Sprachelementen grammatisch richtige Aussagen darstellen. Die Ausgabenfolgen beschreiben sprachliche Transformationen, wie sie in der modernen Theorie der Sprachherzeugung vorkommen, und wie sie von Algorithmen ausgeführt werden müssen, die „Übersetzungen“ im weitesten Sinne vornehmen (und sei es die Übersetzung einer höheren Programmiersprache in Assemblersprache oder die Übersetzung zwischen verschiedenen Formelnotationen, wie sie für symbolische Mathematikprogramme wichtig sein können).

Dieses Thema lassen wir aber jetzt und erinnern uns an den zweiten Teil unseres auf Seite 190 skizzierten Vorhabens für den Entwurf von Automaten. Mit Zustandsgraphen kann man aus den Vorgaben für einen Automaten eine Beschreibung einer Huffman Maschine gewinnen; wie realisiert man diese Huffman Maschine mit digitaler Logik?

Was man in Prinzip machen muss, wissen wir schon, denn wenn wir die Eingaben, Ausgaben und Zustände binär kodieren liefert uns die Funktion f , die die Huffman Maschine beschreibt, eine Wertetabelle für eine boolesche Funktion, und wir haben in Kapitel 2 schon gelernt, wie man boolesche Funktionen effizient mit digitaler Logik berechnet.

In der jetzigen Situation gibt es aber noch eine Besonderheit: wir sind ja davon ausgegangen, dass f nicht für alle Eingabe-Zustandskombinationen definiert sein muss, also dass nicht die ganze Wertetabelle der booleschen Funktion festgelegt ist, sondern nur einige Zeilen. Uns ist egal, wie die anderen Zeilen aussehen, d. h., wie die vollständig definierte boolesche Funktion aussieht, die unter den relevanten Umständen die gleichen Werte wie f hat. Das gibt uns eine gestalterische Freiheit, die man benutzen kann, um die boolesche Funktion und die sie realisierende digitale Logik möglichst einfach zu halten.

Wir illustrieren diesen Gedanken und das allgemeine Vorgehen anhand des Beispiels, das wir gerade besprochen haben. Als Erstes müssen wir die Wertetabelle für f auf Seite 193 binär kodieren, damit wir einen Satz von booleschen Funktionen daraus entnehmen können.

Dazu zählen wir die verwendeten Buchstaben in jeder Gruppe (Eingaben, Ausgaben, und Zustände) binär durch und benutzen die Bits der Binärnumerierung als binären Wert dieser Symbole. Die Anzahl der Bits, also der binären Signale, die man braucht, um eine Symbolengruppe mit k Elementen zu kodieren, ist

$$\lceil \log_2 k \rceil.$$

Das ist der Zweierlogarithmus von k , aufgerundet auf die nächste mindestens

so große ganze Zahl.

In unserem Beispiel hat jede Gruppe vier Elemente und wir benötigen für jede Gruppe 2 Bits. Tabelle 5.1 zeigt eine mögliche Kodierung.

Zustände	Eingaben	Ausgaben	Code
A	a	w	00
B	b	x	01
C	c	y	10
D	d	z	11

Tabelle 5.1: Kodierung der Eingaben, Ausgaben und Zustände

Mit dieser Kodierung können wir die Wertetabelle von f wie folgt binär schreiben:

e_1	e_2	z_1^{alt}	z_2^{alt}	a_1	a_2	z_1^{neu}	z_2^{neu}
0	0	0	1	1	0	1	1
0	1	1	1	1	0	1	0
1	0	0	0	0	0	0	1
1	0	1	0	1	1	0	0
1	1	0	1	0	1	1	1

Tabelle 5.2: Die binäre Kodierung der Funktion f

Wir müssen als Nächstes für diese Funktion, oder genauer für die vier booleschen Funktionen in den Spalten rechts vom Doppelstrich, optimierte boolesche Ausdrücke finden (die wir dann durch digitale Logik realisieren könnten).

Wir betrachten a_1 . Die Bedingungen, unter denen in der a_1 Spalte eine Eins entsteht, führen uns zu der kanonischen disjunktiven Normalform

$$a_1 = \bar{e}_1 \bar{e}_2 \overline{z_1^{\text{alt}}} z_2^{\text{alt}} + \bar{e}_1 e_2 z_1^{\text{alt}} \overline{z_2^{\text{alt}}} + e_1 \bar{e}_2 z_1^{\text{alt}} \overline{z_2^{\text{alt}}},$$

die sich mit den Methoden aus Kapitel 2 nicht vereinfachen lässt.

Aber diese Formel wurde hergeleitet unter der Annahme, dass die Einsen, die in Tabelle 5.2 sichtbar sind, die einzelnen Einswerte von a_1 sind, d. h., dass in allen unsichtbaren Zeilen der vollständigen Wertetabelle eine 0 in dieser Spalte steht. Da uns die Werte von a_1 außerhalb des sichtbaren Bereiches aber egal sind, gibt es keinen Grund, an dieser Annahme festzuhalten.

Man kann für a_1 eine einfachere Formel finden, wenn man in der *sichtbaren* Tabelle eine möglichst kleine Anzahl von möglichst einfachen Bedingungen

sucht, die in diesem Teil der Tabelle mit dem Auftreten von Einsen in der a_1 Spalte zusammenfallen.

Durch Inspektion sehen wir, dass $a_1 = 1$ in den einzigen beiden Zeilen, in denen $e_1 = 0$, und dass der von dieser Bedingung nicht erfasste Einseintrag in der a_1 Spalte in der einzigen Zeile steht, in der $e_1 = z_1^{\text{alt}} = 1$. Eine viel einfachere gültige Formel für a_1 wäre also

$$a_1 = \bar{e}_1 + e_1 z_1^{\text{alt}}.$$

Diese Formel ist auch für andere Kombinationen aus Eingabe und Zustand Eins, aber nur in Situationen, die für diese Aufgabe nicht relevant sind, wo uns also der Wert von a_1 nicht wichtig ist.

Noch eindeutiger ist die Darstellung der booleschen Funktionen

$$z_1^{\text{neu}} = z_2^{\text{alt}} \quad \text{und} \quad z_2^{\text{neu}} = \overline{z_1^{\text{alt}}}$$

Die direkt aus der Tabelle nach unserer klassischen Methode direkt abzulesende DNF ist in beiden Fällen wesentlich komplizierter, und lässt sich mit unseren klassischen Mitteln nicht vereinfachen.

Die optimalen booleschen Funktionen, die man für die Ausgaben und die neuen Werte der Zustände findet, lassen sich direkt in digitale Logik übersetzen, aber auch hier ist manchmal eine zusätzliche Einsparung möglich. Wenn zufällig für einen der Zustände der neue Wert gleich dem alten Wert ist, dann braucht man diesen Wert nicht neu einzuspeichern, sondern kann ihn einfach auf dem alten Stand belassen; auch das würde zur Vereinfachung der endgültigen Schaltung beitragen.

Beispiel 5.1 Als „praktisches“ Beispiel wollen wir jetzt einen einfachen Cola Automaten entwickeln. Der Automat verkauft nur Cola und es gibt keine Auswahl zwischen Getränken oder Ausgabeschächten, und die Cola hat einen festen Preis von 1 €.

Zur Bezahlung akzeptiert der Automat nur Euromünzen und 50-Cent-Münzen, und er gibt Wechselgeld, falls zu viel bezahlt wird.

Da der Automat nur eine Getränksorte verkauft, sind keine Auswahlknöpfe möglich. Ein Verkaufsvorgang wird initiiert, indem ein Kunde Geld einwirft. Die Maschine wird aber die meiste Zeit in Hab-Acht-Stellung sich befinden und prüfen, ob Geld eingeworfen wurde. Deshalb gibt es *drei* mögliche Eingaben:

Symbol	Eingabe
a	nichts
b	50 Cent
c	1 Euro

Auf die Eingaben reagiert die Maschine eventuell mit einer Dose Cola, wenn sie bezahlt ist, aber sonst mit einer Anzeige, wieviel noch nachzuzahlen ist. Wenn man zuerst 50 Cent und dann 1 € einwirft, kann man die Dose sogar überbezahlen, und dann hat die Maschine auf jeden Fall das vom Kunden eingeworfene 50-Cent-Stück und kann Wechselgeld geben.

Es gibt deshalb *vier* mögliche Ausgaben:

Symbol	Ausgabe
w	Anzeige „Noch 1 €“
x	Anzeige „Noch 50 ct“
y	Dose Cola
z	Dose Cola und 50 ct Wechselgeld

Während eines Verkaufsvorgangs können drei Situationen unterschieden werden, auf die die Maschine bei der Verarbeitung von Eingaben individuell reagieren muss. Sie bilden die Zustände des Automaten:

Zustand	Situation
A	Anfangszustand (Warten auf Kunden)
B	Zwischenzustand (Teilzahlung einer Cola)
C	Endzustand (Cola ist voll bezahlt)

Die Maschine beginnt in Zustand A , und solange nichts passiert bleibt sie in diesem Zustand und macht Anzeige w . Wenn 50 ct eingeworfen werden, geht der Automat in den Zwischenzustand B und ändert die Anzeige auf x . Wenn im Anfangszustand 1 € eingeworfen wird, geht die Maschine in den Endzustand C über und gibt eine Dose Cola heraus; genau das Gleiche passiert, wenn im Zwischenzustand 50 ct eingeworfen werden.

Ist die Maschine einmal im Zwischenzustand, dann bleibt sie in diesem Zustand mit der gleichen Anzeige x , solange nichts passiert. Sobald eine Münze eingeworfen wird, geht sie in Zustand C über, aber der Ausgang hängt davon ab, welche Münze eingeworfen wurde. War es eine 50 Cent Münze, gibt der Automat nur eine Cola heraus (wie wir oben schon sahen), aber wenn 1 € eingeworfen wurde, werden zusätzlich zu einer Dose Cola auch die zuerst eingeworfenen 50 Cent erstattet.

Nach dem Endzustand hört die Maschine nicht einfach auf zu funktionieren (es sei denn, sie fasst nur eine Dose und muss nach jedem Verkaufsvorgang gewartet werden, was etwas unrealistisch wäre). Vielmehr beginnt sie dann von neuem auf Kunden zu warten. Wir gehen davon aus, dass der Automat im Endzustand sich kurz umstellen muss (durch Nachrutschen der Dosen im Ausgabeschacht und andere Vorbereitungen), bevor sie bereit ist, wieder Kunden zu bedienen, und modellieren das so, dass die Maschine im

Endzustand keine Münzeingaben annimmt, aber wenn *keine* Eingabe kommt wieder in den Anfangszustand übergeht.

Diese etwas plaudernde Erzählung der beabsichtigten Funktionsweise des Automaten lässt sich direkt und mühelos in folgenden Zustandsgraphen übertragen:

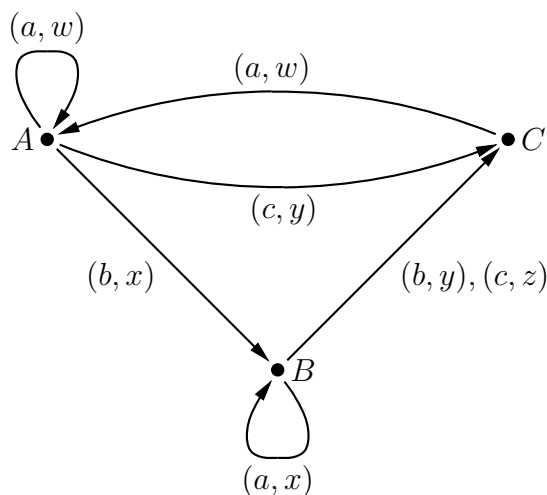


Abbildung 5.3: Der Zustandsgraph des Cola-Automaten

Jetzt müssen wir nur noch aus dem Zustandsgraphen eine geeignete boolesche Funktion herleiten, die für die gegebenen Situationen die richtige Reaktion bewirkt. Wir gehen vor, wie beginnend auf Seite 194 erläutert wurde und schon einmal an einem Beispiel vorgeführt wurde.

Aus dem Zustandsgraphen lesen wir folgende symbolische Wertetabelle für die beschreibende Funktion f ab:

e	z^{alt}	a	z^{neu}
a	A	w	A
b	A	x	B
c	A	y	C
a	B	x	B
b	B	y	C
c	B	z	C
a	C	w	A

Dann wählen wir eine binäre Kodierung für die Eingaben, Ausgaben und Zustände. Für jede Gruppe werden wieder 2 Bits benötigt, da jede Gruppe drei oder vier Elemente hat. Wir kodieren die Daten wie folgt:

Zustände	Eingaben	Ausgaben	Code
A	a	w	00
B	b	x	01
C	c	y	10
		z	11

Mit dieser Kodierung können wir die Wertetabelle von f wie folgt binär schreiben:

e_1	e_2	z_1^{alt}	z_2^{alt}	a_1	a_2	z_1^{neu}	z_2^{neu}
0	0	0	0	0	0	0	0
0	1	0	0	0	1	0	1
1	0	0	0	1	0	1	0
0	0	0	1	0	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	1	1	0
0	0	1	0	0	0	0	0

Tabelle 5.3: Die binäre Kodierung der Funktion f für den Cola-Automaten

Durch Betrachtung der Tabelle finden wir leicht kurze Formeln für die Ausgaben und Neuzuständen, die für die gegebenen Eingaben und Altzustände das richtige Verhalten zeigen (für andere Situationen ist das Verhalten egal):

$$\begin{aligned}
 z_1^{\text{neu}} &= a_1 = e_1 + e_2 z_2^{\text{alt}} \\
 a_2 &= \bar{e}_2 z_2^{\text{alt}} + e_2 \overline{z_2^{\text{alt}}} \\
 z_2^{\text{neu}} &= e_2 \overline{z_2^{\text{alt}}} + \bar{e}_1 \bar{e}_2 z_2^{\text{alt}}
 \end{aligned}$$

Die elektronische Steuerung der Colamaschine könnte man direkt aus diesen Formeln bauen. Wir sehen, dass sie nur wenige Gatter braucht.

Ein wichtiges Thema bei der Entwicklung jeder Art von automatisierten Vorgängen, egal, ob es sich um durch Hardware gesteuerte Geräte handelt oder um Computerprogramme, die Geräte steuern, ist der Nachweis, dass das Gerät unter allen erdenklichen Umständen so funktioniert, wie vorgesehen war. Die Schäden, die entstehen, wenn Designfehler die Funktion beeinträchtigen, können gravierend sein, wie schon am Anfang dieses Kapitels ausgemalt, aber auch wenn sie nur ärgerlich sind, können sie bei einem kommerziellen Produkt sich negativ auf die Kundenzufriedenheit auswirken, mit unangenehmen Folgen für den Hersteller.

Ein allgemeiner Nachweis der richtigen Funktion eines beliebigen Programmes ist schon theoretisch nicht möglich, und für sehr große und komplizierte Programme auch wenn theoretisch möglich, dann aus praktischen Gründen nicht in einer akzeptablen Zeitspanne machbar, was aber nicht heißt, dass es *überhaupt* keine sinnvollen Möglichkeiten gibt, geeignete Programme oder Programmabschnitte mathematisch und theoretisch zu *verifizieren*.

Dazu gibt es viele Ansätze und Methoden, die zumindest in vielen Fällen funktionieren und eine große Hilfe beim Austesten von Programmen sein können (weil Menschen entweder die große Anzahl von auftretenden Fällen nicht erfassen können oder dazu neigen, Möglichkeiten zu übersehen). Wir haben zwar weder die Zeit noch die theoretische Grundlage, um uns sehr weit in dieses Thema einzuarbeiten, aber wir wollen doch einen sehr kurzen und oberflächlichen Einblick in einige Ideen gewinnen, die bei dieser Problematik zur Anwendung kommen.

Bei Korrektheitsbeweisen und der Verifikation von Algorithmen oder Programmen geht es um die Konstruktion von mathematischen Beweisen, dass ein vorgegebenes Programm eine bestimmte Aufgabe richtig erfüllt, und um die Entwicklung von Rechenmethoden, mit denen ein solcher Beweis anhand einer formalen Beschreibung des Programms geführt oder unterstützt werden kann. Wir werden hier ein paar elementare Methoden anhand von Beispielen vorführen.

Beispiel 5.2 Wir betrachten das Programm, das im Flussdiagramm 5.4 auf der nächsten Seite dargestellt wird:

Hier sind x und y *positive* natürliche Zahlen, und das Programm hat den Zweck, das Produkt xy zu berechnen und in der Variablen z zu hinterlassen.

Wir wollen beweisen, dass das Programm diesen Zweck erfüllt, genauer,

- dass das Programm irgendwann stoppen wird, indem es den unteren Ausgang nimmt (den mit „J“ markierten letzten Pfeil unten), und
- dass wenn das Programm stoppt, die Variable z den Wert xy hat.

Um diesen Nachweis zu führen, müssen wir geeignete Eigenschaften und Beziehungen der Variablen finden, die über alle Pfade im Flussdiagramm entweder erhalten bleiben, und am Ende implizieren, dass $z = xy$, oder die sich kontrolliert so verändern, dass damit der Nachweis gegeben ist, dass das Programm irgendwann stoppen muss (das kann man zum Beispiel mit einer Größe erreichen, die immer positiv ist, solange der untere Ausgang nicht genommen wird, und die ständig kleiner wird, so dass sie irgendwann nicht mehr positiv bleiben kann, wodurch der Ausgang erzwungen wird).

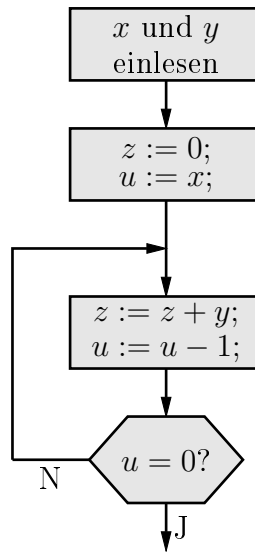
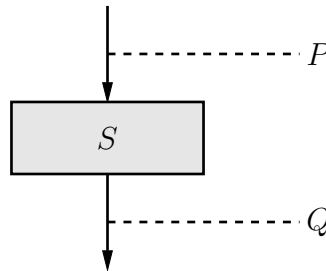


Abbildung 5.4: Flussdiagramm eines Multiplikationsprogramms

Für die Untersuchung dieser Zusammenhänge wollen wir eine geeignete Notation einführen. Sei S ein Ausführungsschritt in dem Programm oder dem Flussdiagramm, also der Inhalt eines der *rechteckigen* Kästen im Flussdiagramm.

Seien P und Q Bedingungen oder Relationen zwischen den Variablen im Programm. Die Notation



die wir abgekürzt auch schreiben als

$$\{P\}S\{Q\},$$

soll bedeuten, dass wenn Bedingung P gilt und dann Schritt S ausgeführt wird, gilt anschließend Bedingung Q . Wir nennen P hier die **Vorbedingung** und Q die **Nachbedingung**.

Man beachte, dass eine Bedingung, die vor einer Abfrage gilt (Abfragen werden im Flussdiagramm durch Kästen mit spitzen Seiten dargestellt) auch nach der Abfrage gilt, und zwar unabhängig vom Ergebnis der Abfrage. Die

ursprüngliche Bedingung wird durch die Abfrage nur verstärkt, denn es gilt das Und der ursprünglichen Bedingung und dem Ergebnis der Abfrage.

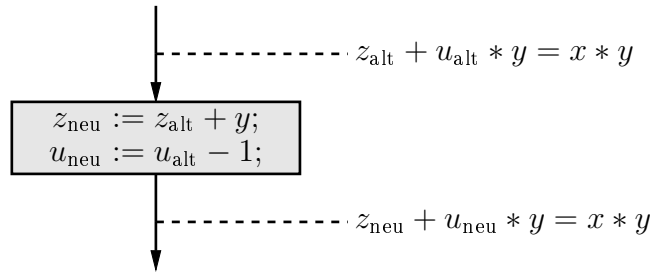
Nach dem zweiten Kasten von oben im Flussdiagramm, in dem

$$z := 0 \quad \text{und} \quad u := x$$

gesetzt wird, gilt offensichtlich (ohne Vorbedingung) die Nachbedingung

$$P : \quad z + u * y = x * y,$$

und für den Ausführungskasten S in der Schleife gilt



Denn wenn wir in der Nachbedingung für diesen Kasten die neuen Werte von z und u ersetzen durch die Formel in den alten Werten, die im Ausführungskasten angibt, wie die neuen Werte gesetzt werden, dann schreibt sich die Nachbedingung um als

$$\begin{aligned}
 z_{\text{neu}} + u_{\text{neu}} * y &= (z_{\text{alt}} + y) + (u_{\text{alt}} - 1) * y \\
 &= z_{\text{alt}} + y + u_{\text{alt}} * y - y \\
 &= z_{\text{alt}} + u_{\text{alt}} * y \\
 &= x * y \quad (\text{nach der Vorbedingung})
 \end{aligned}$$

Also gilt für diesen Ausführungskasten: $\{P\}S\{P\}$.

Diese Bedingung wird für den Nachweis verantwortlich sein, dass das Programm tatsächlich richtig multipliziert, wenn es anhält. Aber wir müssen auch beweisen, *dass* es anhält.

Dazu betrachten wir Bedingungen für die Variable u . Zunächst schauen wir uns den genauen Wert von u an. Offensichtlich gilt für jede natürliche Zahl k , dass

$$\{u = k\}S\{u = k - 1\},$$

denn u wird im Kasten S um Eins vermindert. (Beachten Sie bitte, dass diese Formel eine „wenn, dann“ Beziehung ausdrückt: *wenn* vor dem Kasten $u = k$, *dann* gilt nach dem Kasten $u = k - 1$. Aber sie behauptet *nicht*, dass für

eine vorgegebene Zahl k tatsächlich gilt $u = k$, oder dass später, nach dem Kasten, $u = k - 1$.)

Als Konsequenz dieser Beziehung haben wir die etwas gröbere und schwächere Beziehung

$$\{u > 0\}S\{u \geq 0\}.$$

Um den Korrektheitsbeweis zu führen, müssen wir im *kompletten* Flussdiagramm jeden Pfeil mit einer Bedingung (oder einem Satz von Bedingungen) versehen, die an der Quelle des Pfeils als Nachbedingung bewertet wird und am Ziel des Pfeils als Vorbedingung bewertet wird, und diese Bedingungen müssen so zueinander passen, dass für jeden Ausführungs- oder Abfragekasten Vorbedingung, Ausführung und Nachbedingung im obigen Sinn zusammengehören (das heißt, wenn die Vorbedingung gilt und dann der Kasten durchlaufen wird, gilt anschließend die Nachbedingung).

Danach können wir mit diesen Bedingungen zeigen, dass der Ausgang immer erreicht wird und dass am Ausgang die gewünschte oder angepeilte Beziehung richtig ist.

Abbildung 5.5 zeigt das Flussdiagramm versehen mit solchen Markierungen:

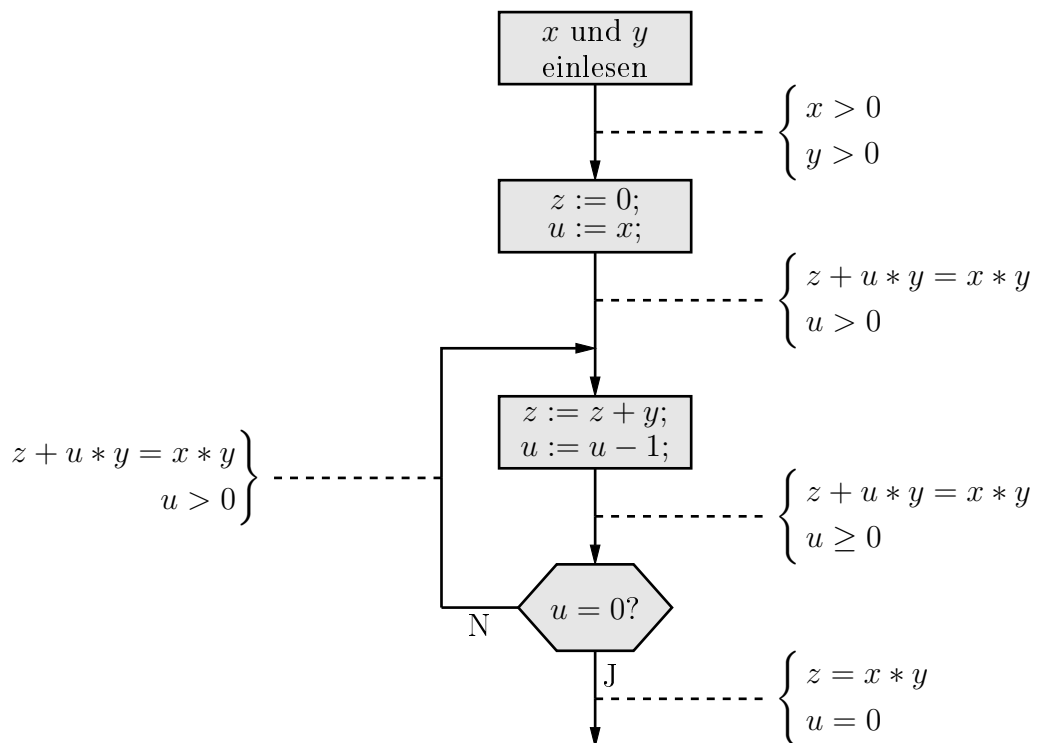


Abbildung 5.5: Das markierte Flussdiagramm des Multiplikationsprogramms

Wir prüfen kurz nach, dass alle Bedingungen zusammenpassen. Die Vor- und Nachbedingung des zweiten Kastens von oben passen zusammen, weil mit den Zuweisungen im Kasten gilt $z + u * y = 0 + x * y = x * y$ und $u > 0$ weil $x > 0$.

Wir haben oben schon gesehen, dass die Vor- und Nachbedingungen des Ausführungskastens in der Schleife zusammenpassen. Zwar gibt es hier *zwei* Vorbedingungen: eine markiert den von oben kommenden Pfeil, die andere den Pfeil, der die Rückkehr zum Anfang der Schleife darstellt. Aber die Bedingungen auf beiden Pfeilen sind gleich, so dass es kein Problem gibt.

Es wäre auch nicht schlimm, wenn die Eingangspfeile verschiedene Bedingungen trugen, aber man müsste dann beide getrennt auf Kompatibilität mit der Nachbedingung prüfen.

Die Abfrage unten lässt ihre Vorbedingung gültig, aber „und“et sie mit dem Ergebnis der Abfrage, so dass beim „N“-Ausgang aus $u \geq 0$ wieder $u > 0$ wird, und beim „J“-Ausgang $u = 0$ wird, womit aus $z + u * y = x * y$ dann $z = x * y$ folgt.

Wenn die Ausführung des Programms mit dem „J“-Ausgang der Abfrage terminiert, haben wir mit dieser Markierung den Nachweis erbracht, dass das Programm wirklich das Produkt von x und y berechnet. Ein Blick auf das Flussdiagramm zeigt, dass solange der „J“-Ausgang nicht genommen wird, immer wieder die Schleife durchlaufen wird, in der nur der letzte Ausführungskasten S ausgeführt wird. Und er vermindert u bei jedem Durchlauf, so dass nur endlich viele Durchläufe möglich sind, bevor im Abfragekasten $u = 0$ zutreffen wird.

Das ist der Nachweis, dass der „J“-Ausgang irgendwann genommen werden muss, und dass das Programm tatsächlich terminiert.

Obwohl dies nur ein Beispiel war, haben wir hier zwei einfache allgemeine Prinzipien angewendet, die auch in anderen Korrektheitsbeweisen benutzt werden können.

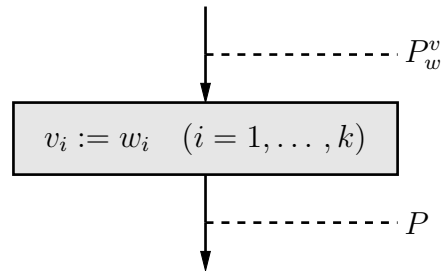
Der erste beschreibt, wie eine gültige Bedingung bei der Passage durch einen Ausführungskasten abzuwandeln ist, damit sie hinterher wieder gültig ist. Damit kann man erkennen, wann eine Vor- und eine Nachbedingung zu diesem Kasten zueinander passen.

In dem Kasten seien gewisse Wertzuweisungen

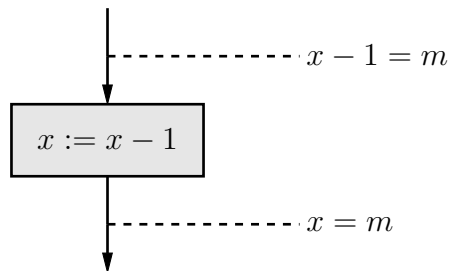
$$v_i := w_i \qquad (i = 1, \dots, k)$$

vorgenommen, wo die v_i Programmvariablen sind und die w_i gewisse Ausdrücke, deren Wert diesen Variablen zugewiesen wird. Sei P eine Bedingung, in der die v_i oder einige von ihnen vorkommen.

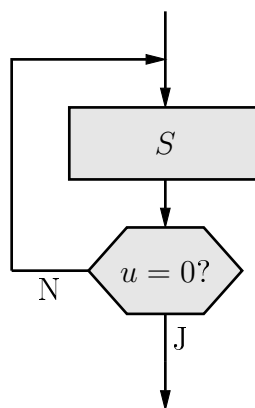
Wir bezeichnen mit P_w^v die Bedingung, die entsteht, wenn in P jedes Vorkommen eines v_i durch den entsprechenden Ausdruck w_i ersetzt wird. Dann gilt



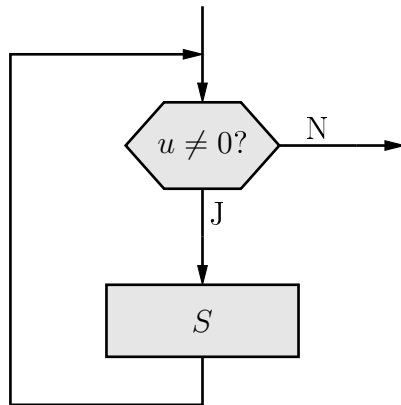
Zum Beispiel, wenn x ein Variablenname und wenn m eine bestimmte natürliche Zahl ist, dann gilt



Unsere Behandlung der Schleife in Beispiel 5.2 ist auch eine Instanz einer allgemeinen Regel. Die Schleife hatte folgendes Aussehen, wo S das Paar von Anweisungen $z := z + y$, $u := u - 1$ war:



Auf dem von oben kommenden Pfad galt automatisch die Bedingung $u \neq 0$, die auch für die Wiederholung der Schleife verantwortlich war, so dass es nichts ändert, wenn wir diesen Programmabschnitt wie folgt umgestalten:



Hier wird die Bedingung *vor* der Schleife abgefragt, und die Schleife wird *so lange* immer wieder ausgeführt, *wie die Bedingung in der Abfrage gilt*.

Schleifen dieser Art sind eine Standardstruktur der Programmierung und nennen sich **while**-Schleifen, weil sie ausgeführt werden, „while“=so lange wie eine steuernde Bedingung B gilt. Abbildung 5.6 zeigt die allgemeine Struktur einer **while**-Schleife.

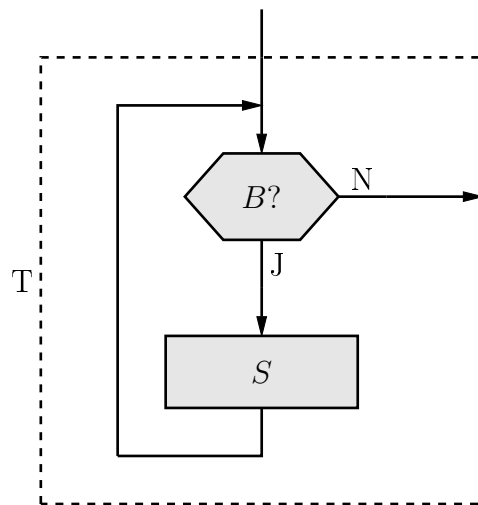


Abbildung 5.6: Flussdiagramm einer **while**-Schleife

Wie in der Abbildung durch den gestrichelten Kasten angedeutet, können wir die ganze **while**-Schleife zu einem neuen Ausführungskasten zusammenfassen.

Auf dem „J“-Zweig nach der Abfrage gilt automatisch Bedingung B , und auf dem „N“-Zweig gilt \overline{B} . Angenommen, wir haben eine Bedingung P , für die gilt

$$\{P \wedge B\}S\{P\}. \quad (5.1)$$

Da die Abfrage die Gültigkeit von P nicht zunichte machen kann, bedeutet (5.1), dass wenn P vor der Abfrage gilt, es in der Schleife gültig bleibt (weil B auf dem „J“-Zweig, also vor S auch gültig ist), und P ist dann auch beim Ausgang aus der Schleife immer noch gültig (natürlich auch, wenn die Schleife nie durchlaufen wird).

Also können wir aus (5.1) folgern, dass $\{P\}T\{P \wedge \overline{B}\}$ oder ausführlicher $\{P\}\mathbf{while}(B) S; \{P \wedge \overline{B}\}$.

In anderen Worten, wir haben eine allgemeine **Verifizierungsschlussregel für Schleifen**

$$\{P \wedge B\}S\{P\} \implies \{P\}\mathbf{while}(B) S; \{P \wedge \overline{B}\}. \quad (5.2)$$

Diese Regel haben wir in Beispiel 5.2 mit

$$P = \{z + u * y = x * y\} \quad \text{und} \quad B = \{u \neq 0\}$$

ausgenutzt, um beim Ausgang zu schließen: $(z + u * y = x * y) \wedge (u = 0)$, also $z = x * y$.

Beispiel 5.3 Als weiteres Anwendungsbeispiel für unsere Schlussfolgerungsregeln präsentieren wir ein Programm, dass den größten gemeinsamen Teiler von zwei positiven natürlichen Zahlen berechnet, und verifizieren dieses Programm.

Zur Erinnerung: Wenn d und n natürliche Zahlen sind, so heißt d ein **Teiler** von n (und wir notieren das mit $d \mid n$), wenn es eine natürliche Zahl k gibt mit

$$n = kd. \quad (5.3)$$

Eine Zahl d heißt ein **gemeinsamer Teiler** von zwei Zahlen a und b , wenn d ein Teiler von a und ein Teiler von b ist.

Jede Zahl hat 1 als einen Teiler, und deshalb ist 1 auch ein gemeinsamer Teiler von jedem Zahlenpaar. Eine positive Zahl n hat nie 0 als Teiler, so dass 1 immer der kleinste Teiler von n ist. Aus der Beziehung (5.3) ist dann klar, dass n keine Teiler größer als n haben kann (und es gilt immer $n \mid n$, so dass n tatsächlich der *größte* Teiler von sich ist).

Aus diesen einfachen Überlegungen folgt, dass auch ein Zahlenpaar nicht beliebig große *gemeinsame* Teiler haben kann. Da ein Paar von positiven natürlichen Zahlen aber immer mindestens einen gemeinsamen Teiler besitzt (nämlich 1), besitzt es auch einen eindeutig bestimmten *größten* gemeinsamen Teiler d . Wir kürzen die Bezeichnung „größter gemeinsamer Teiler“ ab mit „ggT“.

Folgendes Programm, geschrieben in der Programmiersprache **C**, berechnet den größten gemeinsamen Teiler $a := \text{ggT}(x, y)$ von zwei eingegebenen

positiven natürlichen Zahlen x und y und hinterlässt das Ergebnis in der Variablen a .

Wir erläutern ganz kurz einige Besonderheiten von **C**. Anweisungen in **C** werden mit einem Semikolon „;“ abgeschlossen und voneinander getrennt. Eine Wertzuweisung an eine Variable wird mit einem einfachen Gleichheitszeichen „=“ notiert (was rechts vom Gleichheitszeichen steht ist der neue Wert von der Variablen, die links vom Gleichheitszeichen steht). Zur Unterscheidung wird die Gleichheitsrelation mit einem doppelten Gleichheitszeichen „==“ geschrieben, und die im Programm erscheinende Relation „!=“ bedeutet *ungleich*.

Kommentare werden von den Zeichenfolgen „/*“ und „*/“ begrenzt. Eine Bedingung B kann abgefragt werden mit der Anweisung „if(B)“ und wenn die Bedingung gilt, wird die darauf folgende Anweisung ausgeführt. Wenn die Bedingung nicht gilt, wird die Anweisung, die nach einem folgenden Schlüsselwort **else** steht, ausgeführt (oder nichts wird ausgeführt, wenn es kein **else** gibt). Eine **while**-Schleife, die unter Kontrolle der Bedingung B ausgeführt werden soll, wird durch „while(B) Anweisung“ kodiert.

Wo immer eine einzelne Anweisung stehen kann (zum Beispiel nach einem **if** oder einem **while**), kann sie ersetzt werden durch einen **Block** von Anweisungen, eingeschlossen in geschweiften Klammern „{“ und „}“.

Diese kurzen syntaktischen Angaben reichen aus, um das folgende Programm verstehen zu können:

```
a = x; b = y;          /* x und y sind ganzzahlig, >0 */
while (a != b) {
    if (a > b)
        a = a - b;
    else b = b - a;
}                      /* hier enthält a den ggT(x,y) */
```

Wir erproben das Programm an einem Beispiel mit $x = 15$ und $y = 27$. Der größte gemeinsame Teiler ist 3, wie man schnell nachprüfen kann.

Wir führen in einer Tabelle auf, welche Programmzeilen in welcher Reihenfolge ausgeführt werden und welche Werte die Variablen und Bedingungen vor und nach der Zeile haben (die Bedingungen werden nur ausgewertet, wenn sie für die auszuführende Programmzeile relevant sind, und die „nachher“-Werte werden nur angegeben, wenn die Werte in der Zeile geän-

dert wurden):

Zeile	Vorher				Nachher	
	a	b	$a \neq b?$	$a > b?$	a	b
1					15	27
2	15	27	Ja			
3	15	27		N		
5	15	27			15	12
2	15	12	Ja			
3	15	12		Ja		
4	15	12			3	12
2	3	12	Ja			
3	3	12		Nein		
5	3	12			3	9
2	3	9	Ja			
3	3	9		Nein		
5	3	9			3	6
2	3	6	Ja			
3	3	6		Nein		
5	3	6			3	3
2	3	3	Nein			
6	3	3				

Am Ende der Bearbeitung steht tatsächlich der größte gemeinsame Teiler 3 in Variable a .

Wir wollen beweisen, dass das Program *immer* funktioniert.

Sei P die Bedingung

$$(\text{ggT}(a, b) = \text{ggT}(x, y)) \wedge (a > 0) \wedge (b > 0).$$

Wir zeigen zunächst, dass für je zwei Zahlen $a \neq b$ gilt

$$\text{ggT}(a, b) = \text{ggT}(a - b, b). \quad (5.4)$$

Sei s ein gemeinsamer Teiler von a und b . Dann gibt es Zahlen m und n mit

$$a = m \cdot s \quad \text{und} \quad b = n \cdot s,$$

woraus folgt

$$a - b = m \cdot s - n \cdot s = (m - n) \cdot s,$$

d.h., s ist auch ein Teiler von $a - b$ und somit ein *gemeinsamer* Teiler von $a - b$ und b .

Umgekehrt, sei t ein gemeinsamer Teiler von $a - b$ und b . Dann gibt es Zahlen p und q mit

$$a - b = p \cdot t \quad \text{und} \quad b = q \cdot t,$$

woraus folgt

$$a = (a - b) + b = p \cdot t + q \cdot t = (p + q) \cdot t,$$

d. h., t ist auch ein Teiler von a und somit ein *gemeinsamer* Teiler von a und b .

Damit ist gezeigt, dass das Paar (a, b) und das Paar $(a - b, b)$ die gleichen gemeinsamen Teiler insgesamt haben, und deshalb auch den gleichen *größten* gemeinsamen Teiler, wie in Gleichung (5.4) behauptet.

Weil offensichtlich die gemeinsamen Teiler eines Zahlenpaares nicht davon abhängen, in welcher Reihenfolge man die Zahlen des Paares hinschreibt, können wir daraus auch folgern, dass

$$\text{ggT}(a, b) = \text{ggT}(a, b - a). \quad (5.5)$$

Mit diesen Behauptungen ist auch klar, dass für die Ausführungszeilen für beide Zweige der **if**-Anweisung im Programm gilt

$$\{(P \wedge a \neq b) \wedge a > b\} \mathbf{a} = \mathbf{a} - \mathbf{b}; \{P\}$$

und

$$\{(P \wedge a \neq b) \wedge b > a\} \mathbf{b} = \mathbf{b} - \mathbf{a}; \{P\}.$$

Die in P enthaltenen Bedingungen $a > 0$ und $b > 0$ bleiben bei der ersten Anweisung erhalten wegen der dort eingefügten zusätzlichen Voraussetzung $a > b$ und weil b dort nicht verändert wird, und sie bleiben bei der zweiten Anweisung erhalten, weil a dort nicht verändert wird und wegen der dort eingefügten zusätzlichen Voraussetzung $b > a$.

Daraus folgt, dass für die ganze **if**-Anweisung gilt

$$\{P \wedge a \neq b\} \mathbf{if}(a > b) \mathbf{a} = \mathbf{a} - \mathbf{b}; \mathbf{else} \mathbf{b} = \mathbf{b} - \mathbf{a}; \{P\},$$

denn die **if**-Verzweigung sorgt dafür, dass die zusätzliche Voraussetzung $a > b$, von der diese Behauptung abhängt wenn die erste Anweisung im **if**-Block ausgeführt wird, auch tatsächlich erfüllt ist, falls die Verzweigung zur ersten Anweisung führt, und sorgt in Kombination mit der Bedingung $a \neq b$ dafür, dass die zusätzliche Voraussetzung $b > a$, von der diese Behauptung

abhängt wenn die zweite Anweisung im **if**-Block ausgeführt wird, auch tatsächlich erfüllt ist, falls die Verzweigung *nicht* zur ersten Anweisung und deshalb zur Ausführung der zweiten Anweisung führt (in diesem Fall ist $a \not> b$ und $a \neq b$, also tatsächlich $b > a$).

Mit der Schlussregel (5.2) für **while**-Schleifen folgt dann für die gesamte **while**-Anweisung W in unserem **C**-Programm, dass

$$\{P\}W\{P \wedge a = b\} \quad (5.6)$$

Bedingung P besagt $\text{ggT}(a, b) = \text{ggT}(x, y) \wedge a > 0 \wedge b > 0$ und gilt offensichtlich nach der ersten Programmzeile *immer* (weil $x > 0$ und $y > 0$):

$$\{\} \mathbf{a} = \mathbf{x}; \mathbf{b} = \mathbf{y}; \{P\}.$$

Damit folgt aus Schluss (5.6), *wenn* die **while**-Schleife und damit das Programm terminiert, dass dann gilt

$$\text{ggT}(a, b) = \text{ggT}(x, y) \quad \text{und} \quad a = b > 0.$$

Aber ein Paar (a, a) , in der beide Zahlen gleich und positiv sind, hat alle Teiler dieser einzigen Zahl a als gemeinsame Teiler, und wir haben gesehen, dass der größte Teiler einer einzigen positiven Zahl a diese Zahl selber ist. Das heißt, $\text{ggT}(a, a) = a$ und somit gilt wie behauptet

$$a = \text{ggT}(x, y),$$

falls das Programm terminiert!

Es bleibt nur noch zu zeigen, *dass* das Programm immer terminiert. Dazu suchen wir wieder eine positive „Leitgröße“, die in jedem Durchgang der Schleife reduziert wird; das kann dann nur endlich oft passieren, so dass die Schleife irgendwann verlassen werden muss.

Die Größen a und b taugen nicht als Leitgrößen, weil jede einzelne von ihnen nur manchmal, aber nicht jedes Mal, in der Schleife vermindert wird. Allerdings, wenn a nicht kleiner wird, dann wird b kleiner, und deshalb kann man die *Summe* $s := a + b$ als Leitgröße verwenden.

Wir erläutern das ganz genau.

Nach der ersten Programmzeile ist $a + b > 0$, weil P gilt und deshalb a und $b > 0$ sind.

Wenn die Schleife durchlaufen wird, ist $a \neq b$. Wenn $a > b$, dann wird die erste Anweisung im **if**-Block ausgeführt und weil $b > 0$ wird a kleiner, somit auch $a + b$ kleiner (da b nicht verändert wurde). Wenn $a \not> b$, dann wird die zweite Anweisung im **if**-Block ausgeführt und weil $a > 0$ wird b kleiner, somit auch $a + b$ kleiner (da a nicht verändert wurde).

Das heißt, bei jeder Ausführung des **if**-Blocks und somit bei jedem Durchgang durch die **while**-Schleife wird $s := a + b$ auf jeden Fall kleiner, bleibt aber positiv, da P gültig bleibt.

Eine positive Größe kann nicht beliebig oft verkleinert werden und immer noch positiv bleiben. Aus diesem Grund muss die Voraussetzung für die **while**-Schleife irgendwann ungültig werden und die Schleife und mit ihr das Programm terminieren.

Damit sind wir mit unserem Korrektheitsbeweis fertig.

Eine wichtige Frage beim Einsatz von automatischen elektronisch gesteuerten Geräten besteht in der Suche nach Wegen, ihre Anwendung bequemer, effizienter und schneller zu machen. Dabei kann es sowohl darum gehen, einzelne Aufgaben zügig und effektiv zu erledigen, wie auch um die effiziente Abwicklung von mehreren Aufgaben und ihre Aufteilung unter mehreren ausführenden Geräten.

Eine wichtige Idee, die sowohl zur Beschleunigung der Bearbeitung wie auch zum sparsamen Einsatz von Ressourcen angewendet werden kann, ist die **Parallelität**, d. h., die „gleichzeitige“ Ausführung von Teilaufgaben durch mehrere Ausführende, um ihre „Arbeitskraft“ möglichst effizient einzusetzen. Dieser Weg ist oft einfacher und Kosten sparer als der Versuch, die Leistung eines einzelnen ausführenden Gerätes mit Anstrengung zu steigern.

Der Gedanke beginnt schon bei einfachen Schulmathematikaufgaben wie „Wenn ein Bauarbeiter 10 m Mauer an einem Tag bauen kann, wieviele Meter Mauer können drei Bauarbeiter in einem halben Tag bauen?“

Bei der Mauer ist klar, dass die drei Bauarbeiter wohl gleichzeitig an Mauerabschnitten arbeiten können, ohne sich gegenseitig zu stören, aber wenn es darum geht, einen Graben in der Straße auszuheben, dann spielt es vielleicht eine Rolle, dass ein Arbeiter für das Aufreißen der Teerdecke zuständig ist, dass der zweite mit einem Bagger die Erde aushebt und der dritte für die Abstützung des entstehenden Grabens zuständig ist. Hier gibt es Nebenbedingungen, die die Kooperation einschränken, denn die Erde kann in einem Abschnitt erst ausgehoben werden, wenn der erste Arbeiter mit der Teerdecke schon fertig ist, und der dritte Arbeiter kann erst ansetzen, wenn der Graben schon ausgehoben ist.

Parallele Verarbeitung spielt seit den Anfängen im Computerbau und in Computeranwendungen eine sehr wichtige Rolle. Hier einige Beispiele:

- Schon die IBM 7094 und andere Rechner dieser Zeit hatten getrennte Ein-Ausgabeeinheiten, die sich unabhängig von der CPU um die Kommunikation mit angeschlossenen Bandlaufwerken, Kartenlesern, Druckern und Festplatten kümmerten, und die mitteilen konnten, ob sie frei oder mit einem anderen Auftrag beschäftigt waren.

- Ein Wallace Baum für die Multiplikation besteht aus mehreren in Ebenen organisierten Full Adder, und die Full Adder in jeder Ebene können unabhängig voneinander und gleichzeitig operieren.
- Ein Nadelöhr bei der Ausführung von Programmen ist die Tatsache, dass CPUs sehr viel schneller rechnen können, als die Programmbefehle und Daten über relativ langsame Datenleitungen und Speicherzugriffen aus dem Speicher geholt werden können. Das Problem kann gelindert werden, indem man Datenbereiche aus dem Hauptspeicher im Voraus in einen CPU-nahen schnelleren Speicher (*Cache*) holt, aber eine weitere Technik, um die Bearbeitung wesentlich zu beschleunigen, ist das *Pipelining*.

Beim Pipelining werden mehrere hintereinander auszuführende Programmbefehle auf einmal aus dem Speicher geholt und in eine *Pipeline* gelegt, eine Art Warteschlange, die so funktioniert wie ein Förderband. Wir haben in Kapitel 4 gesehen, dass die Ausführung eines Befehls mehrere Schritte erfordert und mehrere Stadien durchläuft. In einer Pipeline werden diese Schritte von unabhängigen Recheneinheiten ausgeführt, die nur ihren Teil der Dekodierung und Befehlsverarbeitung für die Befehle in Reihenfolge ausführen und den Befehl dann „weiterreichen“ an die Einheit für den nächsten Bearbeitungsschritt. So überlappt sich die Ausführung der Befehle in der Pipeline; während ein Befehl auf seine Daten operiert, können die Daten für den nächsten Befehl schon aus dem Speicher geholt werden, während der folgende Befehl entziffert wird und der darauf folgende aus dem Speicher geholt wird.

Obwohl jeder einzelne Befehl eine längere Zeitspanne zur Ausführung benötigt, ist der Befehlsdurchsatz so, als würde jeder Befehl nur so viel Zeit beanspruchen, wie die Ausführung eines einzelnen Befehlsausführungsteilschrittes verlangt.

Hier kann es auch Konflikte geben. Befehle, deren effektive Datenadresse vom Ergebnis des vorherigen Befehls abhängt, müssen eventuell auf die vollständige Ausführung des vorhergehenden Befehles warten; Befehle, die in der Pipeline hinter einem bedingten Sprung vorkommen, müssen eventuell gelöscht werden und durch neue Befehle ersetzt werden, wenn der bedingte Sprung einen anderen Zweig anspringt.

Deshalb sind auch hier Strategien erforderlich, um die parallele Bearbeitung zu erleichtern und zu ermöglichen (zum Beispiel werden manchmal vorsorglich die Befehle aus *allen* möglichen Sprungzweigen in verschiedene Pipelines gepackt, damit wenigstens eine Pipeline weiterrechnen kann, während die andere neu beladen wird).

- Externe Geräte wie Drucker oder Datenleitungen (Modems) können von mehreren Computern oder mehreren Programmen gleichzeitig benutzt werden. Deshalb werden Druckaufträge in der Regel nicht direkt an einen Drucker weitergeleitet, sondern nur an das Betriebssystem gemeldet, das sie in eine Druckerschlange (*Spooler*) lädt, wo ein dediziertes Programm sie einen nach dem anderen an den Drucker schickt, sobald er frei ist. Das Hauptprogramm kann in der Zwischenzeit weiterrechnen und muss nicht auf den Drucker oder die Datenleitung warten.
- Eine Standardmethode, um Prozessoren zu beschleunigen, besteht darin, die Schaltungen immer kleiner zu machen und immer dichter zu packen, so dass sie auch schneller schalten und schneller miteinander kommunizieren können.

Leider sind dieser Entwicklungsrichtung aber Grenzen gesetzt, an die die Technologie inzwischen gestoßen ist. Desto schneller eine Schaltung schaltet, desto mehr Energie verbraucht sie und desto heißer wird sie, so dass aufwendige Kühlsysteme notwendig sind, um schnelle CPUs vor der Selbstzerstörung zu bewahren.

Ein Ausweg aus diesem Dilemma mit wenigen Einschränkungen und vielen Vorteilen besteht darin, *mehrere* langsamere und sparsamere CPUs fest oder lose zusammenzukoppeln und die anliegenden Rechenaufgaben unter ihnen aufzuteilen. Das gleiche kann man auch mit wesentlichen Teilen von CPUs (wie Addierern oder ganzen ALUs) machen, oder in der anderen Entwicklungsrichtung mit ganzen Rechnern, die über Datennetze sich koordinieren.

Beispiele für solche Lösungen sind: Vektorrechner, die mit einem Befehl zum Beispiel mehrere Datenpaare addieren können und als Summe einen aus mehreren Zahlen bestehenden Datenvektor errechnen, Multiprozessorrechner (wie der HP 32000 an der Ruhr-Universität, in dem 28 Prozessoren sich die Arbeit aufteilen und parallel laufen), oder Rechnernetze, die über lokale Datenleitungen oder sogar über das Internet koordiniert sein können.

Es gibt zum Beispiel ein Projekt GIMPS („Great Internet Mersenne Prime Search“), an dem jeder Computerbenutzer sich beteiligen kann, um neue große Primzahlen zu entdecken (der letzte Erfolg datiert vom 4. September 2006 mit der Entdeckung einer Primzahl mit 9.808.358 Dezimalziffern). Zur Teilnahme am Projekt muss man von www.mersenne.org ein kleines Computerprogramm herunterladen und starten. Das Programm läuft nur, wenn der eigene Rechner sonst keine Aufgaben ausführt oder freie CPU-Zeit hat, führt Teilberechnungen

der Primzahlprüfung aus und meldet die Ergebnisse an einen zentralen Server, der das ganze Projekt koordiniert. Diese internationale Internet-koordinierte parallele Anwendung von Tausenden von Computern hat bisher 10 Mersenne Primzahlen entdeckt.

Wer vor zwei Jahren einen neuen Rechner kaufte, konnte ihn vielleicht mit einem Intel Pentium 4 mit einer Taktfrequenz von bis zu 3,8 GHz bestücken, der intern mehrere Prozesspfade parallel ausführen konnte, um seinen Durchsatz zu steigern. Dieser Prozessor musste beim Betrieb mit einem riesigen Kühlkörper abgekühlt werden. Über diese Taktfrequenz ist Intel nie hinausgegangen, weil die physikalischen Probleme zu groß geworden wären.

Zur Leistungssteigerung wurde eine andere Lösung eingesetzt, die auch bei größeren Rechnern schon länger im Gebrauch ist: mehrere CPUs oder CPU-Kerne, die parallel arbeiten.

Wer heute einen neuen Rechner kauft und ihn mit einer Intel CPU ausstatten will, wird zu einem langsameren Prozessor als den älteren Pentium 4 HT greifen (bei vergleichbarem Preis vielleicht zu einem mit einer Taktfrequenz von 2,66 GHz), der aber zwei oder sogar vier Rechenkerne hat, die gleichzeitig rechnen können. *Bei geeigneter Software* sind diese neuen CPUs trotz der niedrigeren Taktfrequenz schneller als die alten Rechner und sind sparsamer und leichter zu kühlen.

Die Verbesserungen, die Parallelität wie oben erläutert mit sich bringt, sind aber nicht ganz kostenlos (wie die italische Schrift im letzten Satz schon andeutet).

Parallele Verarbeitung erfordert viel Planung und viel Organisation, um sicher zu gehen, dass die parallel verfügbare Rechenkraft wirklich ausgenutzt wird und nicht brach liegt, und noch wichtiger, um sicher zu gehen, dass die parallel arbeitenden Recheneinheiten sich nicht gegenseitig in die Quere kommen oder sich blockieren.

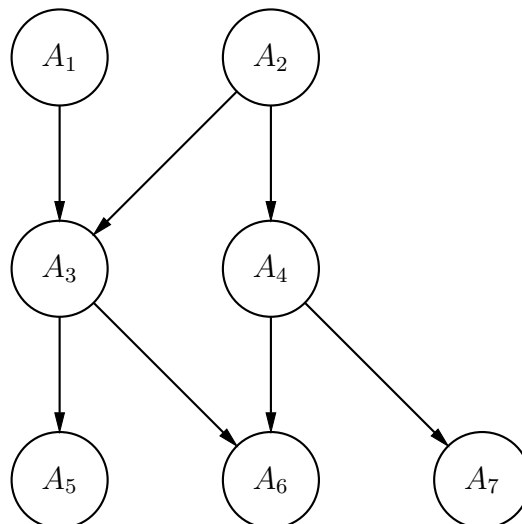
Rechenschritte können nur dann parallel ausgeführt werden, wenn sie wirklich unabhängig sind, wenn sie keine Ressourcen, die nur einmal vorhanden sind, gleichzeitig beanspruchen, und wenn ein Prozess nicht die Ergebnisse eines anderen benötigt und darauf warten muss. Noch schlimmer ist es, wenn ein Prozess Registerinhalte oder andere Daten zerstört, die ein anderer Prozess noch braucht, denn wenn das unbemerkt passiert, kommen trotzdem Rechenergebnisse heraus, aber eben falsche.

Wir wollen jetzt untersuchen, wie man die Parallelität theoretisch gut beschreiben und handhaben kann, um Probleme, wie sie im letzten Absatz genannt wurden, abzufangen und zu vermeiden.

Einige Ideen zur Beherrschung der Parallelität kommen aus der Biologie und der Untersuchung des Aufbaus und der Funktionsweise von Nervensystemen in Lebewesen. Es handelt sich hier um die Theorie der **neuronalen Netze** und um verschiedene theoretische Modelle, die die Funktion echter neuronalen Netze beschreiben oder nachahmen sollen.

Eine ganz einfache Art von Parallelität besteht in der Zerlegung einer Gesamtaufgabe in einmalig auszuführende Teilaufgaben, von denen manche unabhängig voneinander erledigt werden können aber andere in einer bestimmten zeitlichen Reihenfolge bearbeitet werden müssen (weil das Ergebnis von einer Teilaufgabe vielleicht von einer anderen weiterverarbeitet werden muss oder die Ausführung einer anderen Teilaufgabe auf andere Weise bestimmt).

Solche zeitlichen Abhängigkeiten können in einem einfachen **azyklischen Graphen** dargestellt werden.



Auf jeder Höhe in diesem Bild können die Aufgaben A_i unabhängig voneinander und in beliebiger Reihenfolge oder gleichzeitig ausgeführt werden, aber jede Aufgabe kann erst beginnen, wenn die Aufgaben in der Ebene über ihr, mit denen sie durch einen Pfeil verbunden ist, erledigt sind.

Die Teilaufgaben müssen also nicht in einer bestimmten Abfolge bearbeitet werden, aber die Reihenfolge ist durch die Pfeile eingeschränkt.

Natürlich darf der Graph der Abhängigkeiten keine **Zyklen** enthalten, also keine geschlossenen Pfade den Pfeilen folgend, die wieder zu ihrem Ausgangspunkt zurückkehren. Sonst kann die zeitliche Reihenfolge nicht eingehalten werden.

Eine wichtige Verfeinerung dieser einfachen Darstellung von parallelen Prozessen und ein wichtiges Werkzeug überhaupt für die Untersuchung des Zusammenspiels von Prozessen, die teils parallel ablaufen können aber auch

verschiedenartige Abhängigkeiten untereinander haben können, bilden die **Petri-Netze** (erfunden von Carl Adam Petri in seiner Dissertation 1962). Petri-Netze sind nicht nur in der Informatik geläufig, sondern werden für die Beschreibung von Prozessabläufen in verschiedenartigsten Anwendungsgebieten benutzt, auch in den Ingenieurwissenschaften.

Wir werden hier nur eine relativ einfache Art von Petri-Netzen besprechen. Das wesentliche Merkmal aller Petri-Netze ist die Beschreibung der Einzelschritte von Prozessabläufen als „Übergänge“ oder **Transitionen**, die von Vorbedingungen oder lokalen Zuständen explizit ausgelöst oder „gefeuert“ werden müssen. Die Verfügbarkeit eines lokalen Zustandes für das Auslösen einer Transition wird durch eine Markierung gekennzeichnet, die der Zustand unter den richtigen Bedingungen erwirbt, und die ihn befähigt, an der Auslösung der angeschlossenen Transitionen mitzuwirken.

Das Auslösen einer Transition *kann* sie in Gang setzen, aber wenn mehrere Transitionen gleichzeitig ausgelöst werden, kann nur eine von ihnen tatsächlich aktiv werden; man sagt dann, dass sie **schaltet**.

Eine Transition, die schaltet, verändert damit einige der lokalen Zustände, und ermächtigt sie durch Verleihung der oben genannten Markierung, weitere Transitionen auszulösen.

Diese Idee, die wir gleich genauer präzisieren und illustrieren werden, ermöglicht eine feine und detaillierte Beschreibung von Abhängigkeiten und Wechselwirkungen zwischen Prozessschritten, und liefert ein Modell für parallele Abläufe, mit dem weitgehende theoretische Untersuchungen möglich sind.

Unsere Version von Petri-Netzen ist aber nicht die einzige und nicht die leistungsfähigste Variante, und es sind viele Zusätze und Verallgemeinerungen erfunden worden, die eine noch genauere und noch feiner abgestimmte Beschreibung von speziellen Abhängigkeiten und Situationen ermöglichen.

Bei uns sind die Markierungen ganze Zahlen, damit die Fähigkeit, einen Übergang auszulösen, eine bestimmte Anzahl von Malen vorhanden sein kann. In der graphischen Darstellung, die wir gleich erläutern werden, werden diese „Auslösefähigkeiten“ durch kleine Punkte dargestellt, die **Tokens** genannt werden. Das Auslösen einer Transition kostet 1 Token, und alle Tokens haben die gleiche Fähigkeit und sind nicht unterscheidbar.

Es gibt Varianten von Petri-Netzen, in denen Tokens verschiedenartig sein können (dargestellt durch Farben) und verschiedene Wirkungen haben; es gibt Varianten, in denen die Bereitstellung von Vorbedingungen für Transitionen mit „Kosten“ gewichtet werden kann und in denen die Markierungen reelle Zahlen sind, die beim Auslösen einer Transition diese Kosten ausgleichen müssen; es gibt Petri-Netz-Modelle, in denen Übergänge nicht nur gefeuert sondern auch blockiert oder erschwert werden können; es gibt Modelle,

in denen Markierungen durch boolesche Werte (**wahr** oder **falsch**) dargestellt werden und somit nur einmal einen Übergang auslösen können, auch wenn sie mehrmals gesetzt wurden; und es gibt Varianten von Petri-Netzen, in denen die lokalen Zustände nur begrenzt viele Tokens aufnehmen können (und wenn sie voll sind, deshalb Transitionen blockieren, die beim Schalten eine neue Markierung setzen würden). Alle diese Varianten werden wir hier ignorieren und nur eine Standardvariante weiter behandeln.

Die genaue Definition eines Petri-Netzes in unserem Sinne wollen wir kombinieren mit einer graphischen Beschreibung, die sie ein bisschen verständlicher macht.

Petri-Netze sind mathematische Objekte, die eine formale mathematische Definition haben, anhand derer man mathematische Sätze über ihre Struktur und ihre Funktionsweise beweisen kann. Nur auf diese Weise sind sichere theoretische Aussagen über sie möglich.

Aber eine abstrakte formale Definition ist eher hinderlich beim Versuch, sich vorzustellen, was ein gegebenes Petri-Netz wirklich macht oder wie es reale Prozesse aus Anwendungen modelliert. Für das Verstehen der Wirkung eines Petri-Netzes gibt es eine *bildliche* Darstellung, die die Struktur mit den Augen erfassbar macht und so das Erfassen der Wirkung mit dem Verstand erleichtert.

Wir werden *beide* Beschreibungen „parallel“ erläutern, damit man anhand der Bilder die Intention der abstrakten eigentlichen Definition besser begreifen kann.

Definition 5.4 (Petri-Netz, zeichnerisch) Zur zeichnerischen Darstellung eines Petri-Netzes gehören:

- Eine Anzahl von Kreisen, die wir **Stellen** oder **Plätze** nennen, und die lokale Zustände darstellen sollen;
- Eine Anzahl von Balken oder kleinen Rechtecken, die wir **Transitionen** nennen, und die für Prozessschritte stehen (*durch* die der Gesamtprozess fortschreitet — deshalb der Name „Transitionen“);
- Pfeile, die von (manchen) Stellen zu (manchen) Transitionen gehen, und die die Abhängigkeiten von Transitionen von Vorbedingungen darstellen;
- Pfeile, die von (manchen) Transitionen zu (manchen) Stellen gehen, und die kennzeichnen, welche lokalen Zustände von der Ausführung einer Transition verändert werden.

Die Pfeile (beider Gattungen) werden auch (gerichtete) **Kanten** genannt.

Es gibt keine Pfeile, die direkt von Stellen zu Stellen oder direkt von Transitionen zu Transitionen führen.

Von einer bestimmten Stelle zu einer bestimmten Transition kann es *höchstens* einen Pfeil geben, und von einer bestimmten Transition zu einer bestimmten Stelle kann es auch höchstens einen Pfeil geben, aber es darf zwischen einer Stelle und einer Transition *in jeder Richtung* einen Pfeil geben.

- Eine Anzahl von Punkten innerhalb mancher Stellenkreise; die Punkte werden **Tokens** oder **Marker** genannt und kennzeichnen die Erfüllung der von einer Stelle dargestellten Vorbedingung und die damit verbundene Fähigkeit der Stelle, zur Aktivierung der von dieser Bedingung abhängigen Transitionen beizutragen.

Die Gesamtzuteilung von Tokens zu den Stellen nennt sich eine **Markierung** des Petri-Netzes und kennzeichnet seinen **momentanen Zustand**.

Petri-Netze sind nicht statische Gebilde, sondern dynamische Modelle für einen Prozessablauf, der schrittweise fortschreitet. Wie Petri-Netze sich verändern, wird in Definition 5.7 auf Seite 223 erläutert.

Abbildung 5.7 auf der nächsten Seite zeigt ein Beispiel eines Petri-Netzes.

Definition 5.4 ist nicht die *richtige* Definition eines Petri-Netzes, sondern nur die Beschreibung, wie man sie zeichnet. Die eigentliche Definition geben wir jetzt:

Definition 5.5 Ein **Petri-Netz** ist ein Quintupel

$$PN = (S, T, SNT, TNS, M), \quad \text{wobei}$$

- S eine nichtleere endliche Menge ist, deren Elemente **Stellen** oder **Plätze** heißen (sie stellen die lokalen Zustände dar);
- T eine nichtleere endliche und von S disjunkte Menge ist, deren Elemente **Transitionen** heißen (sie stellen die auszulösenden Prozessschritte dar);
- $SNT \subseteq S \times T$ (diese Menge von Stellen-Transitionen-Paaren beschreibt im Modell, welche lokalen Zustände Vorbedingung für das Auslösen welcher Prozessschritte sein sollen);

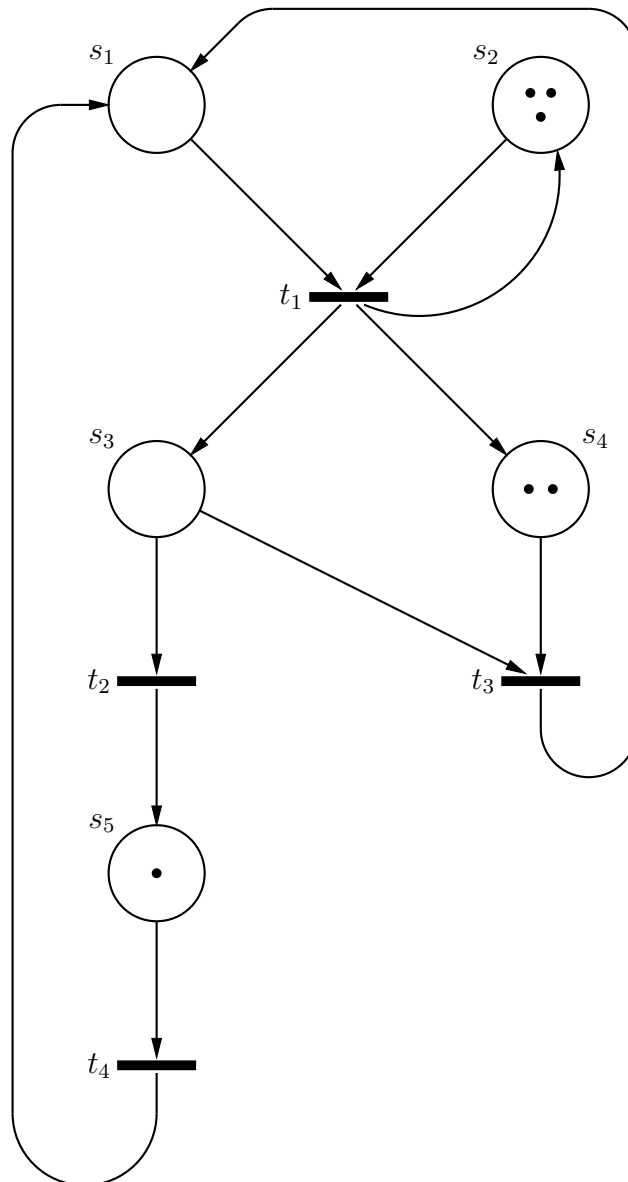


Abbildung 5.7: Ein Petri-Netz (zeichnerisch)

- $TNS \subseteq T \times S$ (diese Menge von Transitionen-Stellen-Paaren beschreibt im Modell, welche Transitionen oder Prozessschritte, wenn sie schalten oder ausgeführt werden, welche lokalen Zustände „setzen“ und ermöglichen, weitere Prozessschritte auszulösen);
- M ist eine Funktion $S \rightarrow \mathbf{N}$, genannt die **Markierung**. Sie kennzeichnet, welche lokalen Zustände wie oft die Vorbedingung für das Auslösen der von ihnen abhängenden Prozessschritte erfüllen.

Auch diese Definition ist noch nicht vollständig, weil wir noch nicht bestimmt haben, wie das Petri-Netz sich dynamisch verhält, also wie es sich fortentwickelt. Das wird in Definition 5.8 auf Seite 225 nachgeholt.

Der Zusammenhang zwischen der formalen Definition und der zeichnerischen Definition eines Petri-Netzes ist aber klar. S ist die Menge der Stellen und wir stellen die Elemente von S in der Zeichnung durch Kreise dar. T ist die Menge der Transitionen, die wir in der Zeichnung als Balken darstellen.

Dass S und T disjunkt sein sollen (also keine gemeinsamen Elemente haben dürfen) bedeutet nur, dass Stellen und Transitionen verschiedenartig sind und dass kein Ding gleichzeitig eine Stelle und eine Transition sein kann. Dass wir das so haben wollen, ist offensichtlich, aber wenn wir Stellen und Transitionen nur als Elemente irgendwelcher Mengen definieren, müssen wir den Fall, dass etwas zu beiden Mengen gehören könnte, *explizit* ausschließen.

Die Mengen SNT und TNS sind die formale Definition der Pfeile in der Zeichnung. Wichtig ist, dass es bei einem Pfeil nur darauf ankommt, wo er beginnt und wo er endet; wie er in der Zeichnung aussieht und welchen Verlauf er dort nimmt gehören *nicht* zu seinen identifizierenden Eigenschaften.

Deshalb reicht es in der formalen Definition, die Pfeile als Elementpaare (s, t) oder (t, s) zu definieren, denn nur die Endpunkte bestimmen den Pfeil. Da ein Element eines Pfeiles immer eine Stelle und das andere immer eine Transition sein muss, und da Stellen nie Transitionen sein können und umgekehrt, ist immer klar, ob ein Pfeil von einer Stelle zu einer Transition führt oder umgekehrt, und es ist deshalb auch immer eindeutig bestimmt, aus welcher der Mengen $S \times T$ oder $T \times S$ er kommt.

Da in der formalen Definition Pfeile nur durch das geordnete Paar ihrer Endpunkte dargestellt werden, ist es auch klar, dass es in einem Petri-Netz nie mehr als einen Pfeil in jeder Richtung zwischen einer Stelle und einer Transition geben kann. Zwei Pfeile in der gleichen Richtung, auch wenn man sie unterschiedlich zeichnen würde, hätten den gleichen Anfang und den gleichen Schluss und wären deshalb *der gleiche Pfeil*.

Die Markierungsfunktion ordnet jeder Stelle s eine natürliche Zahl $n = M(s)$ zu (die auch 0 sein kann). In der Zeichnung werden die Werte von M

durch die Anzahl der Tokens in den Stellenkreisen dargestellt. Wenn s eine Stelle ist und $M(s) = k$, dann enthält der Kreis für s genau k Tokens.

Beispiel 5.6 Für das Petri-Netz in Zeichnung 5.7 haben wir folgende formalen Daten:

$$\begin{aligned} S &= \{ s_1, s_2, s_3, s_4, s_5 \} \\ T &= \{ t_1, t_2, t_3, t_4 \} \\ S \times T &= \{ (s_1, t_1), (s_2, t_1), (s_3, t_2), (s_3, t_3), (s_4, t_3), (s_5, t_4) \} \\ T \times S &= \{ (t_1, s_2), (t_1, s_3), (t_1, s_4), (t_2, s_5), (t_3, s_1), (t_4, s_1) \} \\ M(2) &= 3, \quad M(4) = 2, \quad M(5) = 1, \quad M(s) = 0 \text{ sonst.} \end{aligned}$$

Wir haben gesagt, dass Petri-Netze dynamische Modelle sind, d. h., sie verändern mit der Zeit ihren globalen Zustand. Diese Veränderung geht schrittweise vor.

Die Stellen und Transitionen eines Petri-Netzes sind permanente Einrichtungen; sie verändern sich *nie*. Was sich von Schritt zu Schritt ändert und die „Prozessbeschreibung“ eines Petri-Netzes ausmacht, ist der **globale Zustand** des Petri-Netzes, der durch die Verteilung der Tokens oder äquivalent durch die Markierung bestimmt wird. Bei jedem Schritt werden also nur die Tokens anders verteilt, und im formalen Modell ändert sich nur die Markierung.

Wir definieren jetzt für beide „Sichtweisen“ auf Petri-Netze, wie diese Veränderung vor sich geht.

Definition 5.7 Sei ein (gezeichnetes) Petri-Netz gegeben.

Wir führen zunächst ein paar Hilfsbegriffe ein. Wir sagen, dass eine Stelle **vor** einer Transition liegt, wenn es einen direkten Pfeil von dieser Stelle zu dieser Transition gibt. Wir sagen, dass eine Stelle **hinter** einer Transition liegt, wenn es einen direkten Pfeil von dieser Transition zu dieser Stelle gibt.

Die Stellen **vor** einer Transition sind also die, die auf die Transition zeigen, und die Stellen **hinter** der Transition sind die, auf die die Transition zeigt.

Eine Transition heißt **aktiviert** oder **schaltbereit**, wenn jede Stelle, die vor ihr liegt, mindestens ein Token besitzt.

Aktiviert Transitionen *können* „**schalten**“ (dass bedeutet, dass der von der Transition beschriebene Prozessschritt zur Ausführung kommt), aber es ist nicht immer der Fall, dass eine aktivierte Transition tatsächlich schaltet. Es kann nämlich vorkommen, dass mehrere Transitionen gleichzeitig aktiviert sind; dann darf nur *eine* von ihnen schalten, und es ist nicht determiniert, welche schalten wird (das heißt, es gibt keine festen Kriterien dafür; manchmal wird der Zufall darüber entscheiden, manchmal vielleicht eine externe Instanz).

Wenn mehrere Transitionen gleichzeitig schaltbereit sind, sagt man deshalb, sie seien *in Konflikt*.

Bei jedem Schritt wird aber eine der aktivierten Transitionen tatsächlich schalten (wenn es überhaupt aktivierte Transitionen gibt). Dabei werden die Tokens auf folgende Weise umverteilt:

- jede Stelle vor der schaltenden Transition verliert ein Token;
- jede Stelle hinter der schaltenden Transition gewinnt ein Token hinzu;
- es ist möglich, dass eine Stelle sowohl vor wie auch hinter einer Transition liegt (zum Beispiel liegt die Stelle s_2 in Abbildung 5.7 sowohl vor wie auch hinter der Transition t_1); in diesem Fall verliert sie ein Token, aber gewinnt es gleich zurück, so dass die Anzahl ihrer Tokens sich nicht verändert;
- die Anzahl der Tokens verändert sich auch nicht bei Stellen, die weder vor noch hinter der schaltenden Transition liegen.

Ein paar Anmerkungen dazu: beim Schalten kommt es nicht darauf an, wieviele Tokens insgesamt im Spiel sind, sondern die Tokens kennzeichnen nur die Erfüllung einer Bedingung in einem lokalen Zustand, und diese Bedingungen sind die Voraussetzung dafür, dass Transitionen schalten dürfen, sowie ihr Schalten die Voraussetzung dafür ist, dass die relevanten Bedingungen in anderen lokalen Zuständen in Erfüllung gehen.

Insbesondere soll man nicht die Vorstellung haben, dass Tokens „durch“ die Transitionen fließen oder dass dieselben Tokens, die manchen Stellen weggenommen werden, dann auf andere Stellen „umverteilt“ werden (etwa so, dass vier Stellen je ein Token weggenommen wird und diese vier Tokens dann auf zwei andere Stellen verteilt werden, von denen jede dann zwei Tokens bekommt).

Die richtige Vorstellung ist die von zwei getrennten Akten. Die Stellen vor der schaltenden Transition verlieren je ein Token, weil die Vorbedingung, für die das Token wie ein Zertifikat bürgt, durch das Schalten „verbraucht“ ist. Die Stellen hinter der schaltenden Transition erhalten je ein Token als „Nachweis“ oder „Bescheinigung“, dass das Schalten bei diesen Zuständen etwas verändert hat und sie jetzt befähigt hat, weitere Aktionen auszulösen.

Das Petri-Netz wird den gerade beschriebene Schritt, bei dem jedesmal eine der aktivierten Transitionen schaltet und dadurch die Tokens verändert werden, immer wieder ausführen.

Jedesmal schaltet natürlich eine andere Transition, aktiviert durch die sich immer verändernde Verteilung der Tokens. Was für den Gesamt Ablauf

des beschriebenen Prozesses wichtig ist, ist welche Prozessschritte wann ausgeführt werden, und das wird im Petri-Netz durch die Folge der schaltenden Transitionen modelliert.

Diese Folgen

$$t_1 t_2 \dots t_k$$

von nacheinander schaltenden Transitionen heißen **Schaltfolgen** des Petri-Netzes und sind untersuchungswürdig.

Genau so wie Zustandsgraphen die Syntax einer Sprache beschreiben können, kann die Menge aller Schaltfolgen eines Petri-Netzes als eine Sprache aufgefasst werden, und Petri-Netze können verwendet werden, um Sprachen im allgemeinsten Sinn zu modellieren, sowie auch die Sprachtheorie Petri-Netze beschreiben kann.

Wir bemerken zum Schluss noch, dass es in einem Petri-Netz auch vorkommen kann, dass *keine* Transition aktiviert ist. In diesem Fall heißt das Netz **tot**; es kann seinen Zustand nicht mehr ändern und seine bisherige Schaltfolge kann nicht mehr verlängert werden.

Mit dieser Definition und Beschreibung kann man den zeitlichen Ablauf eines Petri-Netzes in der Zeichnung gut nachvollziehen und verfolgen.

Für das Beispielnetz aus Abbildung 5.7 werden wir das gleich durchführen, aber vorher müssen wir wieder betonen, dass die gerade präsentierte Definition 5.7 nur eine hilfreiche Erläuterung und nicht die *wirkliche* Definition des Ablaufs eines Petri-Netzes ist.

Die genaue und mathematisch zumindest leichter auswertbare Definition ist wieder die formale, die wir jetzt noch vor dem Beispiel angeben möchten.

Definition 5.8 Sei

$$PN = (S, T, SNT, TNS, M)$$

ein Petri-Netz. Dieses Petri-Netz erfährt eine schrittweise zeitliche Veränderung durch eine Operation, die das **Schalten einer Transition** genannt wird, und die wir hier definieren wollen.

Für jede Transition $t \in T$ setzen wir

$$\begin{aligned} \bullet t &:= \{ s \in S \mid (s, t) \in SNT \} && \text{(der **Vorbereich** von } t) \\ t^\bullet &:= \{ s \in S \mid (t, s) \in TNS \}. && \text{(der **Nachbereich** von } t) \end{aligned}$$

Die Stellen aus $\bullet t$ heißen **Eingangsstellen** von t und die Stellen aus t^\bullet heißen **Ausgangsstellen** von t .

Eine Stelle $s \in S$ heißt **markiert** unter M , falls $M(s) > 0$.

Eine Transition $t \in T$ heißt **aktiviert** oder **schaltbereit**, wenn jede Eingangsstelle von t markiert ist.

Eine Transition, die aktiviert ist, *kann* schalten (aber sie muss nicht). Wenn keine Transition aktiviert ist, dann heißt das Petri-Netz **tot** und verändert seinen Zustand nicht mehr. Wenn mehrere Transitionen gleichzeitig aktiviert sind, dann darf nur eine von ihnen schalten (wir sagen, die aktivierten Transitionen, wenn es mehrere sind, sind *in Konflikt* miteinander). Wenn nur eine Transition aktiviert ist, dann gibt es kein Problem und diese Transition wird irgendwann schalten, weil keine andere schalten kann.

In anderen Worten, bei jedem Schritt der dynamischen Entwicklung eines Petri-Netzes wird eine der aktivierten Transitionen ausgewählt (wenn es welche gibt) und die ausgewählte Transition t **schaltet**.

Das **Schalten** von t ersetzt die Markierung M durch eine neue Markierung M' , definiert durch

$$M'(s) := \begin{cases} M(s) - 1, & \text{falls } s \in \bullet t \setminus t^\bullet; \\ M(s) + 1, & \text{falls } s \in t^\bullet \setminus \bullet t; \\ M(s), & \text{sonst} \end{cases}$$

für jedes $s \in S$.

Sonst ändert sich nichts am Petri-Netz.

Diese Veränderung der Markierung durch die Schaltung von t notieren wir auch durch

$$M \xrightarrow{t} M'.$$

Für jede mögliche Folge

$$M_0 \xrightarrow{t_1} M_1, \quad M_1 \xrightarrow{t_2} M_2, \quad \dots \quad M_{k-1} \xrightarrow{t_k} M_k \quad (5.7)$$

von aneinander anschließenden Schaltungen von Transitionen von PN nennen wir die Folge

$$\sigma = t_1 t_2 \dots t_k$$

der dabei schaltenden Transitionen eine **Schaltfolge** von PN .

Man sieht sofort, dass die formale Definition die gleiche Bedeutung hat, wie die informale Definition des Schaltens für gezeichnete Petri-Netze.

Die Eingangsstellen einer Transition in der formalen Definition sind genau die Stellen, die vor der Transition liegen, und die Ausgangsstellen sind genau die Stellen, die hinter der Transition liegen. Eine Stelle heißt in der formalen Definition genau dann markiert, wenn sie in der informalen Definition Tokens enthält.

Somit entsprechen sich genau die Bedingungen für die Aktivierung einer Transition in beiden Definitionen, und beide Definitionen sagen das Gleiche darüber aus, wann Transitionen schalten.

Da die formale Markierung nur die Tokens in jeder Stelle zählt, ist die Wirkung der Schaltung einer Transition in beiden Definitionen gleich. Die Beschreibung dieser Wirkung entspricht sich genau in beiden Definitionen, außer in einem Fall, nämlich für Stellen, die gleichzeitig sowohl vor und hinter der schaltenden Transition t liegen. In der formalen Definition liegen solche Stellen in $\bullet t \cap t \bullet$ und gehören zum Fall „sonst“ in Gleichung (5.7), weshalb ihre Markierung sich nicht ändert. In der informalen Definition verlieren solche Stellen ein Token und bekommen gleichzeitig eines zurück, so daß auch hier die Anzahl der Tokens sich nicht ändert: die Wirkung ist gleich.

Und natürlich entspricht sich der Begriff einer Schaltfolge in beiden Definitionen.

Beispiel 5.9 In diesem Beispiel wollen wir die zeitliche Entwicklung des Petri-Netzes aus Abbildung 5.7 untersuchen.

Wir zeigen die verschiedenen Stadien in Abbildung 5.8 auf der nächsten Seite. Als Hilfe haben wir in jedem Bild die jeweils aktivierten Transitionen als *weißes* Rechteck gekennzeichnet.

In der Ausgangsmarkierung, die in Abbildung 5.8(a) gezeigt wird, ist nur Transition t_4 aktiviert.

Wenn diese Transition schaltet, erhalten wir den Zustand aus Abbildung 5.8(b), und jetzt ist Transition t_1 aktiviert.

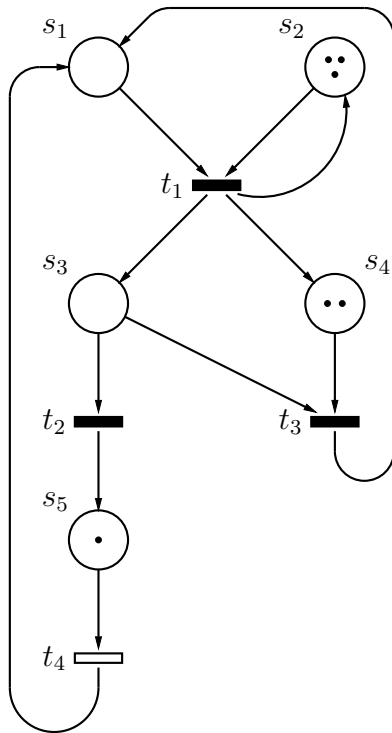
Wenn t_1 schaltet, dann erhält Stelle s_3 ein Token und Stelle s_4 erhält ein zusätzliches Token. Bei Stelle s_2 ändert sich nichts, weil diese Stelle sowohl eine Eingangs- wie auch eine Ausgangsstelle für t_1 ist. Diese Situation wird gezeigt in Abbildung 5.8(c)

Jetzt sind zwei Transitionen aktiviert: t_2 und t_3 . Nur eine kann schalten.

Wenn t_3 schaltet, dann verlieren s_3 und s_4 die Tokens, die sie gerade erworben hatten, und s_1 erhält wieder ein Token. Diese Situation hatten wir schon zwei Schritte zurück; es ist die Markierung aus Abbildung 5.8(b). Danach kann wieder t_1 schalten, und wir erhalten wieder den Zustand *vor* dem Schalten von t_3 .

Die Schaltfolge $t_3 t_1$ kann beliebig oft wiederholt werden, und es ändert sich nichts.

Wenn aber nach dem Schalten von t_1 anstelle von t_3 die andere aktivierte Transition t_2 schaltet, verliert s_3 sein Token, s_5 gewinnt ein Token und hat den gleichen Zustand wie am Anfang, aber s_4 steht nicht vor t_2 und muss deshalb das gerade gewonnene Token nicht aufgeben. Es entsteht der Zustand aus Abbildung 5.8(d), der fast so aussieht wie der Anfangszustand (und sich



(a) Anfangszustand

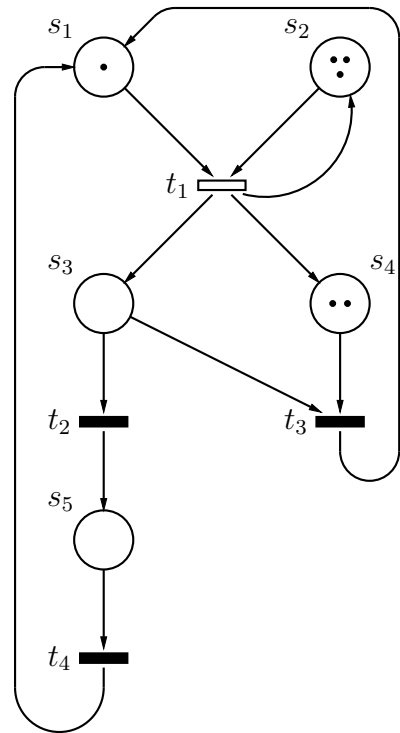
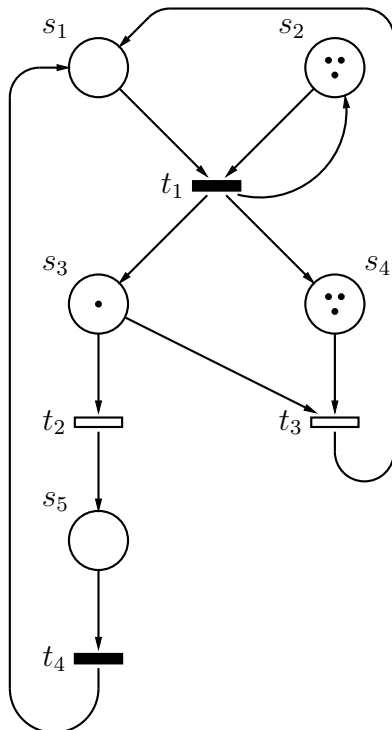
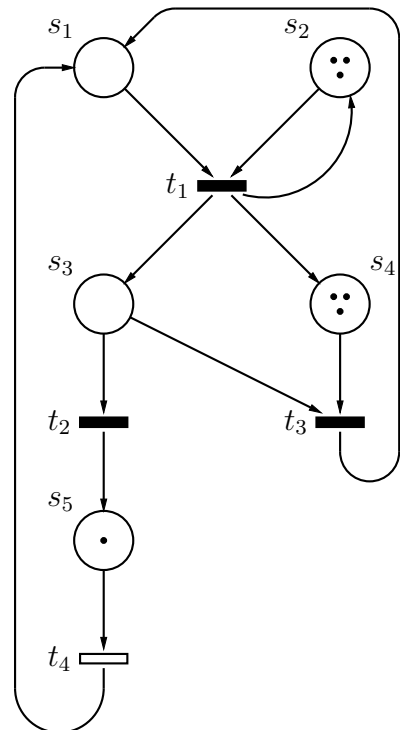
(b) Nach Schalten von t_4 (c) Nach Schalten von t_1 (d) Nach Schalten von t_2

Abbildung 5.8: Das Schalten des Petri-Netzes aus Abbildung 5.7

ähnlich weiterentwickeln kann), außer dass s_4 nun ein Token hinzugewonnen hat.

Die möglichen Schaltfolgen bestehen aus *einer* Ausführung von t_4t_1 und anschließend beliebige Wiederholungen von Zykeln $t_2t_4t_1$ oder t_3t_1 in beliebiger Reihenfolge. Jedes Mal, dass der Zyklus $t_2t_4t_1$ vollendet wird, erhält s_4 ein zusätzliches Token, so dass diese Stelle im Wesentlichen die Durchläufe durch diesen Zyklus mitzählt. Sonst gibt es keine dauerhaften Änderungen in den lokalen Zuständen.

Bevor wir uns einige wichtige Beispiele von Petri-Netzen oder Petri-Netz Abschnitten ansehen, wollen wir kurz etwas zum Problem der Parallelität in Verbindung mit Petri-Netzen sagen.

Die Tatsache, dass zu jedem Zeitpunkt nur eine Transition schalten darf, scheint in Widerspruch zu einer Verwendung von Petri-Netzen als Modell für parallele Prozesse zu stehen. Diese Einschränkung im Verhalten der Transitionen ist aber notwendig, weil Transitionen nur schalten können, wenn ihre Vorbedingungen erfüllt sind, und diese werden durch das Schalten *verbraucht*. Wenn mehrere Transitionen also von den gleichen Vorbedingungen abhängen, darf nur eine davon von der Vorbedingung profitieren.

Die Einschränkung könnte man theoretisch aufheben für Transitionen, die disjunkte Vorbereiche haben, aber das würde das Modell komplizierter machen, ohne einen wirklichen Gewinn zu bringen. Das wesentliche Merkmal von parallelen Prozessen ist nicht, dass sie zur gleichen Zeit stattfinden, sondern dass sie keine bestimmte Reihenfolge der Ausführung untereinander einhalten müssen und in diesem Sinne unabhängig voneinander ausgeführt werden können. Das lässt sich sehr wohl durch getrennte Zweige eines Petri-Netzes modellieren.

In einem echten, physikalisch existierenden Rechner ist es ohnehin sehr unwahrscheinlich, dass zwei Dinge wirklich zu genau der gleichen Zeit stattfinden. Auch zwei Prozesse, die nebeneinander her laufen, werden zu leicht unterschiedlichen Zeiten beginnen, und das kann man im Petri-Netz genau nachmachen, solange klar ist, dass die Prozesse sich während ihres Ablaufs nicht gegenseitig beeinflussen können (das Setzen der Ausgangszustände des einen Prozesses darf keine Auswirkung auf den anderen Prozess haben).

Petri-Netze modellieren ihre Prozesse so, als würde die Ausführung eines Prozessschrittes (also das Schalten einer Transition) *keine* Eigenzeit beanspruchen. Die Ergebnisse der Schaltung sind sofort da, und die nächste Transition kann sofort schalten (oder zumindest ist es so, dass die Schaltzeit eine Systemkonstante ist, nicht von Transition zu Transition variieren kann und keine wesentliche Eigenschaft einer Transition ist). Es gibt auch Varianten von Petri-Netzen, die „zeiterweitert“ sind und ihren Transitionen

spezifische Schaltzeiten zuordnen können, aber diese Varianten werden wir nicht besprechen.

Wir wollen uns jetzt einige Grundsituationen in Petri-Netzen ansehen.

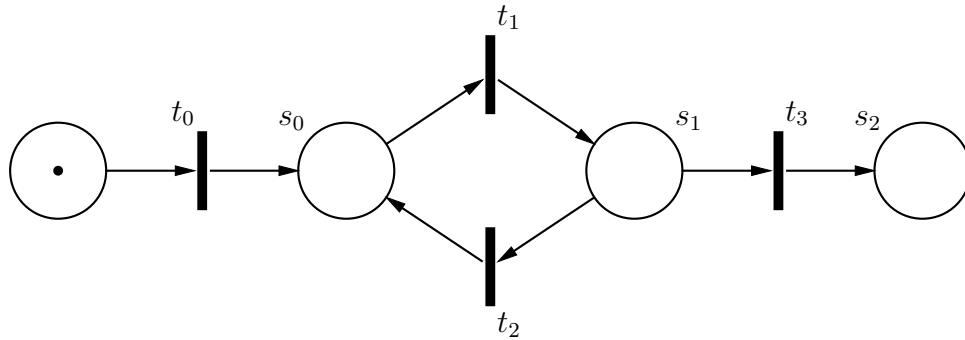


Abbildung 5.9: Eine Schleife in einem Petri-Netz

In Abbildung 5.9 schaltet zuerst Transition t_0 . Anschließend ist das Token an der ursprünglichen Stelle weg und ein Token erscheint in s_0 . Danach schaltet t_1 und das Token befindet sich in s_1 .

Hier entsteht ein Konflikt zwischen t_2 und t_3 ; beide sind aktiviert aber nur eine kann schalten.

Wenn t_2 schaltet, erscheint das Token wieder in s_0 und nur t_1 kann schalten, worauf das Token nach s_1 zurückkehrt. Diese Schleife t_2t_1 kann sich nun mehrmals wiederholen.

Wenn irgendwann t_3 statt t_2 schaltet ist die Schleife beendet, und das Petri-Netz „stirbt“ (wird tot) mit einem Token in s_2 .

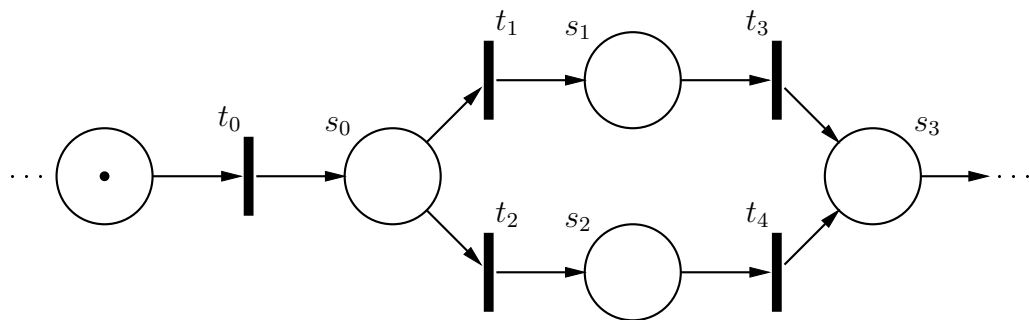


Abbildung 5.10: Alternative Pfade in einem Petri-Netz

Das Token in Abbildung 5.10 aktiviert zunächst Transition t_0 ; nachdem sie schaltet verschwindet das Token von der ursprünglichen Stelle und es

befindet sich ein Token in s_0 . Jetzt sind t_1 und t_2 in Konflikt; beide sind aktiviert, aber nur eine schaltet wirklich.

Wenn t_1 schaltet, dann wird das Token in s_0 gelöscht und in s_1 wird ein Token erzeugt. Anschließend kann nur t_3 schalten, s_3 erhält ein Token und danach wird der nach rechts aus dem Bild führende Pfad abgearbeitet.

Wenn statt t_1 die Transition t_2 schaltet, wird auch das Token in s_0 gelöscht und jetzt wird in s_2 eins erzeugt. Dann kann t_4 schalten, wieder erhält s_3 , die Stelle, an der die Pfade zusammenkommen, ein Token und die Verarbeitung wandert wie im ersten Fall nach rechts.

Wir sehen, dass im Moment, wo das Token die Stelle s_0 erreicht, das Petri-Netz sich für den einen oder den anderen der parallelen Zweige entscheiden muss. Wenn der obere Zweig ausgeführt wird, dann wird der untere nie erreicht, und umgekehrt.

In diesem Beispiel enthält jeder Zweig zwei Transitionen und eine Stelle, aber genau so gut könnten die Zweige aus längeren Folgen von Transitionen und Stellen bestehen, und nur eine dieser Folgen würde zur Ausführung gelangen, die andere nicht.

Trotz des parallelen Aussehens des Petri-Netzes in Abbildung 5.10 findet hier also *keine* parallele Verarbeitung statt.

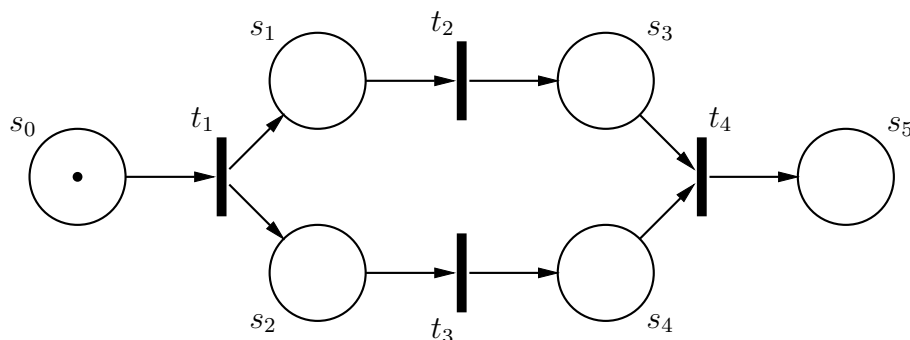


Abbildung 5.11: Parallele Pfade in einem Petri-Netz

Anders sieht es aus, wenn wie in Abbildung 5.11 die Verzweigung an Transitionen stattfindet. Zu Beginn ist t_1 aktiviert und schaltet. Danach haben Stellen s_1 und s_2 je ein Token.

Jetzt sind zwar t_2 und t_3 gleichzeitig aktiviert und deshalb in „Konflikt“, aber da ihre Aktivierung nicht von gemeinsamen Tokens abhängt, werden beide irgendwann schalten können — nur die Reihenfolge, in der sie schalten, ist nicht festgelegt.

Nehmen wir einmal an, dass t_3 zuerst schaltet (im anderen Fall ist die Fortentwicklung ähnlich). Dann enthält Stelle s_4 ein Token, aber Transition

t_4 , an der die Pfade wieder zusammenkommen, kann trotzdem erst schalten, wenn auch s_3 ein Token bekommt, und das passiert erst, wenn auch t_2 geschaltet hat.

Diese Überlegung hängt nicht davon ab, dass die parallelen Pfade je nur eine Transition enthalten. Auch wenn längere lineare Folgen von Stellen und Transitionen parallel verlaufen, aber die Verzweigung in diese Folgen und die Wiederzusammenfügung der Folgen wie in diesem Beispiel an Transitionen stattfinden, werden die Folgen *nebenläufig*, also parallel, abgearbeitet.

Sobald die Eingangstransition (im Bild t_1) schaltet, beginnen beide Pfade zu schalten, allerdings nicht gleichzeitig. Innerhalb jedes der beiden Pfade schalten die Transitionen in einer festen Reihenfolge, aber die Abarbeitung der beiden Pfade kann in beliebiger Reihenfolge hin- und herwechseln (mal zwei Transitionen des oberen Pfades, dann drei des unteren Pfades, dann eine des oberen Pfades, usw.).

Die Ausgangstransition (im Bild t_4) schaltet erst, wenn *beide* Pfade fertig sind, d.h., wenn die letzte Stelle beider Pfade ein Token erhält, auch wenn dies bei einem Pfad viel früher eintreten sollte, als beim anderen. Diese Transition **synchronisiert** also wieder beide Pfade und lässt ihre gemeinsame Fortsetzung erst laufen, wenn beide parallele Prozesse ihre Arbeit beendet haben.

Die parallele Abarbeitung von Prozessen hat große Vorteile (besonders wenn die nebenher laufenden Prozesse von unabhängigen Prozessoren ausgeführt werden können), aber sie birgt auch Gefahren und Komplikationen.

Das illustrieren wir an einem Beispiel. Man stelle sich vor, die Ruhr-Universität würde von einer Software-Firma ein neues E-Mail System kaufen, das den jetzigen Mail-Server ersetzt (weil dessen Sicherheitszertifikat abgelaufen ist und kein Geld dafür da ist, ein neues Zertifikat zu bestellen). Beim neuen E-Mail System muss man auf eine WWW Seite gehen, dort zunächst (A): die Adresse des Empfängers in ein Formular eintragen und abschicken, dann (B): den Text der Mail in ein Fenster eintragen oder hineinkopieren und abschicken, und (C): schließlich mit einem Mausklick bestätigen, dass die Mail tatsächlich versandt werden soll.

Für die Daten aus Schritt A und Schritt B gibt es jeweils eine zentrale Speicherstelle A bzw. B , auf die alle Benutzer des Mailsystems zugreifen, und aus der das Mail-Programm entnimmt, welchen Text es bei einer Bestätigung in Schritt C wohin schicken soll.

Auf dieses System greifen nun parallel zwei Benutzer zu: ein Student, der seiner Freundin (Adresse AS) einen sehr intimen Liebesbrief (Text BS) schreibt und dann mit einem Mausklick CS den Versandauftrag bestätigt, während etwa zur gleichen Zeit ein Professor einer Studentin (mit Adresse

AP) mitteilen will (Text BP), dass wegen Stellenabbaus seine Sprechstunden überlaufen sind und er sich leider nie wieder mit ihr treffen kann; den Versandauftrag bestätigt er mit Mausclick CP.

Wenn der Student und der Professor ihre Mails zu verschiedenen Zeiten eingeben, zum Beispiel wenn die Schrittfolge AS BS CS AP BP CP ist, dann geschieht alles wie vorgesehen: die Freundin erhält den Liebesbrief und die Studentin die unvermeidliche Abweisung. Was passiert aber, wenn beide Versender zur gleichen Zeit an ihren Terminals sitzen?

Dann könnte die Bearbeitungsreihenfolge vielleicht AS AP BP BS CS CP sein. Zum Zeitpunkt des Versands enthalten die Speicherstellen *A* und *B* die Daten, die zuletzt dort eingespeichert wurden, in diesem Fall also die Adresse der abgewiesenen Studentin in Speicherstelle *A* und den heißen Liebesbrief in Speicherstelle *B*. Diese Daten werden vom Studenten und vom Professor bestätigt, so dass beide jeweils mit ihrem eigenen Absender einen Liebesbrief an die eigentlich unglückliche Studentin schicken.

Viele andere Ergebnisse wären auch möglich, denn dieses System achtet nur darauf, dass jeder Benutzer für sich die Schrittreihenfolge einhält (weil die Fenster und Bestätigungsknöpfe nur in der vorgesehenen Reihenfolge angeboten werden), aber nicht darauf, dass eine sinnvolle Reihenfolge zwischen verschiedenen Benutzern eingehalten wird. Wir haben in Tabelle 5.4 auf der nächsten Seite *alle* möglichen Ausführungen des Versands der beiden Mails mit dem erzielten Ergebnis aufgeführt; aus Platzgründen mussten wir die Ergebnisse durch Abkürzungen XY mit zwei Buchstaben angeben.

Hier beschreibt X die versandte Nachricht (L für Liebesbrief oder K für Korb) und Y der Absender (S für den Studenten oder P für den Professor). Das Kürzel KS heißt zum Beispiel, dass die Empfängerin vom Studenten einen Korb erhält, LP hingegen, dass sie vom Professor einen Liebesbrief bekommt. Zwei Angaben getrennt durch ein Komma werden immer in der aufgeführten Reihenfolge versandt.

Wir sehen, dass *jede Überlappung* der Abläufe zu mindestens einer fehlgeleiteten Nachricht führt.

Das Problem liegt natürlich darin, dass das System eine ungeschützte Zugangsstelle benutzt, auf die ein zweiter Benutzer zugreifen kann, bevor der erste damit fertig ist. Ein vernünftiges Programm müsste jedem Benutzer, der einen Versandvorgang beginnt, die *ausschließliche* Benutzung dieser Speicherstellen gewähren, bis sein Brief versandt ist.

Wir bringen auf der nächsten Seite ein weiteres Beispiel mit zwei Schleifen, die auf eine gemeinsame Variable *z* zugreifen, deren anfänglicher Wert 0 ist.

Die beiden C-Programme enthalten ein paar einfache Merkmale, die wir bisher noch nicht eingeführt haben und die wir deshalb kurz erläutern: die Anweisung `z++` erhöht die Variable *z* um 1, und der `do`-Block mit der `while`-

Schrittfolge						Freundin erhält	Studentin erhält
AS	BS	CS	AP	BP	CP	LS	KP
AS	BS	AP	CS	BP	CP	nichts	LS, KP
AS	BS	AP	BP	CS	CP	nichts	KS, KP
AS	BS	AP	BP	CP	CS	nichts	KP, KS
AS	AP	BS	CS	BP	CP	nichts	LS, KP
AS	AP	BS	BP	CS	CP	nichts	KS, KP
AS	AP	BS	BP	CP	CS	nichts	KP, KS
AS	AP	BP	BS	CS	CP	nichts	LS, LP
AS	AP	BP	BS	CP	CS	nichts	LP, LS
AS	AP	BP	CP	BS	CS	nichts	KP, LS
AP	AS	BS	CS	BP	CP	LS, KP	nichts
AP	AS	BS	BP	CS	CP	KS, KP	nichts
AP	AS	BS	BP	CP	CS	KP, KS	nichts
AP	AS	BP	BS	CS	CP	LS, LP	nichts
AP	AS	BP	BS	CP	CS	LP, LS	nichts
AP	AS	BP	CP	BS	CS	KP, LS	nichts
AP	BP	AS	BS	CS	CP	LS, LP	nichts
AP	BP	AS	BS	CP	CS	LP, LS	nichts
AP	BP	AS	CP	BS	CS	KP, LS	nichts
AP	BP	CP	AS	BS	CS	LS	KP

Tabelle 5.4: Ergebnisse des Mail-Versands

Abfrage am Ende funktioniert wie die bisher betrachtete **while**-Schleife, außer dass die Abfrage nicht vor dem Eintritt in die Schleife stattfindet, sondern erst am Ende. Das heißt, dass die Schleife immer mindestens einmal durchlaufen wird, und dann wiederholt wird, solange die getestete Bedingung gilt.

In unserem Beispiel wird auf „!ende“ getestet, d.h., die Schleife wird wiederholt, bis der Benutzer es leid hat und die Programme unterbricht.

Programm I	Programm II
<pre>do { tuwas(); z++; } while(!ende)</pre>	<pre>do { output(z); z=0; } while(!ende)</pre>

Wenn Programm II läuft, gibt es den gegenwärtigen Wert von z aus, aber welcher Wert das ist, hängt von der Verzahnung der beiden Programme miteinander ab.

- Wenn Programm I einige Male ganz durchläuft und dann Programm II einen Durchlauf macht, wird ausgegeben, wie oft Programm I und insbesondere wie oft `tuwas` ausgeführt wurde. Anschließend wird `z` wieder auf 0 gesetzt und der gleiche Ablauf kann sich wiederholen.
- Wenn Programm I einige Male ganz durchläuft und einen weiteren Durchlauf beginnt, aber Programm II zwischen dem Befehl `tuwas()`; und der Anweisung `z++` ausgeführt wird, dann ist das Ergebnis um 1 kleiner als im ersten Fall; die letzte Ausführung von `tuwas` wurde in `z` noch nicht mitgezählt.

Wenn dann beim nächsten Mal Programm II nach einigen vollendeten Durchläufen von Programm I zur Ausführung kommt, ist die Zählung um 1 zu groß (weil Programm I beim vorigen Mal `z` direkt vor einer Erhöhung auf 0 gesetzt hat; der Zähler steht also schon vor dem ersten `tuwas` schon auf 1).

- Wenn Programm I zwar oft ausgeführt wird, aber immer nur zwischen den Befehlen `output(z)` und `z=0` von Programm II, dann wird immer 0 ausgegeben.

Natürlich sind noch etliche Kombinationen dieser Fälle möglich.

Es ist klar, dass dieses Problem daher rührt, dass zwei Programme auf eine *gemeinsam* benutzte Variable zugreifen, *und das diese Zugriffe unkontrolliert und vor allem unkoordiniert vonstatten geht*.

Nur das Teilen der Variablen unter zwei Programmen ist nicht das eigentliche Problem, denn anstelle der Variablen könnten wir eine Ressource wie zum Beispiel einen Drucker betrachten, die tatsächlich im System nur einmal vorhanden ist und die deshalb von mehreren Programmen geteilt werden *muss*. Das lässt sich auch sicher und gefahrlos machen, wenn dafür gesorgt ist, dass jeder Benutzer *für die Zeit seiner Benutzung* die alleinige Verfügung über die Ressource hat und von anderen Benutzern nicht gestört wird; sie müssen warten, bis die Ressource frei ist und erhalten dann selber die alleinige Verfügung über die Ressource.

Diese Reservierung einer Ressource für eine ausschließliche Benutzung durch einen einzigen Benutzer nennt sich ein **Lock** (sinnvollerweise ein **Schloss**, das man absperren kann). Nur wer momentan den Schlüssel besitzt, kann die Ressource benutzen; alle anderen sind ausgesperrt.

Es gibt verschiedene Ideen und Systeme dafür, wie man solche Locks einrichten und zuteilen kann, und wir möchten einige davon besprechen.

Ein Lock lässt sich leicht wie in Abbildung 5.12 auf der nächsten Seite mit einem Petri-Netz modellieren. Hier laufen zwei Prozesse oder Programme

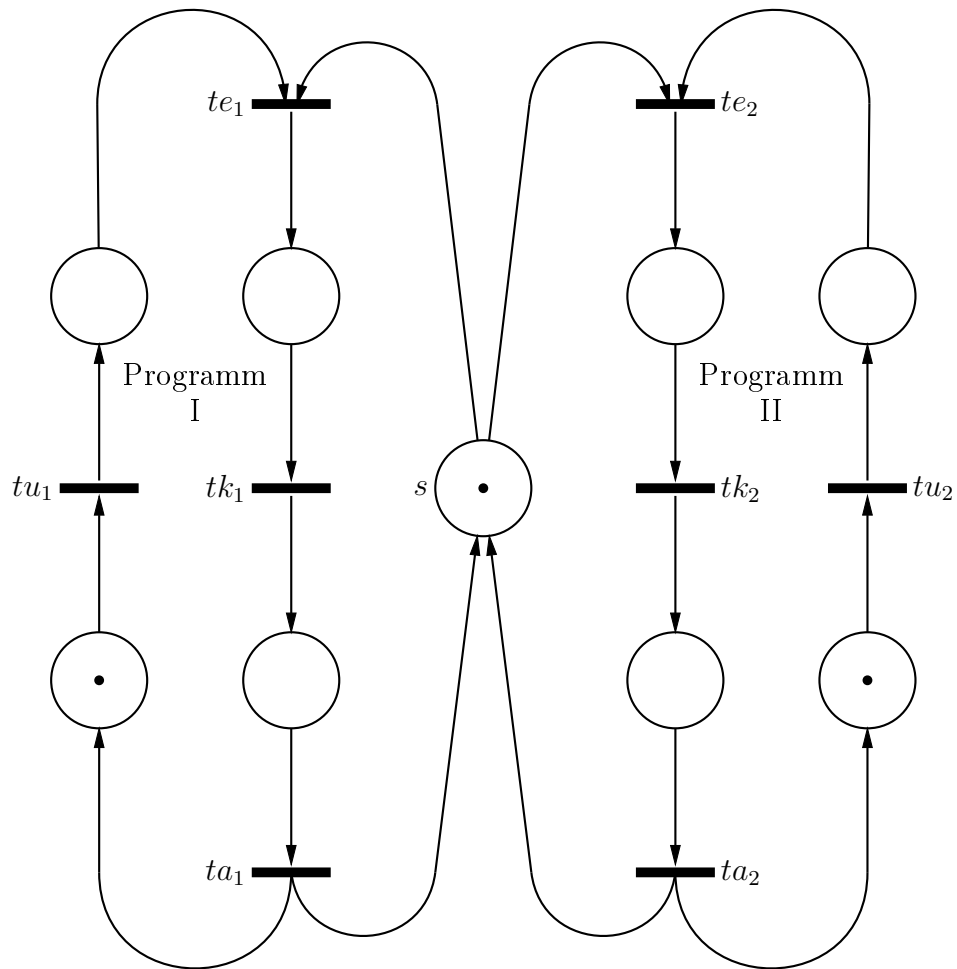


Abbildung 5.12: Ein Petri-Netz vom Typ „mutex“

parallel, dargestellt durch die Schleifen von Stellen und Transitionen auf der linken und der rechten Seite der Zeichnung. Jeder Prozess beansprucht eine gemeinsame Variable oder eine gemeinsame Ressource, dargestellt durch die Stelle s in der Mitte der Zeichnung, über die er in einem **kritischen Abschnitt** alleine verfügen muss. Außerhalb des kritischen Abschnitts benötigt der Prozess die Ressource nicht.

In der Zeichnung tragen alle Transitionen Namen, die ihre Funktion spiegeln, und einen Index, der kennzeichnet, zu welchem Prozess (1 oder 2) die Transition gehört.

Die Transitionen auf mittlerer Höhe stellen die Abschnitte der Programmausführung dar, wobei die tk genannte Transition im inneren Zweig des jewei-

ligen Prozesszyklus den **kritischen** Abschnitt repräsentiert und die Transition *tu* im äußeren Teil des Zyklus den **unkritischen** Abschnitt darstellt.

Die Transitionen *te* oben bilden den **E**ingang zum kritischen Abschnitt und die Transitionen *ta* unten den **A**usgang aus dem kritischen Abschnitt. Die Eingangstransitionen können nur schalten, wenn die in der Prozessschleife reisende Markierung, die den Programmablauf anzeigt, direkt vor dieser Transition steht (was bedeutet, dass der Prozess von seinem Ablauf her jetzt in den kritischen Abschnitt eintreten will), und wenn gleichzeitig die Stelle *s* markiert ist (was bedeutet, dass die Ressource frei ist).

Sobald die Transition *te* eines der beiden Prozesse schaltet, verschwindet die Markierung von *s*. Wenn ein Prozess sich im kritischen Abschnitt befindet, ist *s* also nicht markiert und der andere Prozess kann deshalb nicht in seinen eigenen kritischen Abschnitt eintreten.

Erst wenn der die Ressource beanspruchende Prozess seine Transition *ta* schaltet, erscheint wieder eine Markierung in *s*. Danach können wieder beide Prozesse, wenn sie so weit sind, in ihren kritischen Abschnitt eintreten (aber der erste, der es tut, löscht die Markierung von *s* und blockiert die Ressource für den anderen Prozess).

Diese Petri-Netz Struktur nennt sich ***mutex***. Das ist eine Abkürzung von „**mutual exclusion**“ oder *gegenseitiger Ausschluss*, weil die beiden Prozesse sich in ihren kritischen Abschnitten gegenseitig blockieren.

Petri-Netze sind nur Modelle, um Prozessabläufe sichtbar und theoretisch handhabbar zu machen. Deshalb müssen wir uns fragen, wie man die angegebene Struktur mit Programmmitteln wirklich realisieren kann.

Eine nicht sehr kluge Methode bestünde darin, dass jedes Programm, wenn es die Ressource beanspruchen will, nachschauen muss, ob sie frei ist (das kann man auf verschiedene Weisen durch eine hinterlassene „Nachricht“, ein gesetztes Status-Bit oder dergleichen, kennzeichnen). Wenn die Ressource frei ist, wird sie als besetzt markiert und dann verwendet. Wenn die Ressource nicht frei ist, wird die Nachfrage in einer Schleife dauernd wiederholt, bis der andere Benutzer die Ressource als frei markiert.

Diese ständige Nachfrage hat den großen Nachteil, dass sie viel Rechenzeit verbraucht aber trotzdem nichts produktives rechnet; sie vergeudet also die Rechnerleistung und verlangsamt dadurch andere Prozesse, die die Ressource gerade benutzen oder die sie gar nicht benötigen. Das verlängert auch die Zeit, bis das anfragende Programm die Ressource erhält.

Außerdem muss die Kennzeichnung für eine freie Ressource in dem Sinne geschützt sein, dass eine Abfrage mit Ergebnis „frei“ *im gleichen Moment* eine Neumarkierung als „belegt“ bewirkt, denn sonst kann zwischen dem Zeitpunkt, wo ein Prozess die Kennzeichnung „frei“ liest und dem darauf folgenden Zeitpunkt, wo dieser Prozess die Ressource wieder als belegt markiert,

ein zweiter Prozess die noch „freie“ Kennzeichnung lesen, und denken, dass er diese Ressource verwenden darf. Dann würden trotz Kennzeichnung zwei Prozesse gleichzeitig die Ressource zu benutzen versuchen, mit den bekannten Konsequenzen.

Ressourcenteilung sollte man deshalb besser nicht durch ständige Nachfrage auf ein „Freizeichen“ regeln. Eine effizientere und deshalb meistens angewandte Methode besteht darin, dass das nachfragende oder die Ressource anfordernde Programm sich „*schlafen*“ legt, wenn die Ressource belegt ist, und sich „*wecken*“ lässt, wenn sie wieder frei ist.

„Schlafen legen“ heißt einfach, dass das Programm seine Ausführung abbricht und die Kontrolle an das Betriebssystem zurückgibt, das die Aufgabe hat, verschiedenen gleichzeitig ablaufenden Prozessen kleine Happen an Rechenzeit zuzuteilen. Das Betriebssystem wird bei der Rechenzeitverteilung den schlafenden Prozess nicht mehr berücksichtigen, bis er wieder durch eine Mitteilung an das Betriebssystem „geweckt“ wird.

(Übrigens, jeder Rechner mit einem modernen Betriebssystem führt ständig eine große Anzahl von Prozessen aus, von denen die meisten fast immer schlafen. Dazu gehören neben der Regelung laufender Nutzprogramme solche Dinge wie die Tastatur- und Mausabfrage, Virenschutzprogramme, Programme, die im Hintergrund Aufgaben wie die Defragmentierung der Festplatte verrichten, wann immer der Rechner gerade nichts anderes wichtiges tut, die Bildschirmausgabe, Druckerspools, und vieles andere mehr; auf meinem Windows XP Rechner, während ich dies schreibe und ein Textfenster und ein Vorschaufenster geöffnet habe, sind es im Moment 58 Prozesse, die „gleichzeitig“ ausgeführt werden!)

In den frühen 60er Jahren hat der niederländische Informatiker Edsger Dijkstra ein Verfahren erfunden, um Ressourcen an anfragende Prozesse zu verteilen und das Schlafen und Wecken zu regeln.

Er lies sich dabei von den Eisenbahnsignalmasten zur Streckenfreigabe inspirieren (insbesondere von der Variante mit einem Signalarm, der gehoben und gesenkt wird) und nannte sein Freigabeinstrument *sempaal* oder auf englisch *semaphore*.

Ein **Semaphor** in Dijkstras System ist eine Variable S , die natürliche Zahlen als Werte annehmen kann, oder in einer anderen Variante *boolesche* Werte annimmt, d. h., nur zwei Werte, die wir für diese Anwendung **ein** und **aus** nennen wollen.

Die booleschen Semaphore funktionieren so, wie die Leuchtanzeigen an Flugzeugtoiletten oder wie eine rote Ampel (allerdings wie eine rote Ampel, die wenn sie auf grün schaltet nur ein Auto vorbeilässt, wie an den Autobahnauffahrten in Stoßzeiten) — wenn der Semaphor **ein** ist, ist die von ihm geschützte Ressource belegt oder gesperrt.

Semaphore mit Zahlenwerten hingegen (Dijkstras ursprüngliche Version), zählen einfach, wie viele Exemplare der Ressource noch verfügbar oder frei sind. Sie können demnach sinnvoll eingesetzt werden, wenn eine Ressource mehrmals vorhanden ist, aber jeder Benutzer ein Exemplar alleine für sich beansprucht (denken Sie zum Beispiel an Sitzplätze in einem Bus, Leihwagen aus einem kleinen Fuhrpark, Schalter in einer Bank oder in der Post und dergleichen).

Semaphore schützen ihre Ressourcen dadurch, dass der Zugang zur Ressource nur erlaubt ist, wenn der Semaphor **aus** oder positiv ist, und in einer semaphorgeschützten Umgebung halten sich Programme und Prozesse natürlich an diese Regelung (oder werden dazu gezwungen). Darüber hinaus sind Semaphore auch gegen Konflikte in der Verwendung des Semaphors geschützt, wie sie oben beschrieben wurden.

Benutzer dürfen nämlich nicht selber die Semaphore ein und ausschalten, sondern nur über besondere Prozeduren $p(S)$ und $v(S)$, die als einzige auf den Semaphor direkt zugreifen dürfen und die für alle anderen Benutzer als „Schränkwärter“ fungieren. Diese Prozeduren sind *ununterbrechbar*, d. h., sie führen Aufrufe immer vollständig aus, bevor einer von ihnen auf einen neuen Aufruf reagiert. So haben die Semaphore immer einen konsistenten Zustand und es kommt nicht zu Verwechslungen.

Die Prozeduren p und v verwalten eine **Warteschlange** für die von S geschützte Ressource auf folgende Weise (hier dargestellt für ein Semaphor, der boolesche Werte **ein** und **aus** annimmt):

Die Prozedur $p(S)$ macht

```

if (S==aus)
    S=ein;
else
    warte(S);                                [1]

```

Die Prozedur $v(S)$ macht

```

S=aus;
if (andererwartet(S)) {                    [2]
    S=ein;
    weckeanderen(S);                        [3]
}

```

Wir kommentieren kurz die Methoden, die von p und v in den nummerierten Zeilen aufgerufen werden (und die nicht „öffentlich“ verfügbar sind, sondern nur über diese Prozeduren aufgerufen werden können).

- [1] | Die Methode **warte** fügt den Prozess, der $p(S)$ aufgerufen hat, in die Warteschlange für S , und legt diesen Prozess dann schlafen (der Prozess stoppt und das Betriebssystem teilt ihm keine Rechenzeit zu).

Die Einträge in der Warteschlange sind natürlich nicht Programmabschnitte oder ganze Prozesse, sondern nur identifizierende Daten des Prozesses oder Zeiger auf die relevanten Prozessdaten.

- [2] | Die Methode **andererwartet** ermittelt und teilt durch einen booleschen Wert mit, ob die Warteschlange für S Einträge enthält.

- [3] | Die Methode **weckeanderen** weckt oder reaktiviert den schlafenden Prozess, dessen Eintrag an erster Stelle in der (nichtleeren) Warteschlange für S steht, zum Beispiel indem das Betriebssystem veranlasst wird, diesem Prozess wieder Rechenzeit zuzuteilen. Außerdem wird der Eintrag aus der Warteschlange entfernt.

Damit ist der Ablauf klar. Ein Prozess, der die zu S gehörende Ressource beansprucht, setzt einen Aufruf an $p(S)$ vor den kritischen Abschnitt, in dem er auf diese Ressource zugreift, und ruft $v(S)$ auf, wenn er mit der Ressource fertig ist.

Die Prozedur $p(S)$ prüft dann durch die Abfrage **S==aus** nach, ob die Ressource frei ist. Wenn ja, dann wird die Ressource durch **S=ein** für *andere* Anfragen als besetzt markiert und der aufrufende Prozess darf weiterlaufen. Wenn aber die Ressource nicht frei ist, dann wird der Zustand von S nicht verändert (weil die Ressource besetzt bleibt), die Anfrage wird in der Warteliste vermerkt, und der anfragende Prozess muss schlafen, bis er nach vorne gerutscht ist in der Warteliste und die Ressource nach nochmaligem Freiwerden ihm zugeteilt wird und er geweckt wird. In dem Moment, wo er geweckt wird, steht S schon auf **ein**, genau so, als wenn er gar nicht hätte warten müssen.

Wenn ein Prozess die Ressource gehabt hat und sie nicht mehr braucht, wird sie durch den Aufruf an $v(S)$ richtig weitergereicht. Die Prozedur v markiert die Ressource zunächst als frei durch die Anweisung **S=aus**. Dann schaut sie nach, ob Prozesse auf die Ressource warten. Wenn ja, wird die Ressource sofort wieder durch setzen des Semaphors als besetzt markiert und *einer* der wartenden Prozesse, in der Regel, der an erster Stelle in der Warteliste, wird geweckt und darf die Ressource benutzen.

Der geweckte Prozess verhält sich nach dem Wecken genau so, als hätte er die Ressource sofort bekommen. Der Schlafzustand wirkt sich nicht anders aus, als wenn der Aufruf an p einfach lange Zeit keinen Fortschritt gemacht hätte, d. h., der aufrufende Prozess muss bei der Verwendung von Semaphoren nichts anderes tun, als auf die Vollendung von p zu warten, egal wie lange es dauert.

Semaphore wie hier beschrieben werden natürlich beim Start des Betriebssystems oder des verwaltenden Programms für eine Ressource auf **aus** initialisiert (sobald die Ressource selber „betriebsbereit“ ist).

Die andere Variante von Semaphoren, die statt **ein** und **aus** natürliche Zahlen als Werte annehmen, funktioniert auf ganz ähnliche Weise. Hier wird der Semaphor auf einen positiven Wert initialisiert, der angibt, wie oft die Ressource vorhanden oder „zu vergeben“ ist.

Die Prozedur p prüft bei solchen Semaphoren nicht mehr nach, ob der Zustand **aus** ist, sondern ob er *positiv* ist, und wenn ja, dann wird das **einschalten** des Semaphors dadurch ersetzt, dass der Zahlenwert um 1 vermindert wird. Das entspricht der Vergabe *eines* Exemplars der Ressource.

Nur wenn S schon bei der Anfrage 0 war, kann der Wert nicht vermindert werden, und der anfragende Prozess kommt wie bei der ersten Variante in die Warteschlange und muss schlafen, bis er an die Reihe kommt.

Die Prozedur v wird auf ähnliche Weise modifiziert. Sie setzt S nicht auf **aus**, sondern erhöht den Zahlenwert des Semaphors um 1, was der Rückgabe *eines* Exemplars der Ressource entspricht und den Semaphorwert auf jeden Fall positiv werden lässt. Dann wird die Warteliste abgefragt. Wenn ein Prozess auf die Freigabe gewartet hat, dann wird der Semaphor wieder um 1 vermindert (was auf jeden Fall möglich ist, weil der Wert jetzt positiv ist, und was der Vergabe *eines* Exemplars der Ressource entspricht) und der vorderste Prozess in der Warteliste wird geweckt und darf die Ressource benutzen.

Die Namen p und v für diese Prozeduren sind überall üblich, obwohl sie etwas seltsam klingen. Es handelt sich hier um Dijkstras eigenen Bezeichnungen dafür: „ p “ ist der erste Buchstabe des von Dijkstra erfundenen zusammengesetzten Wortes *Prolaag*, eine Verschmelzung von „probeer te verlagen“ oder „versuche zu vermindern“ — wohlbemerkt: die Verminderung von S gelingt nur, wenn S vorher positiv war, und deshalb ist p wirklich zunächst nur ein Versuch und nicht immer eine Verminderung der Variablen. Das „ v “ der zweiten Prozedur ist der erste Buchstabe von *Verhoog*, das niederländische Wort für „Erhöhung“. Als erste Operation erhöht v immer den Wert des Semaphors.

Mit einem Semaphor s des booleschen Typs ist es ganz einfach, so etwas wie das **mutex**-Petri-Netz in einem Programm zu realisieren, nach folgendem

Schema:

S= aus;		Initialisierung
		(durch das Betriebssystem)
⋮		
do {	do {	
p(S);	p(S);	
kritischer Abschnitt ₁ ;	kritischer Abschnitt ₂ ;	
v(S);	v(S);	
unkritischer Abschnitt ₁ ;	unkritischer Abschnitt ₂ ;	
} while ...	} while ...	
Programm I	Programm II	

So können Programm I und Programm II problemlos parallel ausgeführt werden.

Prozesse in Betriebssystemen werden in der Regel auf diese Weise synchronisiert.

Wenn bei der parallelen Verarbeitung mehrere Ressourcen im Spiel sind, müssen auch mehrere Semaphoren gleichzeitig benutzt werden. Dabei kann ein Phänomen auftreten, das **Deadlock** genannt wird, hier zum Beispiel mit zwei Semaphoren S und T :

p(S)	p(T)
p(T)	p(S)
⋮	⋮
v(T)	v(S)
v(S)	v(T)
Programm I	Programm II

Die parallele Verarbeitung beider Programme bedeutet, dass innerhalb jedes Programms die Befehle in der gegebenen Reihenfolge ablaufen müssen, aber dass die Programme sich in der Abarbeitung beliebig oft abwechseln dürfen. Das gilt *auch wenn Semaphore verwendet werden*, solange kein Programm auf die Vollendung eines p -Aufrufs warten muss.

Wir nehmen an, dass bei Beginn der Programme beide geschützte Ressourcen frei sind. Dann können *ein* Aufruf an $p(S)$ und *ein* Aufruf an $p(T)$ ohne Warten ausgeführt werden, und wenn diese Aufrufe aus dem *gleichen* Programm kommen, gibt es keine Probleme. Dieses Programm blockiert beide Ressourcen, gibt sie später aber mit den entsprechenden v -Aufrufen wieder frei, und dann kann das andere Programm die Ressourcen verwenden.

Es kann aber auch vorkommen, dass beide Programme ihre erste Zeile ohne Warten ausführen können (denn die ersten Zeilen enthalten einen Aufruf an $p(S)$ und einen Aufruf an $p(T)$), und erst danach versucht jedes Programm, seine zweite Zeile auszuführen.

Der Aufruf von $p(T)$ in Programm I ist dann blockiert, weil Programm II in seiner ersten Zeile den Semaphor schon belegt hat. Also legt sich Programm I schlafen. Der Aufruf von $p(S)$ in Programm II ist aber auch blockiert, weil Programm I in seiner ersten Zeile diesen Semaphor belegt hat. Also legt sich Programm II auch schlafen.

Nun schlafen beide Programme und werden nie geweckt, weil sie ihre zweiten p -Aufrufe erst beenden können, wenn das jeweils andere Programm seine letzte Zeile ausgeführt hat und den Semaphor wieder frei gibt. Das andere Programm erreicht diese Zeile nie, weil es ja auch blockiert ist.

Ein **Deadlock** ist eine Pattsituation, aus der es kein Entkommen gibt, und so etwas haben wir hier. Ein anderer Name dafür ist **deadly embrace**, die tödliche Umarmung. Beide Programme sind so ineinander geklammert, dass sie sich nicht mehr lösen können.

Die Lösung für dieses Problem ähnelt sehr der Idee, die den Semaphoren überhaupt zu Grunde liegt. Deadlock entsteht, weil bei der Beantragung mehrerer Semaphore ein Programm dem anderen „dazwischenfunken“ kann, und Semaphore schützen gegen solches Verhalten, indem sie eine Folge von Befehlen in einen unzerbrechlichen Kasten packen, und mit ihren p - und v -Aufrufen dafür sorgen, dass die Befehlsfolge nicht unterbrochen werden kann und in einem Zug ausgeführt wird.

Genau so müsste hier der Block von aufeinanderfolgenden p -Aufrufen mit mehreren Semaphoren geschützt werden. Das kann man erreichen, wenn man p - und v -Operationen zulässt, die mehrere Semaphore in einem einzigen ununterbrechbaren Vorgang gleichzeitig schalten können. Solche Semaphore wurden von Dijkstra berücksichtigt und detailliert beschrieben.

Die Arbeitsweise der p - und v -Prozeduren wird bei dieser Variante wesentlich komplizierter, als wenn nur ein Semaphor zu verwalten ist.

Wir beschreiben die Funktion von Aufrufen $p(S_1, \dots, S_n)$ und $v(S_1, \dots, S_n)$ unter der Annahme, dass alle beteiligten Semaphore S_i vom booleschen Typ mit zwei Zuständen sind; die Zahlenvariante lässt sich daraus leicht herleiten.

Nach wie vor muss für jeden Semaphor eine eigene Warteliste eingerichtet werden, die andere Prozesse enthalten kann, als die Wartelisten für die anderen Semaphore. Denn nicht alle Prozesse werden die gleichen Ressourcen beanspruchen; man muss deshalb davon ausgehen, dass die Aufrufe an p und v nicht immer gleich sind, sondern für jeden Prozess eine andere Auswahl der beteiligten Semaphore betreffen, die sich aber für verschiedene Prozesse

überschneiden können.

Die Prozedur $p(S_1, \dots, S_n)$ überprüft zuerst, ob *alle* genannten $S_i = \mathbf{aus}$; in diesem Fall werden sie alle auf **ein** gesetzt und der aufrufende Prozess darf weiterlaufen, unter Verwendung der beantragten Ressourcen.

Wenn aber manche S_i schon beim Aufruf **ein** waren, dann kommt der aufrufende Prozess in die Warteliste für eines dieser S_i , und der Prozess legt sich schlafen. Es reicht, den Prozess in nur eine Warteliste einzutragen, weil er ohnehin erst geweckt werden darf, wenn dieser Semaphor **ausgeschaltet** ist.

Der Aufruf von $v(S_1, \dots, S_n)$ setzt alle genannten $S_i = \mathbf{aus}$. Anschließend werden die Wartelisten $W(S_i)$ der S_i der Reihe nach untersucht, zuerst die Warteliste für S_1 , dann die für S_2 , usw.

In jeder Warteliste werden die Prozesse untersucht, bis ein Prozess gefunden wird, für den *alle* Semaphore in seinem (ruhenden) p -Aufruf jetzt auf **aus** stehen.

Werden bei dieser Untersuchung Prozesse betrachtet, bei denen manche beanspruchten Semaphore noch auf **ein** stehen, dann wird jeder solcher Prozess in die Warteliste eines dieser auf **ein** stehenden Semaphore nachgetragen (wenn er dort nicht schon eingetragen ist). Andere Wartelisten werden nicht verändert. Wieder gilt, dass die Eintragung in eine Warteliste ausreicht, weil der Prozess erst aufwecken darf, wenn auch dieser Semaphor **aus** ist.

Wird in $W(S_i)$ ein Prozess gefunden, deren beanspruchten Semaphore alle auf **aus** stehen, dann wird der Prozess aus allen Wartelisten entfernt, in denen er steht, die von diesem Prozess angesprochenen Semaphore (auch die, die nicht zu den oben genannten S_i gehören) werden alle auf **ein** gesetzt, der Prozess wird als „startbar“ markiert, und die nächste Warteliste $W(S_{i+1})$ kommt unter Betrachtung (wenn $i < n$ war). Auch wenn kein startbarer Prozess in $W(S_i)$ gefunden wird kommt die nächste Warteliste an die Reihe.

Dieses Verfahren wird fortgesetzt, bis alle $W(S_i)$ untersucht wurden. Dann werden die gesammelten als „startbar“ markierten Prozesse geweckt und dürfen weiterlaufen. Ihr p -Aufruf ist damit beendet, und die beanspruchten Ressourcen sind für sie jetzt reserviert und dürfen benutzt werden.

Weil bei der Entdeckung eines startbaren Prozesses die von ihm beanspruchten Semaphore sofort auf **ein** gesetzt werden, kann danach kein anderer Prozess, der auch diese Semaphore benutzt, als startbar markiert werden. Das heißt, wenn mehrer Prozesse startbar sind, dann ist sichergestellt, dass sie disjunkte Mengen von Semaphore beanspruchen.

Bei der oben beschriebenen Untersuchung der Warteschlangen werden Prozesse in $W(S_i)$, die *hinter* einem startbaren Prozess liegen, übergangen. Solche Prozesse können nicht gestartet werden, weil ein anderer Prozess vor ihnen den Semaphor S_i wieder belegt hat. Und es ist nicht nötig, die Warte-

schlangebelegung dieser Prozesse zu aktualisieren, weil sie ja ohnehin in der Warteschlange für S_i bleiben und deshalb irgendwann wieder erfasst werden. Die Wiedererfassung kommt auch nicht später als unbedingt nötig, denn diese Prozesse können sowieso erst gestartet werden, wenn S_i wieder auf **aus** gesetzt wird und die Warteliste $W(S_i)$ noch einmal durchforstet wird.

Die nichtstartbaren Prozesse, die *vor* der Entdeckung eines startbaren Prozesses untersucht werden, müssen in einer anderen Warteliste stehen und notfalls dort nachgetragen werden, weil zu diesem Zeitpunkt S_i auf **aus** steht und vielleicht immer so bleiben wird. Dann würde nie wieder ein S_i betreffender v -Aufruf kommen, so dass diese immer noch schlafenden Prozesse über die S_i -Warteschlange nicht mehr geweckt werden könnten; damit sie doch irgendwann geweckt werden, müssen sie also auch in anderen Warteschlangen stehen.

Es ist nicht wichtig, dass die Warteschlangenbelegung genau stimmt und vor allem ist es nicht schädlich, wenn ein schlafender Prozess in der Warteschlange eines auf **aus** stehenden Semaphors zu unrecht stehen bleibt; die Warteschlange wird ohnehin bereinigt, wenn der Prozess geweckt wird.

Es ist auch nicht erforderlich, dass ein Prozess in allen Warteschlangen vertreten ist, die zu **eingeschalteten** von ihm beanspruchten Semaphore gehören. Damit er nicht vergessen wird, reicht es, wenn er in *einer* solchen Warteschlange steht, weil er ohnehin nicht geweckt werden darf, so lange dieser Semaphor noch auf **ein** steht.

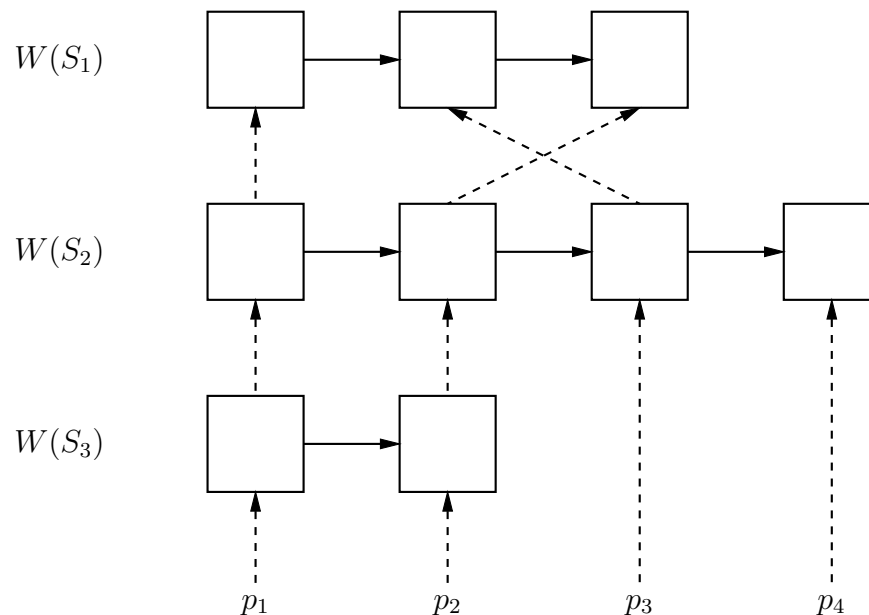
Schädlich ist nur, wenn *aktive* Prozesse in Warteschlangen vorkommen, denn sie müssen nicht mehr und dürfen nicht mehr geweckt werden, bevor sie sich durch einen neuen p -Aufruf eventuell wieder schlafen legen. Aus diesem Grund müssen startbar gemacht werdende Prozesse aus allen Warteschlangen entfernt werden.

Auch die gleichzeitige Verwaltung von mehreren Semaphoren mit p - und v -Aufrufen löst noch nicht alle Probleme, die bei parallelen Prozessen auftreten können. Im obigen System werden Prozesse geweckt, wenn alle ihre Semaphore **aus** sind, oder genauer, wenn als Erstes festgestellt wird, dass alle ihre Semaphore **aus** sind, unabhängig davon, wo diese Prozesse in ihren Wartelisten stehen oder ob andere Prozesse, die auch berechtigt wären, vielleicht länger auf die Freigabe warten, als sie. Die Reihenfolge des Weckens hängt von der Reihenfolge ab, in der die Warteschlangen untersucht werden. Ein Prozess, der an zweiter Stelle in $W(S_1)$ und an zehnter Stelle in $W(S_2)$ steht erhält den Vortritt vor einem Prozess, der an dritter Stelle in $W(S_1)$ und an erster Stelle in $W(S_2)$ steht, oder der in $W(S_1)$ vielleicht gar nicht eingetragen ist.

Die Chancen für eine gerechte Behandlung können erhöht werden, wenn Prozesse in die Wartelisten für alle auf **ein** stehende Semaphore eingetra-

gen werden, die sie beanspruchen, statt nur in eine solche Warteliste, wie wir es oben vorgeschlagen haben. Das ist zwar nicht nötig für das rechtzeitige Wecken, aber macht es wahrscheinlicher, dass der Prozess vor konkurrierenden Prozessen bearbeitet wird, wenn die Voraussetzungen erfüllt sind, dass er starten könnte.

Eine geeignete Datenstruktur für die Verwaltung von Prozessen in mehreren Wartelisten sind **Multi-Listen**, in denen jede Warteliste als eine Datenliste geführt wird, aber die Prozesse auch durch eine Kette von Zeigern mit ihren Einträgen in den Wartelisten verbunden werden, wie in folgendem Diagramm:



Ansonsten kann ein Prioritätensystem und eine umfangreichere Untersuchung der Warteschlangen helfen, die insgesamt am längsten wartende Prozesse als erste zu starten.

Auch diese Maßnahmen können folgendes Phänomen des **Verhungerns** nicht ganz unterbinden. Ein Prozess X , der zwei Semaphore S_1 und S_2 belegen muss, könnte unter Umständen *nie* gestartet werden, wenn andere Prozesse *nur* S_1 und *nur* S_2 abwechselnd so belegen, dass immer mindestens einer dieser Semaphore auf **ein** steht.

Wann immer einer der Semaphore S_i frei wird, kann X nicht gestartet werden, weil der andere Semaphor S_{3-i} belegt ist. Also wird ein Prozess gestartet, der nur S_i beansprucht, und wenn später der andere Semaphor frei wird, steht S_i auf **ein** und X kann wieder nicht gestartet werden. Das Spiel kann sich auf ewig so fortsetzen.

Gegen das Verhungern helfen nur Maßnahmen, die Prozessen eine absolute Priorität einräumen, wenn sie lange genug gewartet haben, so dass andere „jüngere“ Prozesse sogar dann zum Warten gezwungen werden, wenn alle ihre Semaphore **aus** sind. Man sieht, dass die Verwaltung von mehreren Semaphoren auf einmal kein einfaches Problem ist.

Manche Anwendungen von Semaphoren können etwas kontraintuitiv verlaufen. Wir wollen deshalb noch in leicht vereinfachter Form ein Beispiel von Prof. Dr. Lutz Richter in Zürich besprechen, bei dem die Rollen der p - und v -Aufrufe nicht so sind, wie man zunächst erwarten würde. Das Beispiel zeigt auch, dass gepaarte p - und v -Aufrufe nicht in jedem Fall wie in den bisherigen Beispielen vom gleichen Prozess abgegeben werden müssen, sondern auch von verschiedenen miteinander kooperierenden Prozessen kommen können.

Der Grundgedanke der richtigen Anwendung von Semaphoren ist nicht, dass Prozesse, die etwas Anfordern, p -Aufrufe abschicken, sondern, dass Prozesse, die auf ein bestimmtes Ereignis *warten* müssen, die Originatoren solcher Aufrufe sind.

Beispiel 5.10 In diesem Beispiel untersuchen wir das Zusammenwirken eines Gerätetreibers mit dem Gerät, für das er zuständig ist, und mit Anwenderprogrammen, die dieses Gerät benutzen wollen.

Die Anwenderprogramme beanspruchen zwar das Gerät, aber nur indirekt über den Treiber. Weil der Treiber für die richtige und konfliktfreie Weitergabe von Aufträgen an das Gerät sorgt, brauchen die Anwenderprogramme sich nicht darum zu kümmern und sie benötigen auch keinen Semaphor für das Gerät.

In der Regel können sie sogar mit anderen Aufgaben problemlos weiterrechnen, während ihr Auftrag erledigt wird. Erst wenn sie einen Bearbeitungspunkt erreichen, wo sie ein Ergebnis vom Gerät brauchen oder sicher sein müssen, dass das Gerät seine Aufgabe erfüllt hat, müssen sie eventuell eine Pause einlegen.

Der Treiber hingegen muss ständig warten. Solange keine Aufträge vorliegen, hat er nichts zu tun und darf deshalb nicht in Aktion treten. Wenn Aufträge vorliegen, kann er sie abwickeln, aber er muss dafür sorgen, dass das Gerät bereit ist, wenn er es anspricht. Deshalb ist auch der Treiber und nicht das Anwendungsprogramm die Instanz, die darauf warten muss, dass das Gerät den letzten Auftrag beendet hat, oder die feststellen muss, ob das Gerät noch beschäftigt ist.

Der Treiber läuft ständig in einer Schleife, in der, wenn Aufträge angemeldet werden, die richtigen Schritte durchgeführt werden, um den Auftrag vom Gerät ausführen zu lassen, und in dem die Erledigung des Auftrags abgewartet und an den Auftraggeber gemeldet wird, bevor weitere Aufträge

eine Mitteilung, oft durch ein in Hardware implementiertes Unterbrechungssignal, wenn das Gerät mit der Bearbeitung der Daten fertig ist.

Wenn der Treiber sinnvolle Aufgaben erledigen kann, die nicht von der Vollendung des Auftrags abhängen, kann er weiterlaufen, und danach ruft er $p(\text{Interrupt})$ auf, um sich schlafen zu legen, falls das Gerät die Aufgabe noch nicht erledigt hat.

Ein Hardwareinterrupt vom Gerät wird von Software abgefangen, die den Semaphor **Interrupt** durch einen v -Aufruf auf **aus** schaltet, wenn das Gerät dies nicht selber erledigen kann. Der p -Aufruf des Treibers wird erst dann vollendet, wenn **Interrupt** auf **aus** geschaltet worden ist. Bei der Vollendung setzt der p -Aufruf den Semaphor für die nächste Verwendung wieder auf **ein**.

Auch das Anwenderprogramm benötigt in der Regel irgendwann eine Bestätigung, dass der Auftrag bearbeitet wurde. Dies geschieht wie in den anderen Fällen unter Verwendung eines auf **ein** initialisierten Semaphors **fertig**.

Der Treiber teilt die Vollendung des Auftrags mit durch einen Aufruf von $v(\text{fertig})$, der den Semaphor **aus**schaltet. Das Anwenderprogramm fragt die Vollendung des Auftrags ab durch einen Aufruf von $p(\text{fertig})$, und legt sich notfalls schlafen, bis die Bestätigung kommt. Anschließend steht **fertig** wieder auf **ein**.

Semaphore funktionieren in der Tat wie Eisenbahnsignale und können „Strecken“ im Programm sperren oder freigeben, aber sie sind nicht geeignet, um komplizierte Verwaltungsaufgaben in Verbindung mit dem Zugang zu Ressourcen oder der Freigabe von Ressourcen zu bewältigen.

Für diese Zwecke gibt es ein Konzept, das die ganze Verwaltung von einer Ressource oder von miteinander verbundenen Ressourcen in einen „schwarzen Kasten“ packt, der anders als die Semaphore die einzelnen Details der Ressourcenverwaltung versteckt und nur geregelte und überwachte Zugänge zu den Ressourcen erlaubt, die neben dem Schutz der Benutzer vor gegenseitiger Störung auch beliebige andere nötige Vorbereitungs- und Verwaltungsarbeiten erledigen können, ohne dass der Benutzer über deren Details Bescheid wissen muss.

Dieses Modell erfüllt ähnliche Aufgaben wie der Verwalter eines Diskussionsforums im Internet, der Beiträge prüfen und notfalls redigieren kann, bevor sie zugelassen werden und der Beiträge auch ablehnen kann, Zugangsrechte zum Lesen und Schreiben von Beiträgen verteilen kann, und dergleichen. Solche Verwalter heißen auf Englisch „**Monitore**“, also „Aufpasser“, und das dieser Tätigkeit angelehnte Ressourcenverwaltungskonzept heißt entsprechend das **Monitor**-Konzept.

Ein Monitor verwaltet einen Satz D von Daten, die von mehreren Instanzen gemeinsam benutzt werden. Wie auch bei Semaphoren, führt der Monitor eine **Warteliste** für den Zugriff auf die Daten und stellt sicher, dass zu jedem

Zeitpunkt nur eine Instanz auf die Daten zugreifen kann.

Zusätzlich regelt der Monitor nicht nur wann, sondern auch *wie* auf die Daten zugegriffen werden kann, indem er selber Eingänge f_1, f_2, \dots, f_n anbietet, die mit den Daten verschiedene Operationen ausführen. Alle gewünschte und zulässige Operationen sind unter den f_i vertreten, und der Zugang auf die Daten kann nur über diese Eingänge oder „Entries“ geschehen.

Diese Eingänge entsprechen im Wesentlichen die Prozessabschnitte zwischen p - und v -Aufrufen bei der Verwendung von Semaphoren.

Der Monitor muss natürlich erstellt und eingerichtet werden, bevor er benutzt werden kann, und dazu führt er beim Starten einige Initialisierungen aus, um die Entries betriebsbereit zu machen und die Daten auf ihren Urzustand zu setzen.

Der Gesamtaufbau eines Monitors kann schematisch so dargestellt werden:

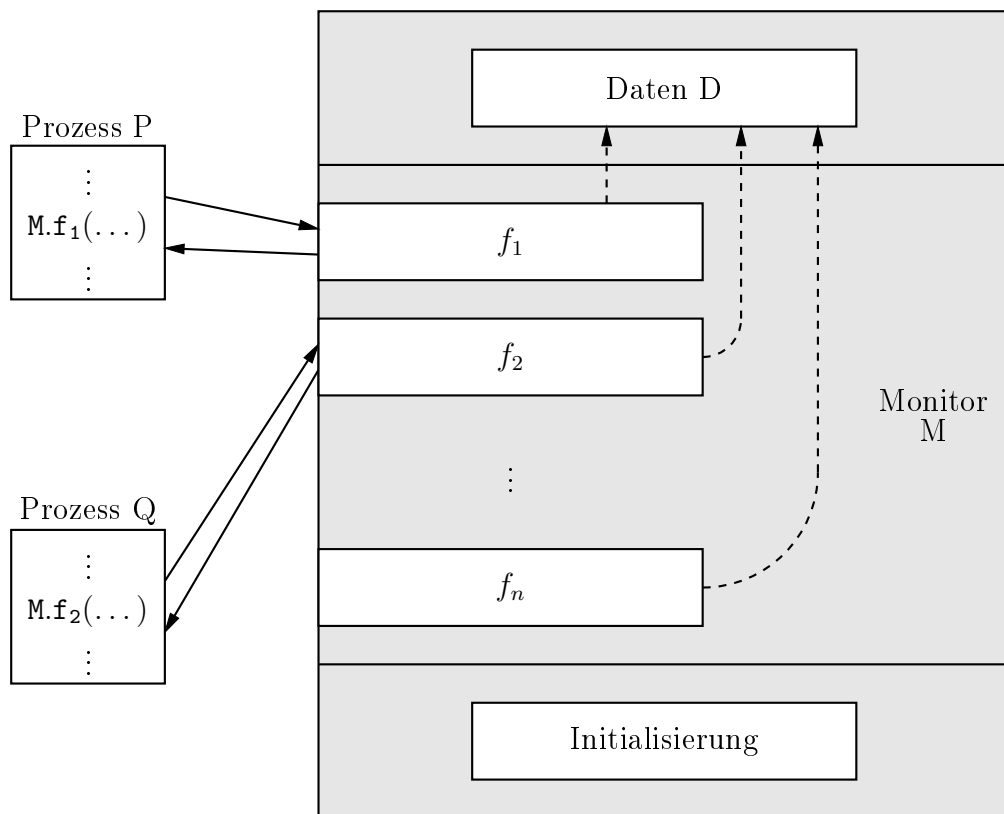


Abbildung 5.14: Schematischer Aufbau eines Monitors

Der Zugriff auf eine der zulässigen Operationen eines Monitors geschieht

durch den Aufruf einer Methode

`MonName.EntryName(argument_list)`

die für jeden Eingang definiert ist. Der Monitor sorgt dafür, dass zu jedem Zeitpunkt höchstens eine Instanz einer dieser Methoden ausgeführt wird. Andere in dieser Zeit eingehende Aufrufe müssen warten und werden schlafen gelegt.

Nicht jede zugelassene Operation kann zu jedem Zeitpunkt sinnvoll ausgeführt werden, und es kann zeitweilig vorkommen, dass manche Operationen machbar sind und andere nicht. Deshalb können die einzelnen Entries Bedingungsvariablen V implementieren, die angeben, ob die Voraussetzungen für eine Operation erfüllt sind.

Für diese Bedingungsvariablen werden eigene Wartelisten geführt und es werden entsprechende Methoden implementiert:

V.wait: fügt den aufrufenden Prozess in die Warteschlange für V ein und legt den Prozess schlafen;

V.signal: entfernt *alle* Prozesse aus der Warteschlange für V und überträgt sie in die Hauptwarteschlange des Monitors.

Bei einem Aufruf eines Prozesses an ein Entry des Monitors kommt der Prozess zuerst in die Hauptwarteschlange des Monitors, denn es werden vielleicht noch vorherige Aufrufe eines der Entries gerade ausgeführt.

Der Monitor arbeitet die Hauptwarteschlange der Reihe nach ab (wenn sie nichtleer ist) und schickt den vordersten Prozess zurück an den vom Prozess aufgerufenen Eingang f_i zur Vervollständigung der gewünschten Operation.

Der Eingang prüft als Erstes die eventuellen Vorbedingungen für seine Ausführbarkeit. Wenn die Vorbedingungen nicht erfüllt sind, wird für diesen Prozess **V.wait** ausgeführt mit der oben beschriebenen Wirkung.

Wenn ein Prozess die obige Prüfung überstanden hat und durchgelassen wurde, wird die gewünschte Operation zu Ende geführt. Anschließend wird der Monitor wieder freigegeben und der nächste Prozess aus der Hauptwarteschlange wird bedient.

Bei der Ausführung der Monitoroperationen ändert sich der Zustand der Vorbedingungen für die Entries ständig. Jedes Mal, dass eine Vorbedingung V gültig wird, wird **V.signal** ausgeführt, um die auf die Bedingung wartenden Prozesse zu wecken und wieder in die Hauptwarteschlange einzufügen.

Wenn ein solcher geweckter Prozess in der Hauptwarteschlange nach vorne gerutscht ist, wird das von ihm aufgerufene Entry an der Stelle fortgesetzt, wo es unterbrochen wurde, nämlich nach dem Aufruf von **V.wait**. In der Zwischenzeit kann die Vorbedingung wieder ungültig geworden sein, während der

Aufruf in der Hauptwarteschlange gewartet hat und andere Aufrufe ausgeführt wurden. Deshalb muss die Vorbedingung *wieder* geprüft werden; ist sie nun ungültig, wird wieder `V.wait` ausgeführt und der Aufruf kommt wieder in die Warteschlange für die Vorbedingung *V*.

Nur wenn die Vorbedingung bei der erneuten Prüfung immer noch gilt, kann die Operation fortgesetzt und zu Ende geführt werden.

Gerade weil die Bedingungen in den Entries sich ändern können, während ein Prozess schläft, ist dafür zu sorgen, dass bei jedem Wecken eine neue Prüfung der Bedingungen erfolgt. Aus diesem Grund muss `V.wait` immer innerhalb einer **while**-Schleife ausgeführt werden, die die Bedingung bei jedem Durchgang prüft, und nicht innerhalb eines **if**-Blockes, das die Bedingung nur einmal prüft.

Man sieht, dass unter Umständen ein Prozess mehrmals zwischen der Warteschlange für eine Vorbedingung und der Hauptwarteschlange hin- und herwandern kann, bevor die von ihm angeforderte Operation ausgeführt werden kann. Auch „Verhungern“ ist durchaus möglich.

Prozesse verlassen die Warteschlange für eine Vorbedingung dadurch, dass die Vorbedingung in Erfüllung geht. Auch wenn mehrere Prozesse warten, werden sie alle gleichzeitig in die Hauptwarteschlange bewegt. Die Reihenfolge, in der sie in der Hauptwarteschlange zu stehen kommen, lässt sich beliebig einstellen, zum Beispiel um Prioritäten zu berücksichtigen (wer wartet insgesamt am längsten?).

Prozesse rutschen in der Hauptwarteschlange durch zwei Umstände nach vorne:

- dadurch, dass das aufgerufene Entry des am Kopf der Schlange stehenden Prozesses die gewünschte Operation zu Ende führt, oder
- dadurch, dass für den Prozess am Kopf der Schlange ein `V.wait` ausgeführt wird, so dass er wieder in die Warteschlange einer Bedingung *V* zurückkehren muss.

Den etwas komplizierten Gesamtablauf des Monitors finden Sie graphisch dargestellt im Flussdiagramm 5.15 auf der nächsten Seite.

Wir präsentieren zur Ergänzung dieser Diskussion noch ein typisches Beispiel, das illustriert, warum Bedingungen in den Entries erforderlich sind und wie sie eingesetzt werden, um die Aufrufe an einen Monitor zu steuern.

Beispiel 5.11 (Briefkastenmonitor) Dieses Beispiel behandelt eine häufig vorkommende Situation, die sich das ***Producer-Consumer-Problem*** nennt. Dabei geht es um das Zusammenwirken von zwei Sorten von Prozessen: eine Sorte verbraucht oder „konsumiert“ einen (Daten-)Gegenstand, den

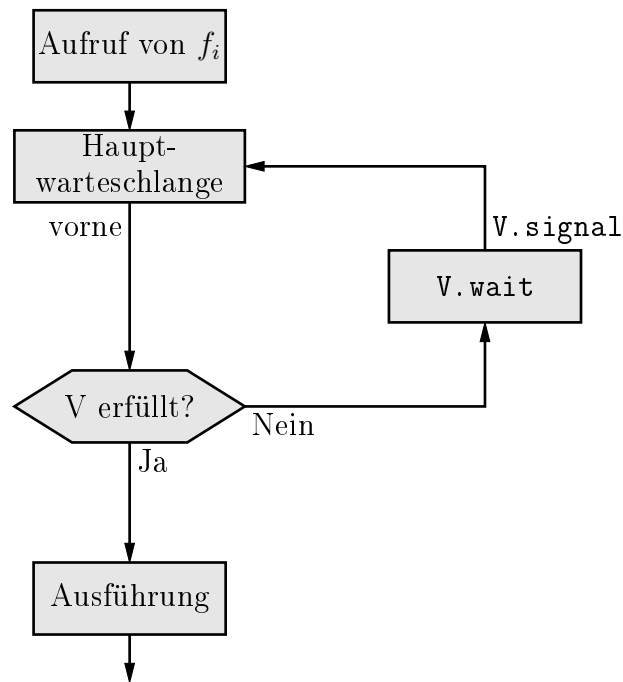


Abbildung 5.15: Flussdiagramm eines Monitoraufrufs

die andere Sorte erzeugt oder „produziert“. Die Verbraucher können aber nur konsumieren, wenn das Produkt schon hergestellt worden ist, und die Erzeuger können nur dann „Ware“ liefern, wenn in der begrenzten „Lagerhalle“ eines Computerspeichers Platz vorhanden ist, um sie aufzunehmen.

In unserem Beispiel wird die „Ware“ durch Briefe dargestellt, die von einigen Prozessen erzeugt und in einen Briefkasten abgelegt werden und von anderen Prozessen dem Briefkasten entnommen werden, vielleicht um sie weiterzuverarbeiten oder um sie auszudrucken oder was auch immer. Erschwert wird dieser Briefverkehr dadurch, dass der Briefkasten nur 10 Briefe fasst und mehr nicht aufnehmen kann.

Wir wollen die Zusammenarbeit der beteiligten Prozesse durch einen **Briefkastenmonitor** regeln lassen, der die Briefe annimmt und wieder ausgibt und verhindert, dass dabei der Briefkasten überfüllt wird (was in einem Rechner bedeuten würde, dass anderer Speicher überschrieben würde).

Der Monitor verwaltet nicht nur die Briefe selber, sondern auch zwei Bedingungen, die angeben, ob der Kasten voll ist und ob er leer ist (so dass es keine Briefe zum Abholen gibt), sowie als interne Variable die Anzahl der vorhandenen Briefe, woraus der jeweilige Zustand der Bedingungen festgestellt werden kann.

Hier ist das Monitorprogramm, in einer leicht zu verstehenden „Pseudo-

sprache“ ähnlich zu Pascal:

```
BKM:      monitor;

    var    kast: Platz für 10 Briefe;
           nonempty, nonfull: condition;
           Zahl: integer;                (* Anzahl der Briefe *)

    put:    entry();
    do      while Zahl=10 do nonfull.wait;
           Ablegen des Briefs;
           Zahl := Zahl + 1;
           nonempty.signal;
    end;

    get:    entry();
    do      while Zahl=0 do nonempty.wait;
           Holen des Briefs;
           Zahl := Zahl - 1;
           nonfull.signal;
    end;

    begin   Zahl := 0;                    (* anfangs leer *)
    end

end        BKM;
```

Der erste Block im Programm enthält die Variablendeklarationen. Vor dem Doppelpunkt in jeder Zeile stehen die Namen der erklärten Variablen und hinter dem Doppelpunkt ihr Datentyp.

Der letzte Block im Programm ist der Initialisierungsabschnitt, der ausgeführt wird, wenn der Monitor eingerichtet wird. Er setzt nur die Anzahl der am Anfang vorhandenen Briefe auf 0.

Der Monitor bietet zwei Entries `put()`, um einen Brief im Kasten abzugeben, und `get()`, um einen Brief aus dem Kasten abzuholen. Der Monitor selber sorgt dafür, dass zu jeder Zeit nur einer dieser Programmblöcke ausgeführt werden kann und führt (hier nicht sichtbar) eine Warteliste der Anwärter, die auf die Erstaussführung ihres Aufrufs oder auf die Fortsetzung ihres Aufrufs nach dem Wecken aus einer Schlafphase warten. Der Moni-

tor sorgt dafür, dass diese Instanzen zur passenden Zeit belebt werden und weiterlaufen dürfen.

Das Entry `put()` prüft als Erstes, ob der Briefkasten voll ist (daran zu erkennen, dass er schon 10 Briefe enthält). Wenn er voll ist, wird der Aufrufer mit `nonfull.wait` schlafen gelegt, bis wieder Platz für einen neuen Brief vorhanden ist.

Wenn der Kasten nicht voll ist, wird der angebotene Brief aufgenommen, der Zähler wird aktualisiert, und weil nach dem Ablegen eines Briefs der Kasten garantiert nicht leer sein kann, werden alle auf `nonempty` wartende Prozesse geweckt und in die Monitor-Warteliste gepackt.

Das Entry `get()` prüft als Erstes, ob der Briefkasten leer ist (daran zu erkennen, dass er 0 Briefe enthält), weil Briefe nur abgeholt werden können, wenn welche auch vorhanden sind. Wenn keine Briefe da sind, wird der Aufrufer mit `nonempty.wait` schlafen gelegt, bis ein anderer Prozess wie oben beschrieben einen Brief eingeworfen hat.

Wenn der Kasten Briefe enthält, wird ein Brief herausgegeben, der Zähler wird aktualisiert, und weil nach dem Holen eines Briefs der Kasten sicher nicht mehr voll sein kann, werden alle auf `nonfull` wartende Prozesse geweckt und in die Monitor-Warteliste gepackt.

Wir gehen davon aus, dass viele Prozess ständig versuchen, Briefe abzugeben oder zu holen. Dadurch kann es auch vorkommen, dass mehrere Abgeber auf einmal vor einem vollen Kasten warten, oder dass mehrere Abholer auf einmal auf die Abgabe eines Briefes warten, den sie abholen könnten. Es entstehen also tatsächlich Wartelisten für die relevanten Bedingungen.

Allerdings kann der Kasten nicht gleichzeitig leer und voll sein, so dass in jedem Moment höchstens einer der Vorbedingungen nicht erfüllt ist und deshalb höchstens eines der Entries gesperrt ist.

Durch die erfolgreiche Ausführung eines nichtgesperrten Entry geht auch die Vorbedingung des anderen Entry in Erfüllung, und die darauf wartenden Prozesse können alle geweckt werden. Sie können aber nicht gleichzeitig in Aktion treten (ein Brief, der in einen leeren Briefkasten eingeworfen wird, kann ja nur einmal von einem Prozess abgeholt werden). Das heißt, nur einer der wartenden Prozesse wird tatsächlich bedient (wenn nicht andere „ältere“ Prozesse vor ihm dran sind) und die anderen, wenn es andere gibt, kommen in die Hauptwarteschlange für den Monitor.

Wenn sie später an der Reihe sind, wird das aufgerufene Entry wieder abgearbeitet ab der Stelle, wo es beim Schlafengehen unterbrochen wurde. Inzwischen kann der Kasten schon mehrmals aufgefüllt und wieder leer gemacht worden sein, so dass die Vorbedingung, auf die die Prozesse gewartet haben, längst nicht mehr den Zustand haben muss, der galt, als die Prozesse in die Hauptwarteliste kamen. Deshalb muss nach dem Wecken aus der

Hauptwarteschlange die Vorbedingung wieder getestet werden. Dafür sorgt automatisch die **while** Abfrage im Programmcode für die Entries; bei der Wiederaufnahme der Verarbeitung wird die Schleife erst verlassen, wenn die abgefragte (verhindernde) Bedingung nicht mehr gilt.

Um den Verarbeitungsablauf im Monitor noch einmal zu illustrieren, spielen wir eine typische Situation mit dem Briefkastenmonitor durch.

Angenommen, dass sofort nach Initialisierung des Monitors zwei Prozesse A und B das Entry `get()` aufrufen, um einen Brief abzuholen, und dass anschließend ein dritter Prozess C das Entry `put()` aufruft, um einen Brief in den Kasten einzuwerfen.

Der Ablauf, den man gut im Programmtext und im Flussdiagramm 5.15 verfolgen kann, wäre folgender:

Prozess A und hinter ihm Prozess B kommen in die Monitor-Warteschlange, und der Monitor holt Prozess A aus der Warteschlange und führt für diesen Prozess die Anweisungen in `get()` aus.

Weil die interne Variable `Zahl` noch auf 0 steht, wird die **while**-Schleife einmal durchlaufen und `nonempty.wait` wird ausgeführt. Prozess A steht jetzt vorne in der Warteschlange für Bedingung `V = nonempty`, und der Platz in der Monitor-Warteschlange ist geräumt. Der Monitor ist jetzt frei, andere Prozesse zu bedienen.

Inzwischen könnte der Aufruf von Entry `put()` aus Prozess C eingegangen sein. Prozess C kommt in die Monitorwarteschlange, aber steht hinter Prozess B! Deshalb wird der Monitor sich als nächstes mit dem Aufruf von Prozess B befassen (auch wenn wir aus unserer Warte sehen können, dass es besser wäre, C zuerst zu bedienen).

Für Prozess B wird wieder Entry `get()` gestartet. Weil `Zahl` immer noch auf 0 steht, wird die **while**-Schleife durchlaufen und `nonempty.wait` wird ausgeführt. Prozess B kommt hinter Prozess A in die Warteschlange für Bedingung `nonempty`, legt sich schlafen, und sein Platz in der Monitor-Warteschlange ist geräumt.

Jetzt ist nur noch Prozess C in der Monitor-Warteschlange und wird bedient, da er als einziger Eintrag am Kopf der Schlange steht. Für Prozess C wird Entry `put()` gestartet. Weil `Zahl` nicht 10 ist, wird die **while**-Schleife übersprungen, der Brief von Prozess C wird im Briefkasten abgelegt, `Zahl` wird auf 1 erhöht, und `nonempty.signal` wird ausgeführt, wodurch Prozesse A und B wieder in die Hauptwarteschlange kommen, A vor B.

Der Monitor ist jetzt mit dem Entry-Aufruf fertig und kann den nächsten verarbeiten. Prozess A am Kopf der Schlange ist jetzt an der Reihe, und Entry `get()` wird wieder aufgerufen am Ende der **while**-Schleife, wo die Bearbeitung beim ersten Durchlauf abgebrochen wurde.

Die **while**-Bedingung wird wieder untersucht, aber ist jetzt falsch, da

Zahl = 1. Also wird die Schleife übersprungen, der nun vorhandene Brief wird an Prozess A „ausgeliefert“, **Zahl** wird um 1 vermindert und steht wieder auf 0, und **nonfull.signal** wird ausgeführt, jetzt ohne Wirkung, da keine Prozesse auf **nonfull** warten.

Der Monitor ist wieder frei und holt den nächsten Aufruf aus seiner Warteschlange, den Aufruf von Prozess B, der als einziger in der Warteschlange steht.

Für Prozess B wird wieder Entry **get()** gestartet, und zwar am Ende der **while**-Schleife, wo der erste Durchlauf abgebrochen wurde. Weil **Zahl** jetzt aber *wieder* auf 0 steht, wird die **while**-Schleife wieder durchlaufen und **nonempty.wait** wird ein zweites Mal für Prozess B ausgeführt. Der Prozess kommt wieder in die Warteschlange für Bedingung **nonempty** und legt sich schlafen.

Da die Monitor-Warteschlange jetzt leer ist, legt auch der Monitor sich schlafen, bis ein neuer Aufruf eines seiner Entries kommt.

Aufrufe an Entry **get()** kommen hinter B in die Warteschlange für Bedingung **nonempty**, und nur ein Aufruf an Entry **put()** kann sofort verarbeitet werden.

B kann erst bedient werden, nachdem ein solcher Aufruf an Entry **put()** verarbeitet wurde und dabei **nonempty.signal** ausgeführt hat, um B wieder in die Hauptwarteschlange zu bringen. Wichtig zu bemerken ist, dass Prozess B in der Zwischenzeit in der **nonempty**-Warteschlange gut aufgehoben ist, denn dort blockiert er nicht die Hauptwarteschlange für andere „glücklichere“ Prozesse.

Zum Abschluss des Kapitels wollen wir kurz beschreiben, wie einige der Konzepte zur Regelung paralleler Verarbeitung in bekannten Programmiersprachen implementiert werden können.

In **Java** können sowohl Methoden (also „Unterprogramme“) wie auch Objekte (oder genauer, Anweisungsblöcke *in Bezug auf* ein Objekt) das Attribut **synchronized** tragen.

Synchronized Methoden haben den ausschließlichen Zugriff auf alle Objekte, auf die sie zugreifen (mit der Ausnahme, dass eine **synchronized** Methode andere **synchronized** Methoden aufrufen kann, die auf die gleichen Objekte synchronisiert sind, ohne dass diese „Helfer“ dann blockieren; die aufgerufenen Methoden haben Zugriff auf Objekte, für die der Aufrufer die ausschließliche Benutzung hat).

Mit **synchronized** Methoden kann man zum Beispiel Monitore realisieren und die internen Objekte des Monitors schützen.

Eine **synchronized** Anweisung

```
synchronized (Objekt) {
    Anweisungen
}
```

hingegen bezieht sich auf ein einzelnes Objekt und benutzt dieses Objekt wie ein Semaphore. Wenn die **synchronized** Anweisung ausgeführt wird, wird ein **Lock** auf das genannte Objekt eingerichtet und der zwischen den geschweiften Klammern stehende Anweisungsblock wird ausgeführt mit ausschließlichem Zugriff auf das gesperrte Objekt.

Die in Monitoren implementierte Behandlung von Vorbedingungen kann auch sehr einfach in Java implementiert werden. Java bietet genau das dazu Erforderliche: eine Methode **wait**, die den gegenwärtigen Ausführungsfaden (also den laufenden Prozess) unterbricht und schlafen legt, und Methoden **notify** und **notifyAll**, die einen speziellen oder *alle* wartenden Prozesse weckt.

Das Testen von Bedingungen, die Weiterbearbeitung im günstigen Fall, und die **wait** Methode sollten (wie bei Monitor Entries) in einem synchronisierten Block liegen, aber die Ausführung von **wait()** hebt die Sperrung wieder auf, so dass die geschützten Objekte dann von anderen Prozessen benutzt werden können (sonst könnte eine ungünstige Bedingung sich nie wieder ändern!).

Anders als wir es für Monitore beschrieben haben, weckt **notifyAll** alle wartenden Prozesse und nicht nur diejenigen, die auf eine bestimmte Bedingung warten. Aber wenn die **wait**-Anweisungen wie in der Monitorbeschreibung gefordert innerhalb von testenden **while**-Schleifen stehen, werden die nicht gemeinten Schläfer sofort feststellen, dass ihre Bedingung noch nicht gilt und ihre Zeit noch nicht gekommen ist, und sich wieder schlafen legen. Das hat im Endeffekt dann doch die gewünschte Wirkung.

Ada erlaubt die synchronisierte Kommunikation und den geregelten Datenaustausch zwischen Prozessen, sowie den sicheren und vor Wechselwirkungen geschützten Aufruf von auszuführenden Operationen wie in einem Monitor.

Dazu gibt es das Schlüsselwort **accept**. Wie im Monitor wird ein Entry *a* erklärt, der den Zugang zu einer bestimmten Operation bietet. Die Ausführung dieser Operation wird von einem bestimmten Prozess angeboten, der mit einer Anweisung der Gestalt

```
accept a do < Anweisungen > end
```

seine Bereitschaft verkundet, *jetzt* Aufrufe des Entry anzunehmen, wobei dann die angegebenen Anweisungen ausgeführt werden.

Dieses Konzept ist auch dazu geeignet, Warteschlangen wie in Monitoren zu implementieren. Die **accept** Anweisung kann nur ausgeführt werden, wenn Aufrufe des Entry *a* vorliegen, und sonst wartet sie auf den nächsten solchen Aufruf.

Aufrufe des Entry *a* können nur verarbeitet werden, wenn ein anderer Prozess eine **accept** Anweisung für den Entry erreicht hat und zur Annahme von solchen Aufrufen bereit ist. Andernfalls werden die Aufrufe von *a* in einer Warteschlange gesammelt, bis eine solche **accept** Anweisung ausgeführt wird.

In anderen Worten, Aufrufe des Entry und der ausführende Prozess warten aufeinander und führen auf diese Weise ein *rendezvous* aus, bis beide Partner „eintreffen“. Diese Einrichtung sorgt auch für einen geregelten Datenaustausch zwischen beiden Partnern. Das System, also die Ada Implementierung, regelt von sich aus den Verkehr durch Warteschlangen.

Es gibt noch eine Variante der **accept** Anweisung, die **select-accept-delay** Anweisung, die die Einrichtung verschiedener Ausführungsvarianten ermöglicht, die auch von weiteren abzuwartenden Ereignissen abhängen können.

Wenn bei dieser Anweisung der durch **accept** abgewartete Aufruf innerhalb der im **delay**-Teil angegebenen Zeitspanne erfolgt, wird eine Variante der Operation ausgeführt, und sonst die angegebene Alternative.

Hier ein Beispiel:

```
accept klick do
select accept klick do
    erkenne ( Doppelklick );
end
or    delay 0.2 ;
    erkenne ( Einfachklick );
end  select ;
end ;
```

Ada ist eine Programmiersprache, die vom amerikanischen Verteidigungsministerium beauftragt wurde, um sehr sichere Programme für Echtzeitsteuerungsaufgaben zu ermöglichen, und sie wird auch gerne für nichtmilitärische Zwecke verwendet, wo höchste Sicherheit verlangt wird (zum Beispiel für die Steuerung von medizinischen Geräten oder von Flugzeugen).

Trotzdem bietet Ada keine absolute Garantie, dass Programmierfehler nicht stattfinden können, obwohl sie viele Hilfen bietet, häufige Gefahrenquellen zu vermeiden. Beim ersten Flug der Ariane 5 Rakete im Jahr 1996 hat ein Ada Programmierfehler (oder eher ein „Bedienungsfehler“, nämlich

das Abschalten einer in Ada implementierten Sicherheitsprüfung) den Absturz der Rakete verursacht.

Kapitel 6

Formale Sprachen und Grammatiken

Es hat früher unter Linguisten eine Kontroverse darüber gegeben (und es gibt sie heute noch), ob menschliche Sprache kulturell erworben wird oder angeboren ist.

Die erste Ansicht behauptet, dass wir in der Lage sind, auf deutsch oder englisch oder jeder anderen Sprache miteinander zu kommunizieren, weil unsere Eltern und die Menschen in unserer Umgebung diese Sprachen benutzen und weil sie sie uns auf die gleiche Weise beigebracht haben, wie sie uns beigebracht haben, mit Messer und Gabel zu essen oder die Toilette zu benutzen. In dieser Ansicht ist die Fähigkeit, eine menschliche Sprache zu erlernen, nicht anders als die allgemeine durch die menschliche Intelligenz begünstigte Fähigkeit, komplizierte Tätigkeiten wie das Lesen und das Schreiben, das Klavierspielen oder das Autofahren mit Übung zu erlernen.

Die zweite Ansicht behauptet, dass wir eine Sprache sprechen, weil die Evolution und der Überlebensnutzen einer differenzierten Kommunikation uns mit einer in unseren Genen verankerte *prinzipielle* und spezielle Begabung dafür ausgestattet haben, die nicht nur die anatomischen Voraussetzungen für die Lautbildung betrifft sondern auch die Strukturen der Sprache selber.

Gegen diese Ansicht scheint die offensichtliche Tatsache zu sprechen, dass nicht alle Menschen die gleiche Sprache sprechen und dass die Einzelheiten der lebenden Sprachen einem ständigen Wandel unterworfen sind. Diese Einzelheiten *müssen* wir von den Menschen um uns lernen, und sie können uns nur durch das Vormachen durch andere Leute vermittelt werden.

Das würde heißen, dass die deutsche oder die japanische Sprache genau so wenig angeboren ist, wie uns das lateinische Alphabet oder die Kanji Schriftzeichen angeboren sind.

Aber natürlich behaupten die Verfechter der „angeborenen“ Schule gar nicht, dass uns der deutsche Wortschatz oder die japanische Grammatik angeboren ist, sondern nur, dass alle Sprachen eine sehr tief liegende gemeinsame Struktur haben, an die die Einzelheiten der verschiedenen Sprachen sich aufhängen lassen, und *dass alle Babys von Geburt an Kenntnis von dieser Struktur haben* und deshalb die *Basis* der Sprache *nicht* erlernen müssen.

Vieles spricht dafür. Nicht alle Menschen lernen Klavierspielen oder Schreiben oder Autofahren, und viele Menschen, auch wenn sie diese Tätigkeiten lernen wollen, werden nie gute Klavierspieler, perfekte Orthographen oder sichere Autofahrer, aber *alle* gesunden Menschen ohne Gehirnschäden lernen im frühen Kindesalter eine menschliche Sprache und verwenden sie virtuos und nach den Regeln ihres Kulturkreises grammatisch richtig. Die Fähigkeit zu diesem Erlernen schwindet aber schon im jungen Alter, so dass Erwachsene es sehr schwer haben, zusätzliche Fremdsprachen noch zu lernen.

Im Gegensatz dazu erfordert das Erlernen der meisten anderen Fähigkeiten eine gewisse Reife oder die Ausbildung von feinmechanischem Können, um erfolgreich zu sein.

Die verschiedenen etwa 6000 Sprachen, die es gibt, variieren sehr stark in den Einzelheiten ihrer Struktur und in ihren grammatischen Merkmalen. Es gibt Sprachen mit bestimmten und unbestimmten Artikeln aber auch Sprachen, die nur eine der beiden Sorten haben oder auf beide verzichten. Es gibt viele verschiedene Regeln über die Reihenfolge der Wörter in einem Satz und viele verschiedene Konstrukte, um bestimmte Zusammenhänge zwischen Satzteilen auszudrücken. Zum Beispiel, wo ein Deutscher von „dem Buch, das ich gestern gelesen habe“ spricht, muss ein Türke aus seinem gestrigen Lesen ein Hauptwort machen, daran ein Possesivsuffix anhängen, um es mit ihm selber in Verbindung zu bringen, und das ganze dann als Adjektiv vor das „Buch“ setzen.

Aber es gibt auch viele Merkmale, die *allen* menschlichen Sprachen gemeinsam sind, zum Beispiel die Verwendung von Verben oder sogar seltsame Besonderheiten, wie die Tatsache, dass alle Menschen, wenn von einem roten Haus gesprochen wird, diese Aussage so auffassen, dass das Haus von *außen* und nicht von innen rot gefärbt ist, unabhängig von der Sprache, die sie sprechen.

Gerade die Fähigkeit, die alle Babys haben, aus dem Kauderwelsch, das sie um sich hören, eine äußerst komplizierte Struktur abzulesen, die sie bald selber fehlerfrei reproduzieren können, sogar in Varianten, die sie nie selber gehört haben, ist ein sehr starkes Indiz für die Richtigkeit der Theorie der angeborenen Sprachfähigkeit. Es ist kaum zu glauben, dass ein völlig unerfahrener Mensch, egal wie intelligent er ist, sonst zu dieser erstaunlichen Dechiffriertat in der Lage wäre. Und Intelligenz ist keine Voraussetzung für

die Spracherwerbung im Kindesalter.

Einer der frühen Verfechter dieser Theorie der angeborenen Sprachfähigkeit war der Linguist Noam Chomsky, der sich in den 50er Jahren des vergangenen Jahrhunderts der von dieser Theorie aufgeworfenen Frage widmete, wie diese universelle Grammatik, die wir alle von Geburt an kennen müssen, beschaffen sein mag.

Chomskys Antwort auf diese Frage wurde in dem 1957 erschienenen Buch *Syntactic Structures* vorgestellt, wo er eine zweistufige strukturelle Hierarchie vorschlägt.

Die Grundlage bildet eine **phrase structure grammar** in seiner Terminologie, mit einfachen Erzeugungsregeln, die den Satzaufbau widerspiegeln und zurückverfolgbar machen. Die Grundlage dieser erzeugenden Grammatik bilden Ersetzungsregeln der Gestalt

$$u \rightarrow v,$$

wo u und v Namen oder Ausdrücke sind und die Regel so zu verstehen ist, dass in einem eventuell größeren Ausdruck ein Vorkommen von u (die linke Seite der Regel) durch v (die rechte Seite der Regel) ersetzt werden kann.

Wir präsentieren als Beispiel eine einfache englische Grammatik von der genannten Art, direkt zitiert aus *Syntactic Structures*:

- i) $Sentence \rightarrow NP + VP$
- ii) $NP \rightarrow T + N$
- iii) $VP \rightarrow Verb + NP$
- iv) $T \rightarrow the$
- v) $N \rightarrow man, ball \text{ usw.}$
- vi) $Verb \rightarrow hit, took \text{ usw.}$

Hier steht NP für „noun phrase“, VP für „verb phrase“, N für „noun“ oder Hauptwort, und alles andere ist klar.

Eine typische Herleitung eines englischen Satzes aus diesen Regeln wäre zum Beispiel

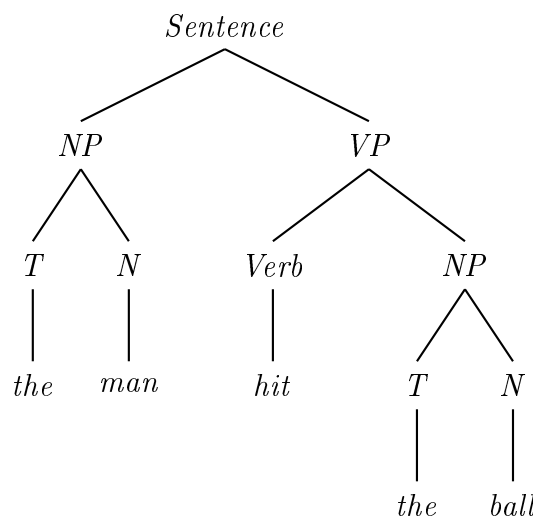
Sentence

- | | |
|-------------------------|------|
| $NP + VP$ | i) |
| $T + N + VP$ | ii) |
| $T + N + Verb + NP$ | iii) |
| $the + N + Verb + NP$ | iv) |
| $the + man + Verb + NP$ | v) |

<i>the + man + hit + NP</i>	vi)
<i>the + man + hit + T + N</i>	ii)
<i>the + man + hit + the + N</i>	iv)
<i>the + man + hit + the + ball</i>	v)

wo die römischen Ziffern am rechten Rand angeben, welche Regel angewendet wurde, um die Zeile aus der vorhergehenden zu erzeugen.

Die Herleitung lässt sich auch durch einen Entwicklungsbaum darstellen:



Chomsky hat erkannt, dass viele Konstruktionen der natürlichen Sprache sich nicht gut durch eine solche erzeugende Grammatik beschreiben lassen, and dass viele Phänomene des richtigen Satzbaues und des richtigen Verständnisses einiger Konstruktionen sich nur erklären lassen, wenn man davon ausgeht, dass menschliche Zuhörer sich nicht alleine nach der Reihenfolge der Wörter im Satz richten, sondern auch nach dem grammatischen Zweck des Wortes. Dazu hat er Satzpaare angegeben mit der gleichen Reihenfolge von Worttypen, die aber auf ganz verschiedene Arten verstanden werden müssen (ein typisches Beispiel, eingedeutscht: „das Brüllen von Löwen“ in Vergleich zu „das Züchten von Blumen“, oder zweideutig: „das Schlagen von Kindern“ — schlagen die Kinder, oder werden sie geschlagen?).

Diese Sätze würden von einer reinen erzeugenden Grammatik auf die gleiche Weise erzeugt, aber sie haben offensichtlich nicht die gleiche Struktur. Um das zu erklären fügt Chomsky eine zweite, Transformationsstufe in seine hierarchische Grammatik ein.

Diese Stufe beinhaltet Transformations- oder Umwandlungsregeln, die auf die Ergebnisse der erzeugenden Stufe und die eben gezeichnete Baumstruktur

wirken und dort die Äste vertauschen und markierende Zusätze in die Wortfolge einfügen. Solche Transformationen wandeln normale Sätze in der Standardwortreihenfolge in Passivsätze, verändern die Zeiten der Verben mit den damit verbundenen Formänderungen (so wird zum Beispiel „Ich ging gestern in die Mensa“ in „Ich bin gestern in die Mensa gegangen“ verwandelt, wobei die Wortreihenfolge und die Schreibweise des Verbs „gehen“ sich ändern) und fügen Hauptsätze und Nebensätze zu einem mehrteiligen Satz zusammen.

Die oben genannten ähnlichen Ausdrücke mit verschiedenen Bedeutungen sind verschieden, weil sie durch verschiedene Transformationen erzeugt wurden, auch wenn sie dem Schein nach den gleichen Aufbau haben.

Zur Entwicklungszeit von Chomskys Theorie gab es schon neben den menschlichen Sprachen auch **formale Sprachen** für wissenschaftliche und technische Zwecke, darunter die Formelsprache der mathematischen Logik und die ersten höheren Computersprachen (FORTRAN). Aber Chomsky war an diesen künstlichen Sprachen nicht interessiert; er war alleine von dem Bedürfnis inspiriert, die natürliche menschliche Sprache zu erleuchten und durch eine geeignete Grammatik eine Erklärung für beobachtete Sprachphänomene zu geben, die neben der Beschreibung von Beobachtungen auch Vorhersagen über Sprachverhalten liefern sollte, anhand derer man das Modell testen und verfeinern könnte. Nebenbei würde diese Grammatik auch helfen, die Theorie der angeborenen Sprachfähigkeit zu untermauern.

Aber es stellte sich schnell heraus, dass die sich entwickelnden grammatischen Ideen auch hervorragend für die Beschreibung formaler Sprachen und damit auch für die Beschreibung von Computersprachen eigneten. Für diese Sprachen wird der Transformationsteil nicht gebraucht, aber die erzeugende Grammatik hat sich zu einem sehr wichtigen Werkzeug in der Informatik entwickelt, einmal, weil es eine präzise und eindeutige Beschreibung der Syntax einer Computersprache erlaubt, mit der sich zum Beispiel Sprachnormen genau definieren lassen, und weiter, weil eine formale Grammatik die Grundlage für theoretische Überlegungen bietet und eine Hilfe ist bei Untersuchungen über Berechenbarkeit und über die Sicherheit und Komplexität von Berechnungen.

Aus diesem Grund wollen wir in diesem Kapitel einige der wichtigen Begriffe präsentieren, die in Verbindung mit formalen Sprachen und ihren Grammatiken verwendet werden.

Um eine formale Sprache zu „schreiben“, brauchen wir Schriftzeichen. Wir stellen uns deshalb einen endlichen Satz irgendwie gearteter Zeichen vor, aus denen wir Worte zusammensetzen werden.

Definition 6.1 Ein **Alphabet** ist eine endliche nichtleere Menge Σ , deren Elemente wir **Buchstaben** oder **Zeichen** nennen werden.

Die Elemente eines Alphabets können im Sinne der Definition beliebige Objekte sein, aber wenn wir darüber sprechen, werden wir sie natürlich tatsächlich durch Schriftzeichen darstellen.

Ein **Wort** über das Alphabet Σ ist eine endliche Folge

$$x_1 x_2 \dots x_n \quad (6.1)$$

von Zeichen $x_i \in \Sigma$ (die nicht notwendigerweise verschieden sein müssen).

Die Anzahl n der Zeichen in einem Wort w (wobei mehrmals erscheinende Zeichen so oft gezählt werden, wie sie erscheinen) heißt die **Länge** des Worts und wird mit $L(w)$ bezeichnet.

Auch die *leere* Zeichenfolge, also der Fall $n = 0$, ist möglich. Da man ein Wort aus „keine Zeichen“ nicht sehen kann und deshalb nicht durch eine lesbare Zeichenfolge benennen kann, brauchen wir für das leere Wort einen anderen Standardnamen. Wir benutzen für das leere Wort den Namen ε .

Wichtig zu verstehen ist: das leere Wort ist *wirklich* ein Wort und kommt in dem Wortschatz vor, auch wenn es unsichtbar ist. Sein Standardname ε ist *kein* Zeichen unseres Alphabets, sondern erfüllt nur die Sonderfunktion, der Name des leeren Worts zu sein, um es möglich zu machen, auf dieses sonst unsichtbare Wort Bezug zu nehmen.

Für jedes $n \in \mathbf{N}$ bezeichnen wir mit Σ^n die Menge aller Worte von Länge n über das Alphabet Σ , und mit

$$\Sigma^* = \bigcup_{n \in \mathbf{N}} \Sigma^n \quad (6.2)$$

die Menge aller Worte überhaupt über das Alphabet Σ .

Beachten Sie, dass

$$\Sigma^0 = \{ \varepsilon \}$$

ein einziges Element enthält, nämlich das leere Wort, und deshalb *nicht* die leere Menge ist!

Für Worte verwenden wir folgende Operationen, Begriffe und Abkürzungen.

Definition 6.2 a) Sei Σ ein Alphabet und seien

$$u := x_1 x_2 \dots x_n \quad \text{und} \quad v := y_1 y_2 \dots y_m$$

Worte aus Σ^* .

Wir führen eine Operation \cdot ein, die wir die **Verkettung** von u und v nennen, und die darin besteht, dass wir die Zeichenfolgen von u und

v direkt hintereinander ohne Trennstelle aneinanderfügen, so dass ein neues Wort $u \cdot v$ der Länge $m + n$ entsteht. In einer Formel:

$$u \cdot v := x_1 x_2 \dots x_n y_1 y_2 \dots y_m \quad (6.3)$$

Nachdem die Verkettung gebildet wurde ist nicht mehr zu erkennen, wo vorher u aufgehört und v begonnen hatte, genau so wenig, wie sie bei einer Summe $2 + 5 = 7$ aus dem Ergebnis 7 erkennen können, dass es aus den genannten Summanden entstanden ist und nicht etwa aus $3 + 4$ oder $1 + 6$.

Insbesondere ist in dem Ergebnis einer Verkettung der Punkt \cdot nicht mehr zu sehen!

Das nutzen wir in der Notation auch aus. Wo immer keine Missverständnisse entstehen können, lassen wir den Punkt in der Notation weg. Zeichen a und b müssen wir ohnehin ohne Punkt zu einem Wort ab zusammenfügen, aber auch für Worte u und v der Länge größer als 1 erlauben wir uns, für die Verkettung einfach uv zu schreiben, mit der Bedeutung wie in Formel (6.3).

Die Verkettungsoperation hat folgende Eigenschaften:

- i) Für jedes Wort u ist $\varepsilon \cdot u = u = u \cdot \varepsilon$.
 - ii) Verkettung ist assoziativ, d. h., wenn u , v und w Worte sind, dann ist

$$(u \cdot v) \cdot w = u \cdot (v \cdot w)$$
 (und wir schreiben dafür deshalb einfach $u \cdot v \cdot w$).
 - iii) Die Verkettung ist *nicht* kommutativ, d. h., $u \cdot v$ ist im allgemeinen nicht das Gleiche, wie $v \cdot u$!
- b) Ein Wort $t \in \Sigma^*$ heißt **Teilwort** eines Wortes w , wenn es Worte u und v (eventuell leer) gibt, so dass

$$w = utv. \quad (6.4)$$

Wenn dabei $u = \varepsilon$, nennen wir t einen **Anfangsteil** von w .

Wenn dabei $v = \varepsilon$, nennen wir t einen **Endteil** von w .

- c) In Beispielen kommt es oft vor, dass ein Wort lange Folgen des gleichen Buchstabens enthält, zum Beispiel $aaaaabbabbbbcccc$. Auch um die Lesbarkeit zu erhöhen verwenden wir dafür als Abkürzung die gewohnte Potenznotation, d. h., wir würden das gerade genannte Wort

als $a^5b^2ab^5c^3$ schreiben. Die gleiche bequeme Notation kann man auch für Teilworte benutzen.

Die allgemeine (und übliche) Definition der ***n-ten Potenz*** eines Wortes w kann wie folgt mittels Rekursion gegeben werden. Wir setzen

$$\begin{aligned} w^0 &:= \varepsilon \\ w^n &:= w \cdot w^{n-1} \quad \text{wenn } n > 0. \end{aligned}$$

Wie man es nicht anders erwarten würde ist w^n die Verkettung von n Kopien von w .

Für „linguistische“ Anwendungen sind negative Potenzen nicht definiert, d. h., n muss eine natürliche Zahl sein.

d) Sei

$$w := x_1x_2 \dots x_n$$

ein Wort der Länge n . Die Reihenfolge der Buchstaben im Wort spielt eine Rolle; insbesondere, wenn wir die Buchstaben von w in die umgekehrte Reihenfolge schreiben, erhalten wir in der Regel ein von w verschiedenes Wort

$$\tilde{w} = x_nx_{n-1} \dots x_1,$$

genannt das ***Spiegelbild*** von w .

Das Spiegelbild kann man auch wie folgt durch Rekursion definieren:

$$\tilde{w} := \begin{cases} \varepsilon, & \text{falls } w = \varepsilon; \\ \tilde{u}a, & \text{falls } w = au \text{ mit } a \in \Sigma. \end{cases}$$

Das Spiegelbild hat folgende Eigenschaften:

- i) $\tilde{\varepsilon} = \varepsilon$.
- ii) Die Bildung des Spiegelbildes ist eine Involution, d. h.,

$$\widetilde{\tilde{w}} = w.$$

- iii) Für alle Worte u und v gilt

$$\widetilde{uv} = \tilde{v}\tilde{u}.$$

Es ist sehr nützlich in der Theorie der formalen Sprachen auch die ***Verkettung von Mengen von Worten*** als eine bequeme Notation einzuführen. Sie bezeichnet einfach die Menge aller möglichen Verkettungen von Elementen der beteiligten Mengen, genauer:

Definition 6.3 Sei Σ ein Alphabet und seien $M \subseteq \Sigma^*$ und $N \subseteq \Sigma^*$ Mengen von Worten über Σ . Wir definieren

$$M \cdot N := \{ uv \mid u \in M, v \in N \}. \quad (6.5)$$

Manchmal schreiben wir dafür auch einfach MN .

Mit dieser Definition können wir dann analog zu den Definitionen für das Alphabet oder für einzelne Buchstaben oder Worte **Potenzen** von Wortmengen bilden (die aber Potenzen bezüglich der Verkettung sind, nicht bezüglich des kartesischen Produkts!) und sogar das **Stern** einer Menge von Worten:

Wir definieren

$$M^n := \begin{cases} \{ \varepsilon \}, & \text{wenn } n = 0; \\ M \cdot M^{n-1}, & \text{wenn } n > 0 \end{cases} \quad (6.6)$$

und daraus abgeleitet

$$M^* := \bigcup_{n \in \mathbf{N}} M^n. \quad (6.7)$$

Man beachte, dass für $M = \Sigma$ diese Definition im Einklang steht mit unserer früherer Definition von Σ^n als die Menge aller Worte der Länge n und von Σ^* als die Menge aller Worte über Σ überhaupt.

Beispiel 6.4 a) Sei $M = \{ a, ab \}$ und $N = \{ a, ba \}$. Dann ist

$$\begin{aligned} M \cdot N = MN &= \{ a^2, aba, ab^2a \} \\ M^3 &= \{ a^3, a^3b, a^2ba, aba^2, a^2bab, aba^2b, ababa, ababab \} \end{aligned}$$

und M^* besteht aus allen Worten in a und b , in denen direkt vor jedem b ein a steht.

b) Wenn $E = \emptyset$, dann ist $E \cdot F = \emptyset$ für jede Wortmenge F . Daraus folgt

$$\begin{aligned} E^0 &= \{ \varepsilon \} \\ E^n &= \emptyset \quad \text{für alle } n > 0 \\ E^* &= \{ \varepsilon \} \end{aligned}$$

Definition 6.5 Eine **Sprache** über ein Alphabet Σ ist eine beliebige Wortmenge über dieses Alphabet, also eine beliebige Teilmenge

$$L \subseteq \Sigma^*.$$

Sprachen kann man aus theoretischen Gesichtspunkten betrachten, in gewissem Sinne als ein interessantes kombinatorisches Problem der Mathematik, oder aus praktischen Gründen, um natürliche Sprachen zu beschreiben oder um die Struktur von **Programmiersprachen** zu erleuchten oder eindeutig festzulegen (zum Beispiel in den Vorschriften einer Industrienorm).

Außerdem haben wir gesehen, dass Automaten und ihre Zustandsgraphen und Petri-Netze und ihre Schaltfolgen Sprachen bestimmen, und Sprachen treten auch in der formalen Beschreibung von Berechnungen, Rechenkomplexität und Berechenbarkeit auf, so dass die Theorie der formalen Sprachen Verbindungen zu vielen wichtigen Strukturen der Informatik hat und deshalb selber zu einem wichtigen Teil der theoretischen Informatik geworden ist.

Um „theoretisch“ über Sprachen sprechen zu können, brauchen wir für sie etwas mehr Struktur als nur die einer beliebigen Menge von Worten. Diese Struktur können die Sprachen verliehen bekommen durch *Mechanismen zu ihrer Erzeugung*, die wir zuerst ganz allgemein und später in immer genaueren Versionen betrachten wollen.

Die gebräuchlichste Methode, um Sprachen zu erzeugen und zu beschreiben, besteht in der Angabe von Regeln für die Produktion von Aussagen oder Worten der Sprache; diese Regeln machen die **Grammatik** der Sprache aus. Dies halten wir in folgender allgemeiner Definition fest.

Definition 6.6 Ein **Produktionssystem** ist ein Paar

$$(\Gamma, R),$$

wo Γ ein Alphabet ist und R ist eine endliche Menge von **Regeln** der Gestalt

$$u \rightarrow v,$$

in denen u und v Worte über Γ sind. Diese Regeln sind wie folgt zu verstehen oder anzuwenden.

Seien α und $\beta \in \Gamma^*$. Wir sagen, α **ist direkt ableitbar zu β über den Regelsatz R** genau dann, wenn es Worte x und $y \in \Gamma^*$ und eine Regel $u \rightarrow v$ in R gibt, so dass

$$\alpha = xuy \quad \text{und} \quad \beta = xvy.$$

Wir schreiben dafür $\alpha \xrightarrow{R} \beta$ oder manchmal sogar einfach $\alpha \rightarrow \beta$.

(In anderen Worten, eine Anwendung der Regel $u \rightarrow v$ ersetzt in einem Wort ein Vorkommen von u als Teilwort durch v an der gleichen Stelle.)

Wir sagen, α **ist ableitbar zu β über den Regelsatz R** , und schreiben

$$\alpha \xrightarrow{*} \beta,$$

wenn eine (eventuell leere) Kette von direkten Ableitungen von α nach β führt, in anderen Worten, wenn $\alpha = \beta$ oder wenn es Zwischenstufen $\alpha = \alpha_0$, α_1 , α_2 , \dots , α_{n-1} , $\alpha_n = \beta$ gibt und eine Folge von direkten Ableitungen

$$\alpha_0 \rightarrow \alpha_1, \quad \alpha_1 \rightarrow \alpha_2, \quad \dots, \quad \alpha_{n-1} \rightarrow \alpha_n.$$

Die Folge $\alpha_0, \alpha_1, \alpha_2, \dots, \alpha_n$ der Zwischenstufen nennt sich eine **Ableitung von β aus α der Länge n** .

Wichtig zu betonen ist, dass die Ableitungsregeln immer als „kann“-Regeln und nie als „muss“-Regeln zu verstehen sind. Sie beschreiben *erlaubte* Ersetzungen und nicht zwingende Ersetzungen, und bei einem gegebenen Wort kann es durchaus vorkommen, dass mehrere Regelanwendungen möglich sind, wovon nur eine tatsächlich ausgeführt wird.

Dabei kann es einerseits sein, dass die linke Seite einer Regel $u \rightarrow v$ an mehreren, vielleicht sogar sich überlappenden Stellen in einem Wort α vorkommt, so dass *eine* Regel auf viele verschiedene Weisen angewendet werden kann, oder es ist möglich, dass verschiedene Regeln auf ein bestimmtes Teilwort oder auf mehrere verschiedene Teilworte von α anwendbar sind.

Es ist auch durchaus zulässig und kommt häufig vor, dass es zu einer bestimmten linken Seite u mehrere Regeln $u \rightarrow v_1$, $u \rightarrow v_2$, \dots , $u \rightarrow v_n$ mit verschiedenen rechten Seiten v_i gibt. Diese Situation kürzt man gerne durch eine Art „oder“-Notation ab. Anstelle der n gerade genannten Regeln schreibt man eine einzige Regel

$$u \rightarrow v_1 \mid v_2 \mid \dots \mid v_n.$$

Zwischen den senkrechten Strichen stehen die Alternativen auf der rechten Seite.

Beispiel 6.7 Sei $\Sigma = \{S, A, a\}$ und sei

$$R = \{S \rightarrow aA, A \rightarrow aA \mid \varepsilon\}.$$

Mögliche Ableitungen sind (unter anderem)

$$\begin{aligned} S &\rightarrow aA \rightarrow a, \\ S &\rightarrow aA \rightarrow aaA \rightarrow aa, \\ S &\rightarrow aA \rightarrow aaA \rightarrow aaaA \rightarrow aaa, \end{aligned}$$

und so weiter.

Wenn wir mit dem Wort S beginnen, so ist zunächst nur die erste Regel anwendbar, die aA liefert. Darauf kann mehrmals die Regel $A \rightarrow aA$ angewendet werden, und jede Anwendung fügt ein weiteres a an den Anfang des

Wortes. Wenn irgendwann die Regel $A \rightarrow \varepsilon$ angewendet wird, wird das A „gelöscht“ und zurück bleibt eine Folge von „ a “s, die nicht weiter verändert werden kann.

Die mit diesem Regelsatz aus S ableitbaren Worte sind S , die Worte $a^n A$ für $n \geq 1$ und die Potenzen a^n für $n \geq 1$.

Um mit einem Produktionssystem eine Sprache zu beschreiben, wählt man ein festes Anfangswort (wie S in dem Beispiel) und definiert die Sprache als die Menge aller Worte, die durch Anwendung einer Folge von Regeln aus dem Anfangswort ableitbar sind. Oft schränkt man die Sprache ein auf die Menge aller *nicht weiter ableitbaren* Worte, die man aus dem Anfangswort ableiten kann.

Ein solcher Regelsatz für eine praktische Anwendung kann sehr umfangreich sein. Zum Beispiel hat das Produktionssystem für die Ausdrücke der Programmiersprache **Java** über 300 Regeln.

Die im vorletzten Absatz ausgesprochene Idee, den Endzustand (und den Anfangszustand) einer Ableitung genauer zu präzisieren und diese Merkmale ins Modell einzubinden, führt zu folgender von Chomsky stammenden Verfeinerung des Begriffs eines Produktionssystems, die schon in der Phrasenstrukturgrammatik auf Seite 263 verwendet wurde. Sie unterscheidet dort sichtbar den grammatischen Aufbau eines englischen Satzes vom fertig erzeugten Satz selber.

Definition 6.8 Eine *Regelgrammatik* ist ein Quadrupel

$$G := (\Psi, \Sigma, R, \sigma),$$

in dem Ψ und Σ Alphabete sind mit $\Psi \cap \Sigma = \emptyset$, in dem R ein Satz von *Regeln* der Gestalt $u \rightarrow v$ ist mit Worten u und v aus $(\Psi \cup \Sigma)^*$, so dass u mindestens ein Zeichen aus Ψ enthält, und in dem σ ein ausgezeichnetes Zeichen aus Ψ ist.

Wir werden später diesen Begriff weiter einschränken und verfeinern, so dass es eine ganze Hierarchie von Chomsky-Typen von Grammatiken gibt. Die hier definierte Version ist die allgemeinste und diese Regelgrammatiken werden deshalb auch *Grammatiken vom Typ 0* genannt.

Das Alphabet Ψ wird das *nicht-terminale Alphabet* genannt und seine Zeichen heißen *nicht-terminale Zeichen*, das Alphabet Σ heißt das *terminale Alphabet* und seine Zeichen werden *terminale Zeichen* genannt, und das nicht-terminale Zeichen σ heißt das *Startsymbol*.

Weil Ψ und Σ disjunkt sind, ist kein Zeichen gleichzeitig terminal und nicht-terminal. Weil jede Regel mindestens ein nicht-terminales Zeichen auf

der linken Seite haben muss, ist jedes Wort, das nur terminale Zeichen enthält, also jedes Wort in Σ^* , tatsächlich nicht mehr ableitbar und deshalb „terminal“.

Die Idee hinter dieser verfeinerten Definition besteht darin, im Produktionssystem (Γ, R) mit $\Gamma = \Psi \cup \Sigma$ nur Ableitungen zu betrachten, die mit dem Wort σ beginnen (deshalb heißt σ das Startsymbol) und die mit einem Wort aus Σ^* enden. Wir nennen die Worte aus Σ^* **terminale Worte**.

Im Sinne dieser Idee definieren wir die **von der Regelgrammatik G erzeugte (Regel-)Sprache** $L(G)$ als die Menge aller terminalen Worte, die aus dem Wort σ ableitbar sind. In anderen Worten, wir setzen

$$L(G) := \left\{ w \in \Sigma^* \mid \sigma \xrightarrow{*} w \right\}. \quad (6.8)$$

Wir bemerken zum Schluss, dass nicht verlangt wird, dass jede Ableitung aus σ unbedingt in einem terminalen Wort enden muss. Es ist durchaus möglich und zulässig, dass Worte mit nicht-terminalen Zeichen trotzdem nicht weiter ableitbar sind.

Das heißt, der Regelsatz muss nicht unbedingt zu jeder Situation mit nicht-terminalen Zeichen auch eine passende Regel enthalten. Aber nicht weiter ableitbare Worte mit nicht-terminalen Zeichen zählen nicht zur von der Grammatik erzeugten Sprache $L(G)$; die Sprache enthält *per Definition* nur Worte aus Σ^* .

Weil dies ein sehr wichtiger Begriff ist (und trickreicher, als er beim ersten Blick erscheint), präsentieren wir eine ganze Reihe von Beispielen.

Beispiele 6.9 a) Sei $\Sigma := \{a, b, c, d\}$ und sei L_1 die Sprache $\Sigma \cdot \Sigma$ aus allen Worten der Länge 2 über Σ .

Eine Grammatik, die L_1 erzeugt, ist die Grammatik

$$G_1 := (\{\sigma\}, \{a, b, c, d\}, \{\sigma \rightarrow aa|ab|ac|ad|ba|bb|bc|bd|ca|cb|cc|cd|da|db|dc|dd\}, \sigma).$$

Die Sprache L_1 hat 16 Worte und die Grammatik G_1 listet sie alle auf. Man kann aber die gleiche Sprache mit folgender kürzeren Grammatik erzeugen:

$$G_2 := (\{\sigma, \tau\}, \{a, b, c, d\}, \{\sigma \rightarrow \tau\tau, \tau \rightarrow a \mid b \mid c \mid d\}, \sigma).$$

G_2 hat nur 5 Regeln in Vergleich zu den 16 Regeln von G_1 .

Allerdings haben alle Ableitungen von terminalen Worten in der Grammatik G_1 die Länge 1, aber in G_2 die Länge 3 (denn es ist σ durch $\tau\tau$

zu ersetzen und dann *jedes* der beiden Taus durch ein Buchstabe zu ersetzen).

- b) Sei wieder $\Sigma := \{a, b, c, d\}$ und sei $L_2 = \Sigma^*$, d. h., L_2 bestehe aus *allen* Worten in den Buchstaben a, b, c und d .

Anders als L_1 in Teil a) hat L_2 unendlich viele Worte. Sie kann trotzdem von einer endlichen Grammatik erzeugt werden (das ist die Stärke der Produktionssysteme und war für Chomsky eines der Indizien für seine Behauptung, dass wir alle eine angeborene Kenntnis vom Prinzip der Produktionssysteme haben, denn auch natürliche Sprachen haben potentiell unendlich viele Aussagen, obwohl kein Mensch in seinem Leben mehr als endlich viele Beispiele zu hören bekommt).

Der Trick dabei ist *Rekursion*, wie in folgender Grammatik für L_2 .

$$G_3 := (\{\sigma\}, \{a, b, c, d\}, \{\sigma \rightarrow \varepsilon \mid a\sigma \mid b\sigma \mid c\sigma \mid d\sigma\}, \sigma).$$

Alle Regeln bis auf $\sigma \rightarrow \varepsilon$ können beliebig oft angewendet werden, um beliebig lange Worte aus den vier Buchstaben des Alphabets in beliebiger Kombination und beliebiger Reihenfolge zu erzeugen.

Auf die gleiche Weise wie in Teil a) kann man G_3 zu einer Grammatik G_4 abwandeln, die auch L_2 erzeugt, aber kürzere Regeln enthält.

$$G_4 := (\{\sigma, \tau\}, \{a, b, c, d\}, \{\sigma \rightarrow \varepsilon \mid \tau\sigma, \tau \rightarrow a \mid b \mid c \mid d\}, \sigma).$$

Allerdings hat G_4 eine Regel *mehr* als G_3 und die Ableitungen sind etwa doppelt so lang.

- c) Als praktisches Beispiel wollen wir die Grammatik für einen kleinen Ausschnitt aus der Programmiersprache **C** angeben.

In **C** gibt es *Zuweisungen* der Gestalt

Variable=Ausdruck;

wo die rechte Seite ein beliebiger Ausdruck ist und die Anweisung bewirkt, dass der Wert dieses Ausdrucks in der Variablen gespeichert wird oder wie man sagt, der Variablen *zugewiesen* wird.

Es gibt viele verschiedene Sorten von Ausdrücken, die auf der rechten Seite stehen könnten, aber wir wollen uns auf den einfachen Spezialfall beschränken, wo der Ausdruck nur eine (eventuell geklammerte) Summe und Differenz von (eventuell signierten) ganzen Dezimalzahlen oder

(eventuell signierten) Variablenwerten ist. Das heißt, die einzigen Operatoren, die in unserer stark eingeschränkten Version eines Ausdrucks vorkommen dürfen, sind $+$ und $-$ als unäre Operatoren (wie in „ -3 “) oder als binäre Operatoren (wie in „ $2 - 3$ “). Die in **C** implementierten Inkrement- und Dekrementoperatoren $++$ und $--$ wollen wir nicht berücksichtigen.

Wir wollen eine Regelgrammatik entwickeln, deren Sprache aus allen in **C** zulässigen Zuweisungen besteht, in denen der Ausdruck auf der rechten Seite von der beschriebenen eingeschränkten Gestalt ist.

Damit man den Sinn der Regeln dieser Grammatik besser versteht, wollen wir uns erlauben, fettgedruckte ganze Worte als „Zeichen“ des nicht-terminalen Alphabets zu verwenden, etwa **Zahl** anstelle eines Zeichens wie ζ für ein nicht-terminales Zeichen, das eine Zahl darstellen soll oder sich zu einer Zahl ableiten lassen soll, und so weiter.

Als *terminales Alphabet* Σ werden wir den gesamten ASCII Zeichensatz verwenden (geschrieben in schreibmaschinenähnlichen nicht-proportionalen Typen a b c und so weiter).

Das *nicht-terminale Alphabet* Ψ besteht aus folgenden „Zeichen“:

$$\Psi = \{ \text{Zuweisung, Variable, Ausdruck, alphanum, Buchstabe, Ziffer, pZiffer, Zahl, pZahl, Term, unsig} \}.$$

Das *Startsymbol* ist **Zuweisung**.

Der *Regelsatz* R besteht aus folgenden Regeln:

$$\begin{aligned} \text{Zuweisung} &\rightarrow \text{Variable} = \text{Ausdruck}; \\ \text{Variable} &\rightarrow \text{Variable alphanum} \mid \text{Buchstabe} \\ \text{alphanum} &\rightarrow \text{Buchstabe} \mid \text{Ziffer} \\ \text{Buchstabe} &\rightarrow _ \mid \text{a} \mid \text{b} \mid \dots \mid \text{z} \mid \text{A} \mid \text{B} \mid \dots \mid \text{Z} \\ \text{Ziffer} &\rightarrow 0 \mid \text{pZiffer} \\ \text{pZiffer} &\rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \text{Zahl} &\rightarrow 0 \mid \text{pZahl} \\ \text{pZahl} &\rightarrow \text{pZahl Ziffer} \mid \text{pZiffer} \\ \text{Ausdruck} &\rightarrow \text{Term} \mid \text{Ausdruck} + \text{unsig} \mid \text{Ausdruck} - \text{unsig} \\ \text{Term} &\rightarrow \text{unsig} \mid + \text{unsig} \mid - \text{unsig} \\ \text{unsig} &\rightarrow \text{Variable} \mid \text{Zahl} \mid (\text{Ausdruck}) \end{aligned}$$

Einige Bemerkungen zu dieser Grammatik sind erforderlich. Natürlich haben wir in diesem Beispiel nicht alle Feinheiten der Sprache **C** berücksichtigen können, auch abgesehen davon, dass wir nur eine sehr kleine Klasse von Ausdrücken beschrieben haben (aber das hatten wir ja schon vorher angekündigt).

In vielen **C** Implementierungen gibt es Einschränkungen über die maximale Länge eines Bezeichners, d. h., eines Variablennamens, oder es gibt zwar keine Einschränkungen über die Gesamtlänge, aber nur die ersten soundsoviele Zeichen des Namens werden bei der Identifizierung von Variablen berücksichtigt. Diese Details haben wir nicht erfasst.

Die Sprache **C** hat eine ganze Reihe von Schlüsselwörtern wie **if**, **else**, **while** usw., die als Variablennamen verboten sind — natürlich haben wir auch diese Bestimmung nicht in unsere Grammatik eingebaut.

Auch Konstanten dürfen in **C** benannt werden, und Konstantennamen folgen den gleichen Regeln wie Variablennamen (es gibt allerdings eine Konvention, dass Konstantennamen in Großbuchstaben geschrieben werden und Variablennamen in Kleinbuchstaben — das ist aber keine Regel der Sprache, sondern soll nur die Lesbarkeit und Verständlichkeit von Programmen erhöhen). Das nicht-terminale Zeichen **Variable** in unserer Grammatik kann also auch eine Konstante bezeichnen oder sich zu einem Konstantennamen ableiten lassen.

In den Ausdrücken auf der rechten Seite einer Zuweisung dürfen neben Variablen und benannten Konstanten auch Zahlen vorkommen. Wir haben uns auf ganze Zahlen in *Dezimaldarstellung* festgelegt. In **C** sind auch andere Zahlenformate zulässig, darunter Gleitkommazahlen, lange Ganzzahlen, unsignierte Ganzzahlen, Hexadezimalzahlen und Oktalzahlen, aber in der Bezeichnung dieser anderen Zahlenarten kommen Zeichen vor, die wir nicht zugelassen haben, wie der Dezimalpunkt bei Gleitkommazahlen, die Ziffern **A** bis **F** und ein kennzeichnendes **x** in der Zahlendarstellung bei Hexadezimalzahlen, und Formatzusätze wie **U** für unsignierte Zahlen oder **L** für Langzahlen (Zahlen mit mehr Ziffern und deshalb einen größeren Speicherbedarf als „normale“). *Oktalzahlen* werden in **C** dadurch gekennzeichnet, dass sie mit einer Null beginnen, und deshalb haben wir in unserer Grammatik ausdrücklich vorgesehen, dass Zahlen außer 0 nicht mit der Ziffer 0 beginnen dürfen.

Auch den Regelsatz wollen wir kurz erläutern. Die erste Regel beschreibt den groben Aufbau einer Zuweisung.

Die zweite Regel hat zwei Varianten, wobei die erste Variante rekursiv ist und bei wiederholter Anwendung beliebig viele alphanumerische

Zeichen (das heißt, Buchstaben *oder* Ziffern) ans rechte Ende eines Variablennamens anfügt, während die zweite Variante die Rekursion abbricht und ein terminales Wort als Ergebnis liefert, in dem das *erste* oder das *linke* Zeichen ein Buchstabe ist. Das entspricht den Syntaxregeln von **C**; Variablennamen müssen mit einem Buchstaben beginnen. Beachten Sie, dass auch der Unterstrich `_` als Buchstabe zählt!

Das nicht-terminale Zeichen **Zahl** soll bei der Ableitung *unsignierte* Dezimalzahlen der für die Maschine natürlichen Größe (in **C** Terminologie: vom Datentyp **short**) als Zahlennamen ohne Vorzeichen erzeugen. Die Regel mit **Zahl** auf der linken Seite leitet dieses Zeichen unmittelbar ab zu der Zahl 0 oder zu einer *positiven* Zahl **pZahl**.

Die Regel mit **pZahl** auf der linken Seite ist so strukturiert, wie die Regel für **Variable**. Sie hat zwei Varianten, wovon die erste rekursiv ist und beliebig viele Ziffern (0 oder positive Ziffern) ans rechte Ende der Zahl anfügt. Die zweite Variante fixiert das Ergebnis mit einer *positiven* Ziffer (1 bis 9) am linken Ende. Die Zahl Null muss hier nicht berücksichtigt werden, weil sie in der Regel für **Zahl** schon erfasst ist.

Die letzten drei Regeln sind auf zyklische Weise miteinander verbunden und beinhalten deshalb eine Rekursion. Die Rückbezüge, zum Beispiel von der letzten Regel auf die drittletzte, machen es ein bisschen schwieriger zu verstehen, was diese Regeln tun.

Die Grundeinheiten in einem Ausdruck (von der Sorte, die wir betrachten wollen), d. h., die Basiskomponenten oder **Atome**, aus denen Ausdrücke zusammengesetzt werden, sind unsignierte Terme, also als eine feste Einheit gesehene Werte ohne Vorzeichen.

Variablen und unsignierte Zahlen stellen solche atomare Werte dar, aber auch zusammengesetzte Ausdrücke können zu einem unzertrennlichen atomaren Block fixiert werden, indem man sie in Klammern setzt. Das sind genau die drei Varianten von „atomaren“ Werten, zu denen sich das Zeichen **unsig** in der letzten Regel direkt ableiten lässt.

Das Zeichen **unsig** repräsentiert einen unsignierten Wert. Auf diesen können die *unären* Operatoren `+` und `-` als Vorzeichen wirken, um daraus einen signierten Wert zu machen. Der Sammelbegriff, der signierte und unsignierte Werte zusammenfasst, ist der **Term**. Terme sind Einzelwerte, die signiert sein dürfen (aber nicht sein müssen). Die vorletzte Regel spiegelt diese Definition des Begriffs **Term** wieder.

Ein einzelner Term bildet schon einen Ausdruck für sich, aber ein Ausdruck kann auch eine Summe oder Differenz von Werten sein. Hier sind

auch Mehrfachsummen erlaubt mit allen möglichen Mischungen von $+$ und $-$ Operationen.

Dieser Aufbau wurde in der drittletzten Regel festgehalten. Die erste Variante der Regel erzeugt Ausdrücke, die nur aus einem Term bestehen, und die beiden letzten Varianten erzeugen Summen und Differenzen.

Die Summen und Differenzen werden rekursiv erzeugt, wobei eine Mehrfachsumme bezüglich des *letzten* in ihr erscheinenden binären Operators in zwei Summanden zerlegt wird. Der linke Summand darf ein beliebiger Ausdruck sein, mit Vorzeichen und mit anderen binären Operationen. Aber der rechte Summand darf selber kein Vorzeichen tragen (denn Ausdrücke wie „ $2 + -3$ “ sind nicht zulässig¹); dem wird Rechnung getragen, indem der rechte Summand in der Grammatik als **unsig** festlegt wird.

Ein Ausdruck darf nicht nur $+$ und $-$ Operationen enthalten; er darf auch *ganz* in Klammern gesetzt werden. Solche Ausdrücke, wie zum Beispiel $(2 - 3 + 7)$, werden von der *ersten* Variante der **Ausdruck** Regel erzeugt, wobei der Klammerausdruck durch eine Anwendung der letzten Variante der **unsig** Regel entsteht, gefolgt durch eine Ableitung des eingeklammerten **Ausdrucks** zu $2 - 3 + 7$.

d) Sei $\Sigma = \{a\}$. Wir betrachten die Sprache

$$L_3 = \{a^{2^n} \mid n \in \mathbf{N}\} = \{a, a^2, a^4, a^8, a^{16}, \dots\}$$

bestehend aus allen Potenzen des einzigen terminalen Zeichens a , in denen der Exponent eine Zweierpotenz ist.

Um die Worte der Sprache L_3 zu erzeugen, müssen wir vom Wort a ausgehen und dann die Anzahl der a s einige Male verdoppeln.

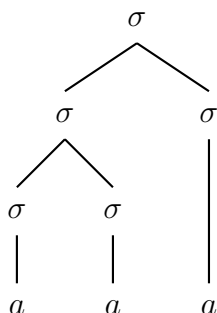
Die Grammatik

$$(\{\sigma\}, \{a\}, \{\sigma \rightarrow \sigma\sigma \mid a\}, \sigma)$$

stellt einen naiven Versuch dar, das Gewünschte zu bewerkstelligen, aber diese Idee funktioniert nicht, weil die verdoppelnde Rekursionsregel auf verschiedene Exemplare von σ verschieden oft angewendet werden darf.

So zeigt der Ableitungsbaum

¹Natürlich darf man in einem Ausdruck -3 zu 2 addieren, aber dazu muss man Klammern verwenden und „ $2 + (-3)$ “ schreiben. Das erlaubt auch unsere Grammatik, denn (-3) ist ein *unsignierter* Term.



eine in dieser Grammatik erlaubte Entwicklung mit der Ableitungsfolge

$$\sigma \rightarrow \sigma\sigma \rightarrow \sigma a \rightarrow \sigma\sigma a \rightarrow a\sigma a \rightarrow aaa.$$

Sie erzeugt das Wort a^3 , das *nicht* zu L_3 gehört, weil 3 keine Zweierpotenz ist.

Eine richtige Grammatik für diese Sprache muss die Ableitungen genauer kontrollieren und erzwingen, dass alle Bestandteile des entstehenden Wortes *gleich oft* verdoppelt werden. Dann kann die endgültige Anzahl von Buchstaben a nur eine Zweierpotenz werden.

Der Trick, um diese Operation (Verdoppelung der as) präzise durchzuführen, besteht darin, das nicht-terminale Alphabet durch zusätzliche Symbole zu erweitern, die unter der Kontrolle der Regeln der Grammatik wie kleine Roboter oder kleine Werkzeuge funktionieren, die an jeder vorgesehenen Stelle im Wort die erforderliche Operation auf den terminalen Zeichen genau wie gewünscht ausführen. Weitere nicht-terminale Zeichen werden danach eingesetzt, um „aufzuräumen“ und die Werkzeuge zu entfernen, wenn ihre Arbeit getan ist.

In unserem Beispiel benötigen wir ein „Werkzeugzeichen“ μ , das die Funktion hat, im Vorbeigehen die as zu verdoppeln, und zum Aufräumen eine Art „Antiteilchen“ ν zu μ , das die μ s anschließend entfernt, indem beide Zeichen sich gegenseitig vernichten.

Die Details sind dann wie folgt:

$$G_5 = (\{\sigma, \mu, \nu\}, \{a\}, \{\sigma \rightarrow \mu\sigma\nu \mid a, \mu a \rightarrow aa\mu, \mu\nu \rightarrow \varepsilon\}, \sigma)$$

Wiederholtes Anwenden der ersten Regel für σ schließt σ in eine Anzahl von μ - ν -Paaren ein, wobei jedes solche Paar eine Verdoppelung aller innerhalb des Paares entstandenen as bewirken wird. Die Rekursion wird verlassen, indem irgendwann mit der zweiten σ -Regel das zwischen den μ s und ν s begrabene σ durch das terminale Zeichen a ersetzt wird.

Die Regel mit μa auf der linken Seite lässt jedes μ an *allen* a s vorbei nach rechts wandern, wobei *jedes* a , über das das μ „hinwegrutscht“, dabei verdoppelt wird. Wenn alle a s auf diese Weise von einem μ „verarbeitet“ wurden, erreicht das μ sein Partner- ν (das mit ihm bei der σ -Rekursion am Anfang entstanden ist). Dann ist für dieses μ nur noch die letzte Regel anwendbar, die das μ - ν -Paar verschwinden lässt.

Wenn alle μ s auf diese Weise ihren Lebenszyklus beendet haben, bleibt ein terminales Wort zurück, das nur aus a s besteht, und deren Gesamtanzahl ist durch eine Folge von Verdoppelungen entstanden und ist deshalb eine Zweierpotenz.

Man beachte, dass in Zwischenstadien dieses Prozesses mehrere μ s gleichzeitig „unterwegs“ sein können, und dass die Ablaufreihenfolge deshalb nicht eindeutig ist. Aber nur die Verschachtelungstiefe bestimmt, an wie vielen a s jedes μ vorbei muss, und das μ kann erst vernichtet werden, wenn es an allen a s, für das es zuständig ist, vorbeigekommen ist, so dass unabhängig von der Ausführungsreihenfolge das *Endergebnis* eindeutig ist, sobald die σ -Rekursionen durch eine Anwendung der Regel $\sigma \rightarrow a$ abgebrochen wurden.

Auch die Anzahl der Regelanwendungen ist (nach Ende der σ Rekursion) eindeutig bestimmt und endlich, so dass jede Ableitung, die nicht *nur* die Regel $\sigma \rightarrow \mu\sigma\nu$ ausführt, irgendwann ganz enden muss und ein terminales Wort aus L_3 liefert.

Eine typische Ableitung mit dieser Grammatik könnte zum Beispiel wie folgt aussehen:

$$\begin{aligned} \sigma &\rightarrow \mu\sigma\nu \rightarrow \mu\mu\sigma\nu\nu \rightarrow \mu\mu\mu\sigma\nu\nu\nu \rightarrow \mu\mu\mu a\nu\nu\nu \rightarrow \\ &\mu\mu a a a \mu\nu\nu\nu \rightarrow \mu\mu a a \nu\nu \rightarrow \mu a a \mu a \nu\nu \rightarrow a a \mu a \mu a \nu\nu \rightarrow \\ &a a \mu a a a \mu\nu\nu \rightarrow a a a a \mu a a \mu\nu\nu \rightarrow a a a a a a \mu a \mu\nu\nu \rightarrow \\ &a a a a a a \mu a \nu \rightarrow a a a a a a a a \mu\nu \rightarrow a a a a a a a a = a^8. \end{aligned}$$

- e) Unser letztes Beispiel wurde als Übungsaufgabe 11-B Teil d) in der vorletzten Übungsstunde gestellt. Da es nie zu einer Besprechung in den Übungstunden kam, wollen wir die Lösung hier erläutern.

Gesucht wird eine Regelgrammatik G_6 , die die Sprache

$$L_4 := \{ ww \mid w \in \{a, b, c\}^* \}$$

erzeugt. Diese Sprache besteht aus allen „Doppelworten“ in den Buchstaben a , b und c , die eine gerade Länge haben und bei denen die zweite Hälfte des Wortes eine genaue Kopie der ersten Hälfte ist.

Auch für diese Aufgabe gibt es keine einfache naive Lösung, und wir werden wieder „Werkzeugzeichen“ oder „Agenten“ einsetzen müssen.

Wir wollen kurz überlegen, wie viele „Agenten“ wir benötigen, und für welche Zwecke. Die Grammatik soll zunächst nur eine einfache Kopie eines beliebigen Wortes in Σ^* erzeugen, und die Agenten werden dafür verantwortlich sein, eine Kopie dieses Wortes zu erstellen und die einzelnen Buchstaben der Kopie in der richtigen Reihenfolge an der richtigen Stelle abzusetzen.

Neben den Werkzeugen für die eigentlichen Aufgaben des Kopierens und des Transports brauchen wir auch Platzhalter, um die richtige Ablagestelle für die kopierten Buchstaben finden zu können, und Markierungen, anhand derer wir erkennen können, wann wir alle Daten schon verarbeitet haben. Ganz zum Schluss wird für die in der Zeichenkette verbliebenen Werkzeuge je ein „Antiteilchen“ benötigt, um die Werkbank aufzuräumen.

Zu diesen Zwecken führen wir neben unserem Startsymbol σ folgende nicht-terminale Zeichen ein:

- ein Hilfszeichen τ für die Erzeugung des ursprünglichen einfachen Wortes w , das anschließend kopiert werden muss;
- einen Kopierer μ , der von jedem Zeichen im ursprünglichen Wort w eine Kopie erstellt und mit einem
- Transportagenten ρ ausstattet, der das kopierte Zeichen an seinen richtigen Platz bringt;
- einen Platzhalter ν , der die Grenze zwischen dem ursprünglichen Exemplar des Wortes w und der Kopie markiert und so angibt, wo das nächste kopierte Zeichen abzulegen ist;
- eine Markierung δ für den linken Rand des Arbeitsbereichs, mit der wir erkennen können, wann alle Daten kopiert worden sind;
- ein ν -Antiteilchen ω , mit dem wir am Ende die Grenzmarkierung ν löschen können.

Hier nun die Daten für die gesuchte Grammatik G_6 :

Der nicht-terminale Zeichensatz ist

$$\Psi := \{ \sigma, \delta, \mu, \nu, \rho, \tau, \omega \}.$$

Der terminale Zeichensatz ist laut Vorgabe

$$\Sigma := \{ a, b, c \}.$$

Der Regelsatz R besteht aus folgenden Regeln:

$$\begin{aligned}
 \sigma &\rightarrow \delta\tau\mu\nu \\
 \tau &\rightarrow a\tau \mid b\tau \mid c\tau \mid \varepsilon \\
 a\mu &\rightarrow \mu a\rho a \\
 b\mu &\rightarrow \mu b\rho b \\
 c\mu &\rightarrow \mu c\rho c \\
 \rho aa &\rightarrow a\rho a \\
 \rho ab &\rightarrow b\rho a \\
 \rho ac &\rightarrow c\rho a \\
 \rho ba &\rightarrow a\rho b \\
 \rho bb &\rightarrow b\rho b \\
 \rho bc &\rightarrow c\rho b \\
 \rho ca &\rightarrow a\rho c \\
 \rho cb &\rightarrow b\rho c \\
 \rho cc &\rightarrow c\rho c \\
 \rho a\nu &\rightarrow \nu a \\
 \rho b\nu &\rightarrow \nu b \\
 \rho c\nu &\rightarrow \nu c \\
 \delta\mu &\rightarrow \omega \\
 \omega a &\rightarrow a\omega \\
 \omega b &\rightarrow b\omega \\
 \omega c &\rightarrow c\omega \\
 \omega\nu &\rightarrow \varepsilon
 \end{aligned}$$

Das Startsymbol für die Grammatik ist natürlich σ .

Es ist nicht schwer zu verstehen, wie die Regeln in dieser Grammatik funktionieren und die gewünschte Wirkung erzielen.

Die erste Regel stellt im Wesentlichen das Werkzeug auf den Arbeitstisch: die linke Randmarkierung δ , den Kopierer μ und den Platzanweiser ν für die Kopie, und zwischen dem Werkzeug den Erzeuger τ für das einfache Wort w (das später kopiert werden muss).

Die zweite Regel hat rekursive Varianten, die vor dem τ eine beliebige Folge von as , bs und cs erzeugen können, und eine Abbruchvariante, die die Rekursion stoppt und den Erzeuger τ entfernt. Zurück bleibt dann

ein Wort w aus Σ^* , in der Umgebung

$$\delta w \mu \nu.$$

Die nächsten drei Regeln implementieren die Kopierfunktion von μ . Mit diesen Regeln rutscht μ an jedem Zeichen von w vorbei und erzeugt dabei ein zweites Exemplar des Zeichens versehen mit einem Transporter ρ auf seiner linken Seite.

Die folgenden $3 \times 3 = 9$ Regeln implementieren die Transportfunktion von ρ . Ihre allgemeine Gestalt ist

$$\rho xy \rightarrow y \rho x,$$

mit der Wirkung, dass ρ und das rechts von ρ stehende Zeichen x an dem rechts von diesem „Gespann“ stehenden Zeichen y aus Σ einfach vorbeirutschen.

Der Transport endet erst, wenn das ρ -Gespann auf den Markierer ν trifft; dann greifen die nächsten drei Regeln und das transportierte Zeichen wird rechts vom ν abgesetzt und der Transporter ρ , der seine Aufgabe damit erfüllt hat, verschwindet.

Man beachte, dass ein ρ -Zeichen-Gespann nur an Zeichen aus Σ vorbeirutschen kann (es begegnet dabei nur die schon vorher kopierten Zeichen aus dem ursprünglichen Exemplar von w), aber nicht andere solche ρ -Gespanne überholen kann, da es keine ρ -Regel gibt, in der das dritte Zeichen auf der linken Seite auch ein ρ ist. Die Regeln mit ν als drittes Zeichen links setzen die Zeichen nur ab, in der Reihenfolge, in der sie ankommen, und beenden dann den Transport. Deshalb kann die Reihenfolge der Zeichen in w beim Transport der Kopie nicht durcheinanderkommen; die Kopie behält die ursprüngliche Zeichenanordnung bei.

Der Kopierer μ kopiert alle *Terminalzeichen*, die es auf seiner linken Seite findet, und schickt die Kopien auf die Reise, aber wenn es an allen Zeichen aus w vorbeigekommen ist, trifft es auf den Randmarkierer δ . In diesem Moment hat μ seine Funktion erfüllt, aber es können noch kopierte Zeichen zu ihrer endgültigen Lage unterwegs sein, und der Platzhalter ν ist noch vorhanden.

Die fünftletzte Regel $\delta \mu \rightarrow \omega$ entfernt μ und erzeugt zum Aufräumen einen „ ν -Vernichter“ ω . Die nächsten drei Regeln ermöglichen ω , sich nach rechts zu bewegen auf der Suche nach dem zu löschenden ν . Da keine Regel ω ermöglicht, an einem ρ vorbeizurutschen, kann ω erst

dann auf ν treffen, wenn alle ρ -Transporte beendet sind und alle kopierten Zeichen ihre vorgesehene Lage erreicht haben.

Wenn aber alle Transporte am Ziel angelangt sind, dann muss ω schließlich auf ν treffen. Dann greift die letzte Regel und entfernt ω und ν als letzte vorhandene nicht-terminale Zeichen aus dem „Werkstück“. Zurück bleibt ein Wort ww der Sprache L_4 .

Wir illustrieren diesen Prozess mit einer typischen Ableitung, wo in jedem Zwischenergebnis durch Fettdruck gekennzeichnet wird, auf welches Teilwort der nächste Ableitungsschritt wirkt. Daraus kann man eindeutig ablesen, welche Regeln aus R in welcher Reihenfolge angewendet werden.

Das Zielwort der vorgeführten Ableitung ist $abbcabbc$ (mit $abbc$ in der Rolle von w).

Dieses Wort erzeugt unsere Grammatik mit der Schrittfolge

$$\begin{aligned}
\sigma &\rightarrow \delta\tau\mu\nu \rightarrow \delta a\tau\mu\nu \rightarrow \delta ab\tau\mu\nu \rightarrow \delta abb\tau\mu\nu \rightarrow \delta abbc\tau\mu\nu \rightarrow \\
&\delta abbc\mu\nu \rightarrow \delta abbc\mu c\nu \rightarrow \delta ab\mu b\mathbf{p}bc\nu \rightarrow \delta ab\mu bc\mathbf{p}bc\nu \rightarrow \\
&\delta ab\mu bc\mathbf{p}b\nu c \rightarrow \delta ab\mu bc\nu bc \rightarrow \delta a\mu b\mathbf{p}bb\nu bc \rightarrow \delta \mu a\mathbf{p}ab\mathbf{p}bb\nu bc \rightarrow \\
&\omega a\mathbf{p}ab\mathbf{p}bb\nu bc \rightarrow a\omega\mathbf{p}ab\mathbf{p}bb\nu bc \rightarrow a\omega b\mathbf{p}a\mathbf{p}bb\nu bc \rightarrow \\
&a\omega b\mathbf{p}ab\mathbf{p}bc\nu bc \rightarrow a\omega b\mathbf{p}abc\mathbf{p}b\nu bc \rightarrow a\omega b\mathbf{p}ac\mathbf{p}b\nu bc \rightarrow \\
&a\omega b\mathbf{p}ac\nu bbc \rightarrow a\omega b\mathbf{p}ac\nu bbc \rightarrow ab\omega bc\mathbf{p}a\nu bbc \rightarrow \\
&ab\omega bc\nu abbc \rightarrow abb\omega c\nu abbc \rightarrow abbc\omega\nu abbc \rightarrow abbcabbc
\end{aligned}$$

Die Beispiele haben uns einen gewissen Überblick gegeben über die Vielfältigkeit von Regelgrammatiken und den verschiedenen Sprachstrukturen, die man damit beschreiben kann. Für viele Zwecke sind aber möglichst einfache Grammatiken von Vorteil, aus dem offensichtlichen Grund, dass eine einfachere Struktur der Grammatik es leichter macht, das Verhalten der Grammatik zu untersuchen oder Aussagen über die erzeugte Sprache zu beweisen, aber auch weil beim praktischen Einsatz (etwa im Übersetzer- und Compilerbau) einfache Grammatiken leichter zu implementieren sind.

Wir wollen deshalb unsere bisherige Definition 6.8 einer Regelgrammatik weiter einschränken, um Grammatiken mit einer einfacheren Struktur zu erhalten. Diese Grammatiken werden nicht mehr alle Aufgaben erledigen können, die wir in den Beispielen betrachtet haben, aber was für die Informatik wichtig ist, ist dass sie für die Definition und Beschreibung der gängigen Computersprachen wie PASCAL, C, Java usw. immer noch ausreichen werden.

Es geht uns übrigens nicht darum, die Sprachen selber zu verändern oder zu vereinfachen, sondern nur darum, für eine vorgegebene und bekannte Sprache eine Grammatik *möglichst einfacher Struktur* zu finden, die diese Sprache beschreibt. Wir wollen also verschiedene Grammatiken für die gleiche Sprache vergleichen, ein Umstand, für den es folgenden Begriff gibt.

Definition 6.10 Zwei Regelgrammatiken G_1 und G_2 heißen *äquivalent*, und wir schreiben

$$G_1 \sim G_2,$$

wenn

- G_1 und G_2 das gleiche terminale Alphabet Σ haben, und
- $L(G_1) = L(G_2) \subseteq \Sigma^*$.

(In anderen Worten, äquivalente Grammatiken erzeugen die gleiche Sprache über das gleiche Alphabet.)

In Beispielen 6.9 d) und 6.9 e) funktionieren die benötigten Werkzeugzeichen mit Hilfe von Regeln, die gewisse Mehrzeichenkonstellationen auf der linken Seite haben und daraus die gewünschte Wirkung auf der rechten Seite der Regel entwickeln.

Für weniger komplizierte Sprachen ist es von Vorteil, Zeichenkonstellationen (d. h., ein Zeichen mit seinem Umfeld oder sein *Kontext*) auf der linken Seite einer Regel zu verbieten. Ableitungen werden dadurch viel leichter zurückzuverfolgen, und das „Parsen“, also die algorithmische Zerlegung einer Zeichenfolge in ihre grammatischen Bestandteile, wird einfacher, was zum Beispiel im Compilerbau ein großer Vorteil ist.

Dieser Gedanke führt uns zur Klasse der *kontextfreien* Grammatiken und Sprachen.

Definition 6.11 Eine Regelgrammatik

$$G = (\Psi, \Sigma, R, \sigma)$$

heißt *kontextfrei* oder *von Typ 2*, wenn jede Regel aus R die Gestalt

$$\xi \rightarrow w$$

hat, wobei die Länge von ξ eins ist, d. h., ξ ist ein *einzelnes* Zeichen (aus dem nicht-terminalen Zeichensatz Ψ , weil die linke Seite einer Regel mindestens ein nicht-terminales Zeichen enthalten muss).

Eine *Sprache* $L \subseteq \Sigma^*$ heißt *kontextfrei*, wenn es eine kontextfreie Grammatik G gibt mit $L(G) = L$.

(Das bedeutet nicht, dass jede L erzeugende Grammatik kontextfrei sein muss.)

Wir haben in vielen der Beispielgrammatiken Rekursionsregeln eingebaut und sie durch eine Löschregel der Gestalt $\xi \rightarrow \varepsilon$ ergänzt, um die Rekursion zu stoppen. Solche Regeln müssen aber zwingend eingesetzt werden; wir werden in Zukunft Grammatiken betrachten, die ohne Löschregeln auskommen.

Definition 6.12 Eine kontextfreie Regelgrammatik

$$G = (\Psi, \Sigma, R, \sigma)$$

heißt ε -frei, wenn jede Regel aus R die Gestalt

$$\xi \rightarrow w$$

hat mit $L(w) > 0$ (in anderen Worten, G hat keine Regeln mit leerer rechter Seite).

Die definierende Grammatik für **Java** ist zum Beispiel ε -frei, aber man kann tatsächlich *jede* kontext-freie Sprache, die nicht das leere Wort enthält, durch eine ε -freie Grammatik erzeugen.

Wir betrachten dazu zuerst ein Beispiel, um zu überlegen, wie man ε -Regeln aus einer Grammatik entfernen kann, und andere Regeln einführen kann, die ihre Wirkung „übernehmen“.

Beispiel 6.13 Wir betrachten die kontextfreie Grammatik G_7 mit dem Regelsatz R bestehend aus den Regeln

$$S \rightarrow AB$$

$$A \rightarrow a \mid \varepsilon$$

$$B \rightarrow bC$$

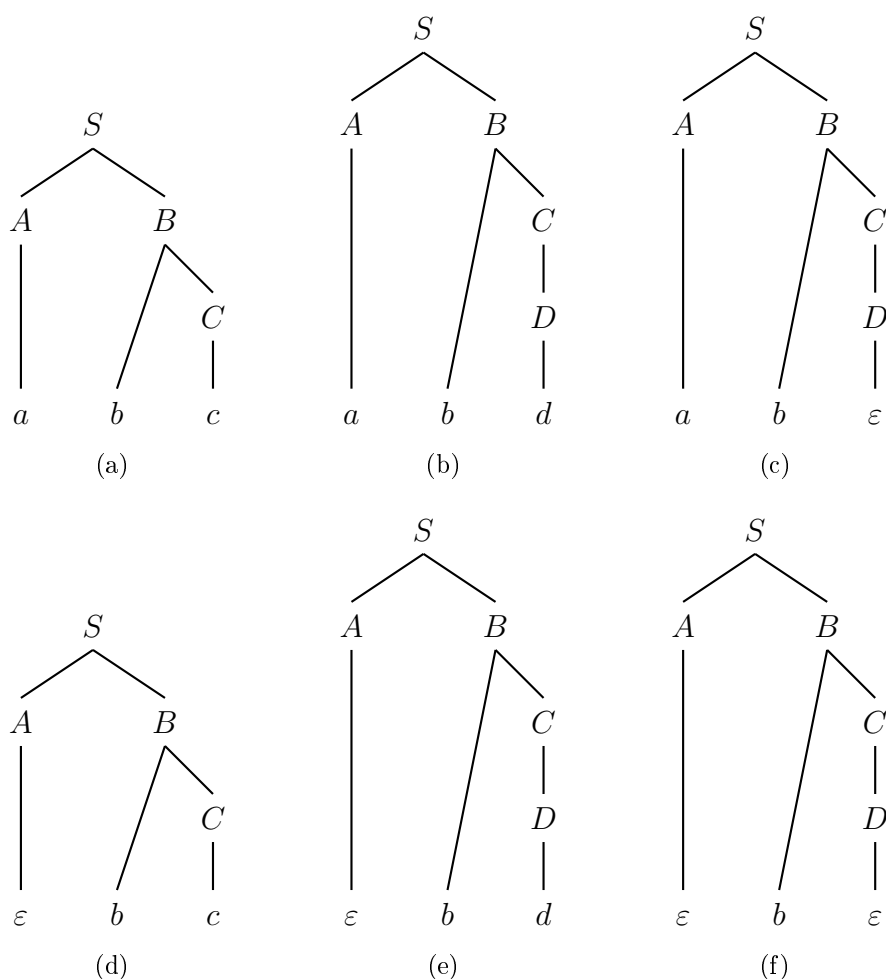
$$C \rightarrow c \mid D$$

$$D \rightarrow d \mid \varepsilon$$

Die Großbuchstaben sind nicht-terminale Zeichen, die Kleinbuchstaben sind terminale Zeichen und das Startsymbol ist S .

Es gibt keine rekursiven Regeln und deshalb insgesamt nur endlich viele verschiedene Verzweigungsmöglichkeiten, die zu den Ableitungsbäumen in Abbildung 6.1 auf der nächsten Seite führen und somit zu den nur 6 Worten der Sprache

$$L(G_7) = \{ abc, abd, ab, bc, bd, b \}.$$

Abbildung 6.1: Alle Ableitungsbäume der Grammatik G_7

Um die Grammatik so umzuformen, dass sie die gleiche Sprache erzeugt, *ohne* ε -Regeln zu enthalten, darf man sich nicht naiv vorstellen, dass, weil die ε -Regeln ohnehin zu nichts führen, man sie einfach weglassen kann. Das würde aus den Ableitungsbäumen die ε -Zweige entfernen, und der Stumpf an der Amputationsstelle würde dann in allen Fällen in einem nicht-terminalen Zeichen enden, so dass die entsprechende Ableitung gar kein (terminales!) Wort der Sprache mehr erzeugen würde.

Da nur die Ableitungen 6.1(a) und 6.1(b) keine ε -Zweige enthalten, würden nur diese noch Worte der erzeugten Sprache liefern. Dies wären auch die einzigen vollen Ableitungsbäume der neuen Grammatik mit dem beschnitte-

nen Regelsatz

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow a \\ B &\rightarrow bC \\ C &\rightarrow c \mid D \\ D &\rightarrow d \end{aligned}$$

Die Sprache hätte sich dann reduziert zu

$$\{ abc, abd \}$$

und wäre nicht die Gleiche geblieben!

Trotzdem ist es möglich, *ohne* Änderung der Sprache die Grammatik zu einer ε -freien Grammatik umzubauen. Dazu muss man aber die Wirkung der ε -Regeln *zurückverfolgen* und durch geänderte Regeln darstellen in allen Aspekten, bevor man die ε -Regeln entfernen darf. Das erläutern wir im Beweis des folgenden Satzes:

Satz 6.14 *Sei G eine kontextfreie Grammatik. Dann existiert eine zu G „fast äquivalente“ ε -freie Grammatik G' , so dass*

$$L(G') = L(G) \setminus \{ \varepsilon \}.$$

Beweis. Wir bestimmen zunächst die Menge

$$E := \left\{ \xi \in \Psi \mid \xi \xrightarrow{*} \varepsilon \right\}$$

der zum leeren Wort ableitbaren nicht-terminalen Zeichen.

Dazu sei

$$E_1 := \left\{ \xi \in \Psi \mid \xi \xrightarrow{R} \varepsilon \right\}$$

die Menge der *direkt* zu ε ableitbaren Zeichen, und für jede positive natürliche Zahl i sei

$$E_{i+1} := E_i \cup \left\{ \xi \in \Psi \mid \text{es gibt } w \in E_i^* \text{ mit } \xi \xrightarrow{R} w \right\}.$$

Beachte, dass E_i immer eine Teilmenge von Ψ und somit eine endlichen Menge von Zeichen ist, und dass E_{i+1} durch Inspektion der endlich vielen Regeln effektiv aus E_i bestimmt werden kann. Und alle Zeichen in E_{i+1} sind zu ε ableitbar.

Die Menge E_i besteht aus allen Zeichen $\xi \in \Psi$, die eine Ableitung höchstens der Länge i zu ε besitzen, und da jedes Element von E eine Ableitung *endlicher* Länge zu ε hat, ist

$$E = \bigcup_{i=1}^{\infty} E_i.$$

Aus der Definition ist aber klar, dass

$$E_1 \subseteq E_2 \subseteq \cdots \subseteq E_i \subseteq E_{i+1} \subseteq \cdots \subseteq E \subseteq \Psi.$$

Weil Ψ eine endliche Alphabet ist, können die Mengen E_i nicht *immer* größer werden, und wenn einmal gilt $E_i = E_{i+1}$, dann sind auch alle weiteren $E_j = E_i$ für $j > i$, so dass für dieses i auch gilt $E = E_i$.

Das heißt, dass E in endlich vielen Schritten durch Betrachtung der Regeln effektiv bestimmt werden kann (für den Beweis des Satzes ist das nicht wichtig, für die praktische Anwendung aber schon!).

Wir ersetzen jetzt den Regelsatz R durch einen neuen Regelsatz R' , indem wir für jede Regel $\xi \rightarrow w$ aus R und für *jede Auswahl* von Instanzen von Zeichen aus E in w genau diese Instanzen löschen, um ein Wort w' zu erhalten, und dann die Regel $\xi \rightarrow w'$ in R' aufnehmen (falls sie dort noch nicht enthalten ist).

Man beachte, dass man nicht *alle* Zeichen aus E in w zu löschen hat, sondern nur eine Auswahl; die nicht ausgewählten Instanzen von Zeichen aus E belässt man unverändert. *Jede solche Auswahl* liefert eine Regel in R' , so dass eine R -Regel auf diese Weise mehrere R' -Regeln ins Leben rufen kann.

Wenn die obige Prozedur fertig ist, entfernen wir aus R' alle Regeln $\xi \rightarrow \varepsilon$ und erhalten so einen Regelsatz R'' , der keine ε -Regeln mehr enthält.

Da aber die Wirkung aller möglichen Anwendungen von ε -Regeln in R in den Regeln von R'' festgehalten wurde (bis auf Ableitungen, die das leere Worte produzieren), gilt für

$$G' := (\Psi, \Sigma, R'', \sigma)$$

tatsächlich, dass $L(G') = L(G) \setminus \{\varepsilon\}$ wie gewünscht.

Und G' ist ε -frei, da R'' keine ε -Regeln mehr enthält. ■

Beispiel 6.13 (fortgesetzt) Wir wenden den Algorithmus aus Satz 6.14 auf die Grammatik G_7 aus Beispiel 6.13 an.

Diese Grammatik enthält zwei ε -Regeln $A \rightarrow \varepsilon$ und $D \rightarrow \varepsilon$, so dass $E_1 = \{A, D\}$.

Die Regel $C \rightarrow D$ hat eine rechte Seite aus E_1^* , so dass

$$E_2 = E_1 \cup \{C\} = \{A, C, D\}.$$

Keine weitere Regel hat eine rechte Seite, in der nur die Buchstaben A , C und D vorkommen, so dass

$$E_3 = E_2 = E.$$

Aus dem Regelsatz von G_7 gewinnen wir einen neuen Regelsatz R' durch Modifizierung der Regeln aus R , indem wir einige der Zeichen aus E auf der rechten Seite löschen und andere nicht. *Jede Möglichkeit*, eine solche Auswahl zu löschender Zeichen zu treffen, liefert eine Regel in R' , so dass R' im allgemeinen mehr Regeln haben wird, als R . *Eine* solche Auswahl besteht darin, dass wir keine Zeichen löschen; deshalb bleibt jede Regel aus R auch in R' erhalten.

Wir erhalten in R' folgende Regeln, wobei die gesternteten neu hinzugekommen sind:

$$\begin{array}{ll} S \rightarrow AB & \\ S \rightarrow B & (*) \\ A \rightarrow a \mid \varepsilon & \\ B \rightarrow bC & \\ B \rightarrow b & (*) \\ C \rightarrow c \mid D & \\ C \rightarrow \varepsilon & (*) \\ D \rightarrow d \mid \varepsilon & \end{array}$$

Den endgültigen Regelsatz R'' der neuen Grammatik G_7' erhalten wir durch Entfernung aller ε -Regeln aus R' :

$$\begin{array}{l} S \rightarrow AB \\ S \rightarrow B \\ A \rightarrow a \\ B \rightarrow bC \\ B \rightarrow b \\ C \rightarrow c \mid D \\ D \rightarrow d \end{array}$$

Die Alphabete und das Startsymbol bleiben unverändert.

Hier haben wir alle Ableitungen zu terminalen Worten in G'_7 :

$$\begin{aligned} S &\rightarrow AB \rightarrow aB \rightarrow abC \rightarrow abc \\ S &\rightarrow AB \rightarrow aB \rightarrow abC \rightarrow abD \rightarrow abd \\ S &\rightarrow AB \rightarrow aB \rightarrow ab \\ S &\rightarrow B \rightarrow bC \rightarrow bc \\ S &\rightarrow B \rightarrow bC \rightarrow bD \rightarrow bd \\ S &\rightarrow B \rightarrow b \end{aligned}$$

Die Sprache $L(G'_7)$ besteht aus den „Endzuständen“ dieser Ableitungen, also aus den Worten abc , abd , ab , bc , bd , und b , die auch $L(G_7)$ ausmachen.

Hier noch ein Beispiel für die korrekte Entfernung von ε -Regeln aus einer Grammatik.

Beispiel 6.15 Wir betrachten die Grammatik $G_8 = (\Psi, \Sigma, R, \sigma)$ mit

$$\Psi = \{ S, A, B, C, D \}, \quad \Sigma = \{ a, b, c \} \quad \text{und} \quad \sigma = S$$

und mit den Regeln

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow a \mid b \\ B &\rightarrow CD \\ C &\rightarrow c \mid \varepsilon \\ D &\rightarrow \varepsilon \end{aligned}$$

Durch einfaches Hinsehen findet man schon eine Vereinfachung. Da D sich *nur* zu ε ableiten lässt, können wir D überall löschen und erhalten den einfacheren Regelsatz

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow a \mid b \\ B &\rightarrow C \\ C &\rightarrow c \mid \varepsilon, \end{aligned}$$

der die gleiche Sprache erzeugt. Die beiden letzten Regeln kann man kürzen zu $B \rightarrow c \mid \varepsilon$, und man sieht leicht, dass aus den ersten beiden und dieser Regel genau vier terminale Worte ableitbar sind, je nachdem, welche

Alternative man für A und unabhängig davon für B wählt:

$$\begin{aligned} S &\rightarrow AB \rightarrow ac \\ S &\rightarrow AB \rightarrow a\varepsilon = a \\ S &\rightarrow AB \rightarrow bc \\ S &\rightarrow AB \rightarrow b\varepsilon = b. \end{aligned}$$

Diese Worte bilden $L(G_8)$.

Wenn wir das Verfahren aus Satz 6.14 auf den ursprünglichen Regelsatz anwenden, erhalten wir die gleiche Sprache, aber die Begründung für das Ergebnis ändert sich in vielen Details.

Für die Menge der zum leeren Wort ableitbaren Zeichen finden wir

$$\begin{aligned} E_1 &= \{ C, D \} \\ E_2 &= E_1 \cup \{ B \} = \{ B, C, D \} && \text{wegen der Regel } B \rightarrow CD \in E_1^* \\ E_3 &= E_2 \end{aligned}$$

Also ist

$$E = E_2 = \{ B, C, D \}.$$

Aus dem Regelsatz von G_8 gewinnen wir einen neuen Regelsatz R' , indem aus den rechten Seiten der Regeln von R selektiv Zeichen aus E löschen. Das ergibt für R' folgende Regeln:

$$\begin{aligned} S &\rightarrow AB \mid A \\ A &\rightarrow a \mid b \\ B &\rightarrow CD \mid C \mid D \mid \varepsilon \\ C &\rightarrow c \mid \varepsilon \\ D &\rightarrow \varepsilon \end{aligned}$$

Den endgültigen Regelsatz R'' der neuen Grammatik G' erhalten wir durch Entfernung aller ε -Regeln aus R' :

$$\begin{aligned} S &\rightarrow AB \mid A \\ A &\rightarrow a \mid b \\ B &\rightarrow CD \mid C \mid D \\ C &\rightarrow c \end{aligned}$$

Die Alphabete und das Startsymbol bleiben unverändert.

Nach dem letzten Schritt gibt es keine Regel mehr mit D auf der linken Seite, so dass kein Wort, das D enthält, zu einem terminalen Wort abgeleitet werden kann.

Die Sprache ist also die gleiche, die von dem kleineren Regelsatz

$$\begin{aligned} S &\rightarrow AB \mid A \\ A &\rightarrow a \mid b \\ B &\rightarrow C \\ C &\rightarrow c \end{aligned}$$

erzeugt wird.

Man sieht leicht, dass sie, wie oben, nur aus den vier Worten ac , bc , a und b besteht.

Bei der Diskussion in Beispiel 6.15 haben wir einige neue Argumente verwendet, dahingehend, dass gewisse Zeichen nie zu einem terminalen Wort führen können oder dass manchmal zwei einfache „ineinander führende“ Regeln sich zu einer Einheit verschmelzen lassen.

Die hier benutzten Gesichtspunkte können allgemeiner formuliert und angewendet werden, um weitere Standardvereinfachungen in kontextfreien Grammatiken vorzunehmen, wie wir unten erläutern werden.

Wir beginnen mit einer einfachen Frage: Kann man an den Regeln einer Grammatik ablesen, ob die Grammatik überhaupt terminale Worte erzeugt? Wir haben oben gesehen, dass es durchaus nicht-terminale Zeichen geben kann, aus denen keine terminale Worte ableitbar sind. Das kann auch für das Startsymbol der Fall sein, wie das einfache Beispiel

$$G = (\{S, A, B\}, \{b\}, \{S \rightarrow A, B \rightarrow b\}, S)$$

zeigt.

Gibt es einen Algorithmus, der anhand des Regelsatzes entscheidet, ob die Sprache einer Grammatik leer ist oder nicht?

Die Antwort ist „ja“.

Satz 6.16 *Es gibt einen Algorithmus, der für jede kontextfreie Grammatik*

$$G := (\Psi, \Sigma, R, \sigma),$$

entscheidet, ob $L(G) = \emptyset$.

*(Man sagt dazu, die Frage ist **konstruktiv entscheidbar**.)*

Beweis. Wir stellen einen Algorithmus vor, der ähnlich ist zum ersten Schritt des Beweises von Satz 6.14, wo wir feststellen mussten, welche Zeichen aus dem nicht-terminalen Alphabet sich zum leeren Wort ableiten lassen.

Jetzt müssen wir stattdessen bestimmen, welche nicht-terminalen Zeichen zu einem terminalen Wort ableitbar sind, aber bis auf diese kleine Änderung sind die Beweisschritte die gleichen.

Wir konstruieren einen steigenden Turm von Zeichenmengen durch die rekursive Definition

$$\begin{aligned} N_1 &:= \Sigma \\ N_{i+1} &:= N_i \cup \{ \xi \in \Psi \mid \text{es gibt } w \in N_i^* \text{ mit } \xi \xrightarrow{R} w \}. \end{aligned}$$

Aus der Definition ist klar, dass für jedes i gilt

$$N_i \subseteq N_{i+1} \subseteq \Psi. \quad (6.9)$$

Die N_i sind durch Untersuchung des endlichen Regelsatzes effektiv bestimmbar.

Weil Ψ endlich ist, kann die linke Inklusion in (6.9) nicht immer eine echte Inklusion sein, sondern es muss nach spätestens $|\Psi|$ Schritten die Gleichheit herrschen.

Sei k das kleinste i , für das das passiert. Dann besteht N_k aus genau den nicht-terminalen Zeichen, die sich zu einem terminalen Wort ableiten lassen.

Die Sprache $L(G)$ ist nichtleer genau dann, wenn das Startsymbol σ sich zu einem terminalen Wort ableiten lässt, also genau dann, wenn $\sigma \in N_k$. Das ist das gesuchte Kriterium. ■

Auch wenn $L(G) \neq \emptyset$, kann es wie in der ε -freien Grammatik in Beispiel 6.15 vorkommen, dass einige nicht-terminale Zeichen überflüssig sind, weil sie sich nie an der Erzeugung terminaler Worte beteiligen. Aus dem gleichen Grund sind Zeichen überflüssig, die zwar zu terminalen Worten führen aber vom Startsymbol aus nicht erreichbar sind.

Jede konfliktfreie Grammatik lässt sich von solchen überflüssigen Anhängeln befreien:

Definition 6.17 Eine kontextfreie Grammatik

$$G := (\Psi, \Sigma, R, \sigma),$$

heißt **reduziert**, wenn folgende Bedingungen gelten:

- a) für jedes Zeichen $\xi \in \Psi$ gibt es ein Wort $w \in \Sigma^*$, so dass $\xi \xrightarrow{*} w$, und

- b) für jedes Zeichen $\gamma \in \Psi \cup \Sigma$ gibt es Worte u und $v \in (\Psi \cup \Sigma)^*$, so dass $\sigma \xrightarrow{*} u\gamma v$.

Wir führen etwas informale Terminologie ein, um die Bedeutung dieser Definition zu erläutern.

Wir nennen ein Zeichen $\xi \in \Psi$ **produktiv**, wenn es Bedingung a) erfüllt, nämlich wenn es ein Wort $w \in \Sigma^*$ gibt, so dass $\xi \xrightarrow{*} w$. *Produktive Zeichen sind zu einem terminalen Wort ableitbar.*

Wir nennen ein Zeichen $\gamma \in \Psi \cup \Sigma$ **erreichbar**, wenn es Bedingung b) erfüllt, nämlich wenn es Worte u und $v \in (\Psi \cup \Sigma)^*$ gibt, so dass $\sigma \xrightarrow{*} u\gamma v$. *Erreichbare Zeichen befinden sich in Worten, die aus dem Startsymbol ableitbar sind* (aber sie müssen nicht als Einzelzeichen aus σ ableitbar sein).

Eine Grammatik ist genau dann reduziert, wenn jedes nicht-terminale Zeichen produktiv und jedes nicht-terminale oder terminale Zeichen erreichbar ist.

Zeichen, die nicht diese Eigenschaften haben, sind überflüssig (zusammen mit den Regeln, in denen sie vorkommen), weil sie aus dem einen oder dem anderen Grund nie bei der Ableitung eines terminalen Wortes aus dem Startsymbol vorkommen können.

In der Tat kann man sie auch schadlos entfernen:

Satz 6.18 Sei

$$G := (\Psi, \Sigma, R, \sigma)$$

eine kontextfreie Grammatik mit $L(G) \neq \emptyset$.

Aus G ist eine reduzierte kontextfreie Grammatik G' mit

$$G \sim G'$$

effektiv konstruierbar.

Beweis. Um die Beweisstrategie besser erklären und begründen zu können, wollen wir die Bedingung b) aus Definition 6.17 a) ein wenig verallgemeinern.

Sei $\xi \in \Psi$ und $\gamma \in \Psi \cup \Sigma$. Wir nennen ξ einen **Vorfahren** von γ (und γ einen **Nachfahren** von ξ), wenn es Worte u und $v \in (\Psi \cup \Sigma)^*$ gibt, so dass $\xi \xrightarrow{*} u\gamma v$, und wir sagen dann auch, dass γ **aus ξ erreichbar** ist. (Ein Zeichen ist schlicht *erreichbar*, wenn es aus dem Startsymbol σ erreichbar ist).

Wir müssen beweisen, dass man jede kontextfreie Grammatik effektiv in eine reduzierte Grammatik umwandeln kann, ohne die erzeugte Sprache zu verändern.

Die Idee des Beweises besteht darin, aus der vorgegebenen Grammatik G zunächst alle *unproduktiven* nicht-terminalen Zeichen und die solche Zeichen enthaltenden Regeln zu entfernen, und anschließend die *unerreichbaren* Zeichen und ihre Regeln zu entfernen.

Der erste Schritt ändert die Sprache nicht, weil unproduktive Zeichen und Regeln, die sie enthalten, nie an der Erzeugung von terminalen Worten beteiligt sein können, und der zweite Schritt ändert die Sprache nicht, weil unerreichbare Zeichen und ihre Regeln nie in einer Ableitung beginnend beim Startsymbol vorkommen können.

Es ist allerdings wichtig, die genannte Reihenfolge der Schritte einzuhalten, d. h., zuerst die unproduktiven Zeichen zu entfernen und erst dann die unerreichbaren.

Um zu verstehen, warum das etwas ausmacht, müssen wir überlegen, welche Wirkung es hat, wenn wir aus der Grammatik eine Regel

$$\xi \xrightarrow{R} w$$

entfernen (was wir zu tun beabsichtigen, wenn das Wort w entweder unproduktive oder unerreichbare Zeichen enthält). Diese Regel kann nämlich an Ableitungen beteiligt sein, die darüber entscheiden, ob Vorfahren von ξ produktiv sind oder Nachfahren von ξ erreichbar sind.

Wenn wir die Regel entfernen, weil w unproduktive Zeichen enthält, dann kann sie aber nicht an einer Ableitung beteiligt sein, die zeigt, dass ein Vorfahr von ξ produktiv ist; insbesondere macht die Entfernung der Regel kein Zeichen unproduktiv, das vorher produktiv war.

Die Regel kann aber an einer Ableitung beteiligt sein, die zeigt, dass ein Nachfahr von ξ erreichbar ist. Das heißt, die Entfernung „unproduktiver“ Regeln kann dazu führen, dass weitere Zeichen überflüssig werden, weil sie nicht mehr erreichbar sind.

Wenn andererseits w unerreichbare Zeichen enthält, dann folgt aus der Definition von Erreichbarkeit, dass ξ und alle Vorfahren von ξ unerreichbar sein müssen. Die zu entfernende Regel kann also nicht an einer Ableitung beteiligt gewesen sein, die zeigt, dass ein Nachfahr von ξ erreichbar ist. Sie kann höchstens dazu beigetragen haben, dass ein Vorfahr von ξ produktiv ist. Diese Vorfahren sind aber *auch unerreichbar*, und deshalb werden die sie enthaltenden Regeln schon wegen *dieses* Kriteriums entfernt; es ist belanglos, ob sie unproduktiv werden oder nicht, weil sie ohnehin entfernt werden.

Das Fazit ist also: das Entfernen von unproduktiven Zeichen kann die anschließende Säuberung der Grammatik von unerreichbaren Zeichen modifizieren (und vollständiger machen) aber nicht umgekehrt! Würden wir die unerreichbaren Zeichen zuerst entfernen und dann die unproduktiven, könn-

ten nach diesem Schritt neue unerreichbare Zeichen entstanden sein und die Grammatik wäre doch nicht reduziert.

Um sicher zu gehen, dass nach der Reduzierung keine beim Verfahren *neu* entstandenen „unreduzierten“ Zustände übrig bleiben, müssen wir deshalb die unproduktiven Zeichen vor den unerreichbaren entfernen! In dieser Reihenfolge erzeugt der zweite Schritt nämlich *keine weiteren* unreduzierten Zeichen, die nur eine Wiederholung des ersten Schritts entfernen könnte.

Hier nun die Details des Algorithmus:

Schritt 1.

Wir konstruieren wie im Beweis von Satz 6.16 die Menge N der produktiven Zeichen. Wir setzen

$$N_1 := \Sigma$$

$$N_{i+1} := N_i \cup \left\{ \xi \in \Psi \mid \text{es gibt } w \in N_i^* \text{ mit } \xi \xrightarrow{R} w \right\}.$$

Die N_i bilden eine monoton wachsende Folge von Zeichenmengen, die aber wegen der Endlichkeit von Ψ nur endlich oft echt größer werden können. Die nach endlich vielen Rekursionen erreichte maximale Menge N_k ist gleich der Menge N aller produktiven Zeichen.

Wir ersetzen Ψ durch $\Psi' := N$ und entfernen aus dem Regelsatz alle Regeln, in denen Zeichen aus $\Psi \setminus N$ vorkommen; solche Regeln können nicht an der Erzeugung terminaler Worte beteiligt sein. Es bleibt über ein neuer, womöglich kleinerer Regelsatz R' .

Weil nach Voraussetzung $L(G) \neq \emptyset$, gehört σ zu N und wir haben durch diese Kürzung unser Startsymbol nicht verloren.

Schritt 2.

Wir konstruieren nach bewährter Manier die Menge A aller erreichbaren Zeichen. Wir setzen

$$A_1 := \{ \sigma \}$$

$$A_{i+1} := A_i \cup \left\{ \gamma \in \Psi' \cup \Sigma \mid \text{es gibt } \xi \in A_i \text{ und } u \text{ und } v \in (\Psi' \cup \Sigma)^* \right. \\ \left. \text{mit } \xi \rightarrow u\gamma v \in R' \right\}.$$

Die A_i bilden eine monoton wachsende Folge von Zeichenmengen, die aber wegen der Endlichkeit von $\Psi' \cup \Sigma$ nur endlich oft wachsen können. Die nach endlich vielen Rekursionen erreichte maximale Menge A_k ist gleich der Menge A aller erreichbarer Zeichen.

Wir ersetzen Ψ' durch $\Psi'' := \Psi' \cap A$ und Σ durch $\Sigma' := \Sigma \cap A$, und wir entfernen aus dem Regelsatz R' alle Regeln $\xi \rightarrow w$ mit $\xi \notin A$; solche

Regeln können nicht an Ableitungen aus σ beteiligt sein. Es bleibt über ein neuer, womöglich wieder kleinerer Regelsatz R'' .

Die Grammatik

$$G' := (\Psi'', \Sigma', R'', \sigma)$$

ist reduziert (weil wie oben erläutert keine unproduktiven oder unerreichbaren Zeichen übrig bleiben können) und sie erzeugt, auch aus Gründen die wir oben genannt haben, die gleiche Sprache, wie G . ■

Beispiel 6.19 Sei G die Grammatik

$$(\{S, A, B\}, \{a, b\}, \{S \rightarrow a \mid AB, B \rightarrow b\}, S).$$

Die nicht weiter ableitbaren Worte, die diese Grammatik aus S erzeugt, sind a und Ab , wie man schnell sieht, und nur das erste Wort a ist terminal. Also ist $L(G) = \{a\}$.

Wir berechnen aus G eine reduzierte Grammatik G' , die die gleiche Sprache erzeugt.

Erster Schritt: Die erste und letzte Regel haben terminale Worte auf der rechten Seite und wir finden $N_1 = \{S, B\}$. Keine weitere Regel hat rechts ein Wort aus terminalen Zeichen und den Buchstaben S und B , so dass

$$N_2 = N_1 = N = \Psi' = \{S, B\}.$$

Das Zeichen A wird also entfernt und wir entfernen auch alle Regeln, in denen A vorkommt (in diesem Fall: nur die zweite Regel). Es bleibt übrig

$$R' = \{S \rightarrow a, B \rightarrow b\}.$$

Zweiter Schritt: wir suchen die erreichbaren Zeichen. Wir haben

$$\begin{aligned} A_1 &:= \{S\} \\ A_2 &:= \{S\} \cup \{a\} = \{S, a\} \\ A_3 &:= A_2 = A. \end{aligned}$$

Wir entfernen die Zeichen B und b und die Regel $B \rightarrow b$ und übrig bleibt die reduzierte Grammatik

$$G' = (\{S\}, \{a\}, \{S \rightarrow a\}, S).$$

Offensichtlich ist $L(G') = \{a\}$.

Übrigens, wenn wir den zweiten Schritt vorgezogen hätten, hätten wir gefunden

$$\begin{aligned} A_1 &:= \{ S \} \\ A_2 &:= \{ S \} \cup \{ a, A, B \} = \{ S, a, A, B \} \\ A_3 &:= \{ S, a, A, B \} \cup \{ b \} = \{ S, a, A, B, b \} = \Psi \cup \Sigma. \end{aligned}$$

Also hätten wir bei diesem Schritt *keine* Zeichen und *keine* Regeln entfernt.

Nach dem ersten Schritt (als zweiten Schritt ausgeführt) wäre übrig geblieben die Sprache

$$\tilde{G} = (\{ S, B \}, \{ a, b \}, \{ S \rightarrow a, B \rightarrow b \}, S),$$

die nicht reduziert ist (weil B nicht erreichbar ist!)

Hier sieht man wieder, dass die im Algorithmus angegebene Schrittreihenfolge unbedingt eingehalten werden muss.

Die letzte Vereinfachung, die wir betrachten wollen, bringt alle Regeln in eine sehr eingeschränkte und deshalb leicht theoretisch zu handhabende Standardform.

Es ist klar, dass es Regeln geben muss, deren rechte Seiten aus mehr als einem Zeichen bestehen, aber die Normalform einer Grammatik, die wir einführen wollen, erlaubt dann nur eine Länge von zwei Zeichen, die obendrein nicht-terminal sein müssen! Trotzdem ist jede ε -freie Grammatik äquivalent zu einer so stark eingeschränkten Grammatik.

Definition 6.20 Eine kontextfreie Grammatik

$$G := (\Psi, \Sigma, R, \sigma)$$

heißt **Chomsky-normal** oder einfach **normal** genau dann, wenn für jede Regel

$$\xi \rightarrow w$$

aus R gilt, dass $w \in \Sigma \cup \Psi\Psi$.

In anderen Worten, in einer normalen Grammatik ist die rechte Seite jeder Regel entweder ein *einzelnes* terminales Zeichen, oder ein Wort aus *zwei* nicht-terminalen Zeichen (beide Zeichen im Wort müssen nicht-terminal sein).

Satz 6.21 Zu jeder reduzierten ε -freien (und somit kontextfreien) Grammatik

$$G := (\Psi, \Sigma, R, \sigma)$$

ist eine äquivalente normale Grammatik konstruierbar.

Beweis. Wir konstruieren aus G eine normale Grammatik in drei Schritten.

Beim ersten Schritt sorgen wir dafür, dass die rechte Seite jeder Regel höchstens die Länge 2 hat. Das machen wir, indem wir jede Regel

$$\xi \rightarrow x_1 x_2 \cdots x_n$$

mit einer rechten Seite der Länge $n > 2$ auf folgende Weise durch neue Regeln ersetzen, deren rechte Seiten die Länge 2 haben.

Dazu fügen wir zu Ψ neue Zeichen $\xi_1, \xi_2, \dots, \xi_{n-2}$ hinzu und ersetzen die genannte Regel durch den (offensichtlich äquivalenten) Satz von Regeln

$$\begin{aligned} \xi &\rightarrow x_1 \xi_1 \\ \xi_1 &\rightarrow x_2 \xi_2 \\ &\vdots \\ \xi_{n-3} &\rightarrow x_{n-2} \xi_{n-2} \\ \xi_{n-2} &\rightarrow x_{n-1} x_n. \end{aligned}$$

Wir behandeln alle Regeln mit zu langen rechten Seiten so, und erhalten auf diese Weise ein vergrößertes nicht-terminals Alphabet Ψ' und einen veränderten Regelsatz R' , in dem die rechte Seite jeder Regel Länge 1 oder 2 hat, aber vielleicht nicht die spezielle Form, die in einer normalen Grammatik für diese Längen vorgeschrieben ist.

Im zweiten Schritt berichtigen wir wenn erforderlich die Regeln mit rechten Seiten von Länge 2, damit beide Zeichen auf der rechten Seite nicht-terminal sind. Betrachten wir eine solche Regel

$$\xi \rightarrow x_1 x_2.$$

Wenn beide $x_i \in \Psi'$, ist keine Anpassung erforderlich. Aber wenn terminale Zeichen auf der rechten Seite erscheinen, dann fügen wir für jedes solche Zeichen x_i ein neues terminales Zeichen τ_i dem Alphabet Ψ' hinzu, wir ersetzen die terminalen Zeichen x_i in der genannten Regel durch das entsprechende τ_i , und wir fügen für jedes solche x_i eine neue Regel

$$\tau_i \rightarrow x_i$$

dem Regelsatz hinzu.

Die veränderte ursprüngliche Regel und die neu hinzugekommene Regel oder Regeln haben alle die Gestalt, die in einer normalen Grammatik verlangt wird. Die genannten Änderungen führen offensichtlich zu einer äquivalenten Grammatik, die die gleiche Sprache wie vorher erzeugt.

Nach diesen Änderungen haben wir einen vielleicht wieder größeren nicht-terminalen Zeichensatz Ψ'' und einen neuen Regelsatz R'' . Alle Regeln mit rechter Seite von Länge 2 haben jetzt das richtige Format.

Im letzten Schritt kümmern wir uns um die Regeln

$$\xi \rightarrow \omega \tag{6.10}$$

in denen die rechte Seite aus einem *einzelnen nicht-terminalen Zeichen* ω besteht.

Regeln von dieser Form sind nie hinzugefügt worden und waren deshalb schon in dem ursprünglichen Regelsatz vorhanden. Da die ursprüngliche Grammatik reduziert war, sind alle nicht-terminalen Zeichen auf der rechten Seite einer solchen Regel produktiv, d. h., sie lassen sich zu einem terminalen Wort ableiten. Aus diesem Grund muss es Regeln geben, die diese nicht-terminalen Zeichen auf der *linken* Seite haben.

Die Idee zur Entfernung der nicht normgetreuen Regeln mit einem einzelnen nicht-terminalen Zeichen auf der rechten Seite besteht darin, die Wirkung dieser Regel in neuen (normgerechten) Regeln zu erfassen, um anschließend die normverletzenden Regeln entfernen zu können. Das machen wir, indem wir Regeln der Gestalt (6.10) mit den anschließenden Regeln, die ω auf der linken Seite haben, verschmelzen.

Hier die genaue Definition der Erweiterung.

Wir setzen

$$R_0 := R''$$

$$R_{i+1} := R_i \cup \{ \xi \rightarrow w \mid \text{es gibt } \lambda \in \Psi'' \text{ mit } \xi \rightarrow \lambda \in R_i \text{ und } \lambda \rightarrow w \in R_i \}$$

Weil die neuerzeugten Regeln auf die angegebene Weise aus alten Regeln kombiniert werden, ist klar, dass sie keine Ableitungen ermöglichen, die nicht vorher schon möglich waren, d. h., sie erweitern die Sprache nicht.

Die neuen Regeln, die in dieser Konstruktion hinzukommen, haben linke Seiten und rechte Seiten, die auch bei alten Regeln vorkamen, aber in neuer Kombination. Das heißt, dass nur eine beschränkte Zahl von neuen Regeln erzeugt werden kann (höchstens m^2 Stück, wenn m die Anzahl der Regeln in R'' ist, weil m auch die Höchstzahl von linken Seiten und von rechten Seiten ist, die in einer neuen Regel kombiniert werden können).

Die R_i bilden wieder eine aufsteigende Folge

$$R_0 \subseteq R_1 \subseteq R_2 \subseteq \dots$$

von Mengen, und wegen der Beschränkung der Größe dieser Mengen wird diese Folge irgendwann konstant:

$$R_k = R_{k+1} = \dots$$

Jede Ableitung, die mit einer verbotenen Regel der Gestalt (6.10) beginnt, muss nach einer eventuellen Folge von weiteren verbotenen Regeln irgendwann mit einem einzelnen terminalen Zeichen x enden oder zu einer normgerechten rechten Seite $w \in \Psi''\Psi''$ führen, und die Konstruktion der R_i ist so, dass dann R_k sicher die Regel $\xi \rightarrow x$ oder $\xi \rightarrow w$ enthält und die normwidrigen Regeln in der Ableitung für diese Wirkung nicht mehr benötigt werden.

Aus diesem Grund wird die Sprache nicht verändert, wenn wir aus R_k die normwidrigen Regeln (6.10) weglassen.

Der letzte Schritt der Herleitung besteht darin, genau das zu tun. Das heißt, wir erhalten die endgültige normale Grammatik, indem wir zuerst R'' durch R_k ersetzen, daraus alle normverletzenden Regeln $\xi \rightarrow \omega$ entfernen (was zu einer äquivalenten Grammatik führt), und anschließend die resultierende Grammatik *reduzieren* (was jetzt vielleicht erforderlich ist, weil ja Regeln weggeworfen wurden).

Weil die Reduktion Regeln nicht verändert, sondern nur entfernt, werden dabei keine normverletzenden Regeln erzeugt. Die Grammatik bleibt jetzt normal, und ist weiterhin äquivalent zur Anfangsgrammatik G . ■

Wir führen das Normalisierungsverfahren für ein Beispiel aus.

Beispiel 6.22 Wir betrachten die Grammatik G mit dem Regelsatz R gegeben durch die Regeln

$$S \rightarrow AbCD$$

$$A \rightarrow eF \mid B$$

$$B \rightarrow G$$

$$C \rightarrow c$$

$$D \rightarrow d$$

$$F \rightarrow f \mid BC$$

$$G \rightarrow g \mid h$$

Hier ist das Startsymbol S , die Großbuchstaben in den Regeln bilden den nicht-terminalen Zeichensatz Ψ und die Kleinbuchstaben bilden das terminale Alphabet Σ .

Im ersten Schritt der Normalisierung müssen wir die erste Regel (deren rechte Seite Länge 4 hat) durch kürzere Regeln ersetzen. Wir benötigen zwei neue nicht-terminale Zeichen ξ_1 und ξ_2 , und wir *ersetzen* die Regel

$$S \rightarrow AbCD$$

durch die neuen Regeln

$$\begin{aligned} S &\rightarrow A\xi_1 \\ \xi_1 &\rightarrow b\xi_2 \\ \xi_2 &\rightarrow CD \end{aligned}$$

Jetzt gibt es keine Regeln mit überlangen rechten Seiten mehr, aber es gibt die Regeln

$$\begin{aligned} A &\rightarrow eF \\ \xi_1 &\rightarrow b\xi_2 \end{aligned}$$

mit rechten Seiten der Länge 2, die terminale Zeichen enthalten.

Um dies auszubessern, führen wir neue nicht-terminale Zeichen η und τ ein als Ersatz für das e in der ersten Regel und das b in der zweiten. Diese Regeln verändern sich zu dem Satz von Regeln

$$\begin{aligned} A &\rightarrow \eta F \\ \eta &\rightarrow e \\ \xi_1 &\rightarrow \tau\xi_2 \\ \tau &\rightarrow b \end{aligned}$$

Wir haben jetzt einen Gesamtregrsatz R'' bestehend aus den Regeln

$$\begin{aligned} S &\rightarrow A\xi_1 \\ \xi_1 &\rightarrow \tau\xi_2 \\ \tau &\rightarrow b \\ \xi_2 &\rightarrow CD \\ A &\rightarrow \eta F \mid B \\ \eta &\rightarrow e \\ B &\rightarrow G \\ C &\rightarrow c \\ D &\rightarrow d \\ F &\rightarrow f \mid BC \\ G &\rightarrow g \mid h \end{aligned}$$

Im letzten Teil müssen noch die Regeln korrigiert werden, die ein einzelnes nicht-terminales Zeichen auf der rechten Seite haben. Das sind die Regeln:

$$\begin{aligned} A &\rightarrow B \\ B &\rightarrow G \end{aligned}$$

Wir haben

$$\begin{aligned} R_0 &= R'' \\ R_1 &= R'' \cup \{ A \rightarrow G, B \rightarrow g, B \rightarrow h \} \\ R_2 &= R_1 \cup \{ A \rightarrow g, A \rightarrow h \} \\ R_3 &= R_2 \end{aligned}$$

Aus R_3 entfernen wir jetzt die Regeln $A \rightarrow B, B \rightarrow G, A \rightarrow G$ mit einem einzelnen nicht-terminalen Zeichen auf der rechten Seite. Der Gesamtregrsatz sieht dann so aus:

$$\begin{aligned} S &\rightarrow A\xi_1 \\ \xi_1 &\rightarrow \tau\xi_2 \\ \tau &\rightarrow b \\ \xi_2 &\rightarrow CD \\ A &\rightarrow \eta F \\ \eta &\rightarrow e \\ C &\rightarrow c \\ D &\rightarrow d \\ F &\rightarrow f \mid BC \\ G &\rightarrow g \mid h \\ B &\rightarrow g \mid h \\ A &\rightarrow g \mid h \end{aligned}$$

Diesen Regelsatz müssen wir noch reduzieren. Man sieht leicht, dass alle nicht-terminalen Zeichen produktiv sind, also zu terminalen Worten ableitbar sind.

Es ist also nur nötig, die unerreichbaren Zeichen zu bestimmen. Nach der Methode von Satz 6.18 erhalten wir

$$\begin{aligned} A_1 &= \{ S \} \\ A_2 &= \{ S, A, \xi_1 \} \\ A_3 &= \{ S, A, \xi_1, \eta, F, g, h, \tau, \xi_2 \} \\ A_4 &= \{ S, A, \xi_1, \eta, F, g, h, \tau, \xi_2, e, f, B, C, b, D \} \\ A_5 &= \{ S, A, \xi_1, \eta, F, g, h, \tau, \xi_2, e, f, B, C, b, D, c, d \} \\ A_6 &= A_5 \end{aligned}$$

und ein Vergleich mit dem Gesamtzeichensatz der Regeln zeigt, dass nur G unerreichbar ist. Wir müssen also nur dieses Zeichen und die einzige Regel, in der es vorkommt, entfernen.

Die normale Grammatik, die wir zum Schluß erhalten, hat

$$\begin{aligned}\Psi &= \{ S, A, \xi_1, \eta, F, \tau, \xi_2, B, C, D \} \\ \Sigma &= \{ g, h, e, f, b, c, d \}\end{aligned}$$

und die Regeln

$$\begin{aligned}S &\rightarrow A\xi_1 \\ \xi_1 &\rightarrow \tau\xi_2 \\ \tau &\rightarrow b \\ \xi_2 &\rightarrow CD \\ A &\rightarrow \eta F \\ \eta &\rightarrow e \\ C &\rightarrow c \\ D &\rightarrow d \\ F &\rightarrow f \mid BC \\ B &\rightarrow g \mid h \\ A &\rightarrow g \mid h\end{aligned}$$

Das Startsymbol ist natürlich S .

Kapitel 7

Turing Maschinen und Berechenbarkeit

Die wichtigsten Probleme der theoretischen Informatik sind die Fragen nach der algorithmischen **Berechenbarkeit** und der algorithmischen **Komplexität** einer vorgegebenen Aufgabe.

Was beide Fragen gemeinsam haben ist ihre Relevanz zur praktischen Anwendbarkeit eines Algorithmus oder, genau so wichtig für manche Zwecke, ihre Relevanz zur *praktischen* Lösbarkeit einer Aufgabe.

Letzteres ist zum Beispiel das entscheidende Kriterium für den praktischen Nutzen der modernen kryptographischen Verfahren, mit denen man elektronisch übermittelte Nachrichten vor dem Lesen durch Unbefugte schützen oder ihre Authentizität mittels einer elektronischen Unterschrift bekräftigen kann. Solche Verfahren werden im Internetkommerz und im Online-Banking eingesetzt, um nur zwei der wichtigsten Anwendungen zu nennen.

Das Knacken dieser Codes erfordert vom Angreifer die Fähigkeit, Produkte von sehr großen Primzahlen zu faktorisieren. Prinzipiell ist das für einen Rechner immer möglich, weil er ja alle Zahlen bis zu einer geeigneten Größe immer als Faktoren „ausprobieren“ kann, bis er einen wirklichen Faktor findet. Die Sicherheit des Verschlüsselungsverfahrens basiert darauf, dass diese (und jede andere bisher bekannte Methode) bei ausreichend großen Primfaktoren *zu langsam* ist, um praktikabel zu sein. Ein Verfahren, das einige Jahrtausende oder auch nur einige Jahre braucht, um ein Ergebnis zu liefern, ist eben nicht *praktisch* anwendbar und deshalb nicht nützlich.

Hier ist der entscheidende Faktor die **Komplexität** der Berechnung, womit im Wesentlichen nicht die Rechenzeit gemeint ist (die man durch den Einsatz eines schnelleren Rechners verkürzen kann), sondern *wie schnell die Rechenzeit mit der Größe der Daten wächst*.

Exponentielles Wachstum bietet einen billigeren Schutz als polynomiales

oder lineares Wachstum, weil man eine Steigerung der Rechengeschwindigkeit gängiger Rechner im ersten Fall durch eine kleine und vertretbare Vergrößerung der Datengröße wettmachen kann. Bei linearem Wachstum müsste man bei einer Vervierfachung der Rechengeschwindigkeit die Länge der Primfaktoren schon verdoppeln, und es ist eben nicht leicht, sehr große Primzahlen zu finden.

Die Komplexität einer Berechnung stellt nur ein relatives Hindernis zur praktischen Ausführbarkeit einer Berechnung dar. Überraschend ist die Tatsache, dass es Fragen gibt, die in jeder Situation eine eindeutige und wohlbestimmte Antwort haben, von der aber bekannt ist, dass sie prinzipiell von *keinem* Algorithmus und somit von *keinem* Computerprogramm berechnet werden kann.

Auf die Komplexitätsfrage können wir aus Zeitgründen leider überhaupt nicht weiter eingehen, aber wir wollen in der letzten Vorlesungsstunde wenigstens die andere, die Berechenbarkeitsfrage kurz erläutern und Beispiele für unberechenbare Aufgaben geben, mit einer Skizze des Unmöglichkeitsbeweises. Die Methoden hierzu wurden in der mathematischen Logik (also in der Theorie der Grundlagen der Mathematik) entwickelt und gehören zu den wichtigsten und gleichzeitig leider zu den enttäuschendsten mathematischen Ergebnissen des zwanzigsten Jahrhunderts.

Wenn man über die Berechenbarkeit von Aufgaben sprechen will, braucht man eine vernünftige und vor allem genaue Definition davon, was **Berechenbarkeit** eigentlich heißen soll. Im letzten Jahrhundert wurden mehrere solche Definitionen vorgeschlagen, manche eher algebraischer und kombinatorischer Natur, manche etwas direkter durch die Beschreibung eines „Standardrechners“. Zu den Beschreibungen der ersten Sorte gehören die **rekursiven Funktionen** und ein vom Logiker und Grundlagenforscher Alonzo Church vorgeschlagenes Formelkalkül, das so genannte **Lambdakalkül**; der wichtigste Vertreter der zweiten Sorte wird durch die von Alan Turing im Jahre 1936 beschriebenen **Turingmaschinen** gebildet.

Die unter dem Namen **Church's thesis** bekannt gewordene Behauptung, dass alle diese Definitionen richtig beschreiben, welche Aufgaben prinzipiell durch einen Algorithmus berechenbar sind, ist zwar nicht beweisbar, weil sie nicht genau genug formulierbar ist (wie sollen wir uns alle „möglichen“ Algorithmen oder maschinellen Berechnungen vorstellen können, wenn wir nicht wieder eine formal eingeschränkte Definition davon aufstellen, die dann nicht mehr alle „möglichen“ Auslegungen beinhaltet), aber es gibt zumindest viele Anhaltspunkte, die diese Behauptung sehr plausibel machen.

Zu diesen Anhaltspunkten gehören die Tatsache, dass man sehr wohl beweisen kann, dass die oben genannten und auch andere sehr verschiedenartige Definitionen von Berechenbarkeit alle genau die gleiche Klasse von berechen-

baren Funktionen festlegen, sowie die Tatsache, dass viele bekannte Beispiele von „offensichtlich“ berechenbaren Funktionen in allen oben genannten Modellen sich tatsächlich als berechenbar herausstellen.

Um über Berechenbarkeit sprechen zu können, müssen wir uns für eine der genannten möglichen Definitionen entscheiden, und wir wählen dafür auch aus Glaubwürdigkeits- und Verständlichkeitsgründen die direkteste Definition, das Modell der Turingmaschinen. Diese sehen so aus wie sehr einfache Computer (die allerdings nicht den praktischen Einschränkungen echter Computer unterliegen, wie die physikalische Begrenzung des Speicherraums und der CPU-Komplexität), und deshalb glaubt man gerne, dass sie auch tatsächliche Computer beschreiben können. Man muss nur davon überzeugt werden oder sich überzeugen lassen, dass sie trotz ihres sehr einfachen Aufbaus und ihrer sehr einfachen Arbeitsweise auch komplizierte Operationen ausführen können.

Es gibt mehrere Möglichkeiten, die Details einer Turingmaschine festzulegen, aber es ist sehr leicht einzusehen, dass sie zueinander äquivalent sind und deshalb die gleichen berechenbaren Funktionen bestimmen werden. Wir wählen eine Auslegung, die sich etwas knapper und schneller erklären lässt, als andere Varianten (dafür aber geringfügig längere Programme erfordert). Folgende Details gelten aber für alle Varianten.

Eine ***Turingmaschine*** ist eine gedachte Maschine wie in Abbildung 7.1 auf der nächsten Seite, die ausgestattet ist mit

- einem unendlich langen *Band*, eingeteilt in einzelne *Felder* oder *Karrees*, die entweder leer sein können oder Schriftzeichen tragen können; zu jedem Zeitpunkt sind nur endlich viele der Felder beschriftet;
- einem endlichen *Alphabet* von Schriftzeichen, die auf das Band geschrieben werden können;
- einem beweglichen *Schreib-Lese-Kopf*, der zu jedem Zeitpunkt auf genau einem Feld des Bandes steht und dieses Feld lesen oder beschreiben kann, oder sich um ein Feld nach rechts oder nach links bewegen kann;
- einer *Steuereinheit für den Schreib-Lese-Kopf* mit einer endlichen Anzahl von *internen Zuständen* und mit
- einem *Programm aus Befehlen*, die in Abhängigkeit vom gelesenen Zeichen und dem internen Zustand der Steuereinheit den Schreibkopf anweisen, ein bestimmtes Zeichen zu schreiben oder sich zu bewegen, und die Steuereinheit anweisen, in einen neuen internen Zustand überzugehen.

Übrigens, Bewegung ist immer ein Relativbegriff. Würde man eine Turingmaschine physikalisch realisieren, würde sich vermutlich das Band unter dem Schreib-Lese-Kopf bewegen, und nicht umgekehrt, aber die Sichtweise bei der Beschreibung einer Turingmaschine ist immer die, dass der *Kopf* sich nach rechts oder nach links über das Band bewegt.

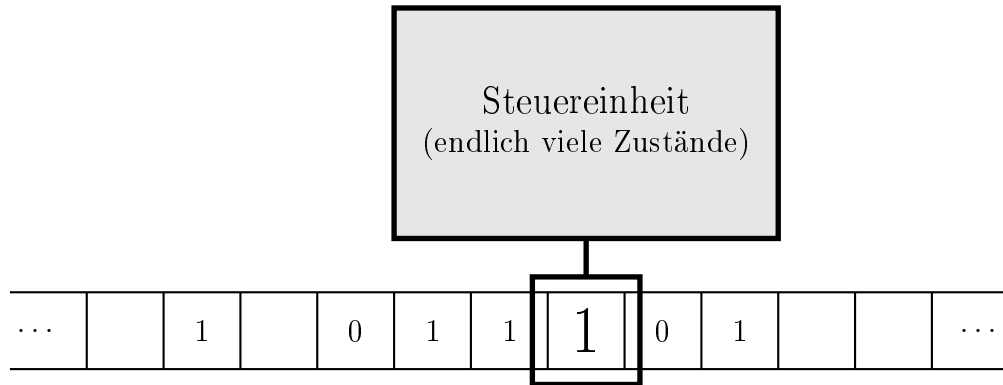


Abbildung 7.1: Eine Turingmaschine

Hier eine etwas formale aber genauere Definition:

Definition 7.1 Eine *Turingmaschine* ist ein Tripel

$$M = (\Sigma, Q, P)$$

bestehend aus einer endlichen Menge Σ von **Zeichen** (Σ heißt das **Alphabet** der Turingmaschine), einer endlichen Menge Q von **Zuständen** q_0, q_1, \dots, q_n , und einer endlichen Menge P , genannt das **Programm**, von **Befehlen**, deren Aufbau wir gleich beschreiben.

Die genannten Merkmale definieren eine bestimmte Turingmaschine, aber sie sagen noch nicht alles über sie aus, denn die Turingmaschine führt Schritt um Schritt auf einem gedachten **Band** beschriftet mit Symbolen aus Σ gewisse Operationen aus und verändert dabei den Zustand des Bandes.

Dabei durchläuft sie gewisse **Konfigurationen**, die wir uns so vorstellen, wie in der informalen Beschreibung oben, nämlich dass in einer bestimmten Konfiguration gewisse Zeichen aus dem Alphabet auf dem Band stehen, dazwischen sind eventuell leere Felder, und der Schreibkopf ist über eines dieser Zeichen oder ein leeres Feld positioniert und die Turingmaschine befindet sich in einem der verfügbaren Zuständen aus Q .

Formal definieren wir eine **Konfiguration der Turingmaschine** M als ein Ausdruck

$$\alpha q \beta,$$

wo α und β endliche (eventuell leere) Folgen von Zeichen aus Σ oder dem nicht zu Σ gehörenden Sonderzeichen \square sind (welches ein leeres Bandfeld darstellt) und wo $q \in Q$. Das Wort α darf nicht mit dem Zeichen \square beginnen, und das Wort β darf nicht mit dem Zeichen \square enden.

Dieser Ausdruck soll die Situation darstellen, dass die Turingmaschine sich in Zustand q befindet, dass auf dem Band die Zeichenfolge $\alpha\beta$ steht, und rechts und links davon alle Felder leer sind, und dass der Schreibkopf auf dem ersten Zeichen von β steht (oder auf einem leeren Feld, wenn β das leere Wort ist).

Jetzt können wir sagen, wie die Befehle im Programm aussehen. Jeder Befehl ist ein Quadrupel

$$q_i \ S \ A \ q_j,$$

wo q_i und q_j Zustände aus Q sind, S ein Zeichen aus Σ oder \square ist, und A entweder \square oder ein Zeichen aus Σ ist, oder eines der Buchstaben R oder L.

Die Bedeutung des Befehls in der informalen Beschreibung ist folgende Anweisung an die Turingmaschine: „Wenn du dich in Zustand q_i befindest und dein Lesekopf das Zeichen S liest, dann mache folgendes: wenn A ein Zeichen ist, schreibe dieses Zeichen auf das Band an der jetzigen Stelle; wenn A der Buchstabe R ist, bewege den Schreib-Lese-Kopf ein Feld nach rechts; wenn A der Buchstabe L ist, bewege den Schreib-Lese-Kopf ein Feld nach links; gehe anschließend in Zustand q_j über.“

Dadurch entsteht eine neue momentane Konfiguration, und es ist anhand der informalen Beschreibung leicht zu verstehen, wie sie aussehen wird. Das kann man ohne Mühe auch formal beschreiben, aber wir verzichten darauf, weil kein Mensch mit den formalen Beschreibungen denkt. Für das Verständnis ist die informale Beschreibung viel bequemer, und dabei bleiben wir.

Das Programm enthält eine *endliche* Anzahl solcher Befehle, wobei keine zwei Befehle die gleichen q_i und S haben (damit in jeder Konfiguration höchstens ein Befehl anwendbar ist und die Maschine eine *eindeutige* Operation durchführt).

Damit haben wir eine einzelne Operation der Turingmaschine beschrieben, aber nicht ihr Gesamtablauf. Dieser sieht wie folgt aus.

Die Turingmaschine beginnt nach Konvention in Zustand q_0 und es befindet sich eine Anfangskonfiguration auf dem Band, also eine beliebige endliche Zeichenfolge und die Festlegung, wo der Schreibkopf positioniert ist (wir werden das aber gleich ein bisschen einschränken)

Die Turingmaschine beginnt dann, ihr Programm zu durchsuchen, bis sie ein Befehl findet, bei dem q_i ihr momentaner Zustand ist und S das vom Schreib-Lesekopf gerade gelesene Zeichen ist. Wird ein solcher Befehl gefunden, dann ist er eindeutig; er wird wie oben beschrieben ausgeführt und

ausgehend von der dann entstehenden neuen Konfiguration wird die Suche nach einem ausführbaren Befehl wiederholt.

Wenn kein ausführbarer Befehl gefunden wird, bleibt die Turingmaschine stehen.

Zunächst ein paar Bemerkungen über die Details dieser Definition. Es gibt viele Varianten dieses Aufbaus, aber die meisten beinhalten keine wesentliche Abweichung. Man kann zum Beispiel den Zeichensatz auf nur zwei Zeichen 0 und 1 (oder sogar auf nur ein Zeichen 1 und das Leerzeichen \square) einschränken, aber die so eingeschränkten Maschinen können genau die gleichen Berechnungen durchführen wie unsere Maschine, denn unserer größerer Zeichensatz kann als Bitfolgen von Nullen und Einsen kodiert werden, und mit zusätzlichen Zuständen kann die eingeschränkte Maschine diese Bitfolgen eindeutig lesen, entziffern und schreiben und auf diese Weise unsere bequemere Maschine genau nachmachen.

Manchmal werden Turingmaschinenbefehle als Quintupel aufgefasst, bei denen an den vorletzten Stellen *sowohl* ein Zeichen wie auch eine Bewegungsrichtung R, L oder N angegeben wird. Bei diesen Befehlen wird in einer einzigen Operation ein Zeichen auf das Band geschrieben (eventuell das Zeichen, das schon da stand) und der Kopf bewegt (eventuell auch nicht, wenn die Operation N heißt). Das macht die Maschine aber nicht leistungsfähiger als unsere; unsere kann das Gleiche, aber braucht zwei Befehle dafür.

Unsere Maschine hat einen eindeutigen Anfangszustand q_0 aber keinen eindeutig bestimmten Endzustand. Die Maschine stoppt einfach, wenn sie keinen passenden Befehl findet. Oft wird verlangt, dass eine Turingmaschine in einem *eindeutig festgelegten* Endzustand q_e stoppen muss, wenn sie stehen bleibt. Das ist leicht einzurichten mit zusätzlichen Programmbefehlen, die unsere Endkonfigurationen in Zustand q_e überführen, ohne sonst etwas zu verändern. Für Zustand q_e sind keine Befehle anzugeben.

Wir bleiben aber bei der Version, die wir oben beschrieben haben.

Wir wollen mit Hilfe von Turing Maschinen beschreiben und untersuchen, was eine **berechenbare Funktion** ist. Dazu müssen wir doch etwas dazu sagen, wie eine Turingmaschine eine zahlenwertige Funktion berechnen kann. Wir erlauben sogar Funktionen von mehreren Variablen, die auch mehrere Variablen als Wert ausgeben können, und treffen folgende Vereinbarung:

Konvention 7.2 Eine natürliche Zahl n stellen wir auf einem Turingmaschinenband durch eine Folge von $n + 1$ Einsen dar.

Ein k -Tupel (n_1, n_2, \dots, n_k) von natürlichen Zahlen stellen wir dar durch k Einserblöcke, für jedes i mit $n_i + 1$ Einsen im i -ten Block, wobei jeder Block vom folgenden durch ein einzelnes Feld mit dem Zeichen 0 getrennt wird.

Das Zahlentripel $(3, 0, 1)$ wird zum Beispiel durch die Bandbeschriftung 111101011 dargestellt (alle anderen Felder sind leer).

Wir sagen, dass eine Turingmaschine M eine partiell definierte Funktion

$$f: D \subseteq \mathbb{N}^k \longrightarrow \mathbb{N}^r \quad (7.1)$$

berechnet, wenn gilt:

Wenn man für ein k -Tupel (n_1, n_2, \dots, n_k) aus dem Definitionsbereich D von f die Turingmaschine startet mit diesem k -Tupel auf dem Band und dem Schreibkopf positioniert auf der rechtensten Eins des rechtensten Feldes, dann läuft die Turingmaschine eine Weile und stoppt irgendwann mit dem r -Tupel $f(n_1, \dots, n_k)$ als einzige nichtleere Daten auf dem Band und mit dem Schreibkopf positioniert am rechten Ende des rechtensten Einserfeldes; wenn man die Turingmaschine in einer Bandkonfiguration startet, die nicht ein Zahlentupel aus D darstellt, dann stoppt die Maschine nie.

Eine Funktion wie in (7.1) heißt **berechenbar**, wenn es eine Turingmaschine gibt, die sie berechnet.

Ich möchte Sie jetzt überzeugen, dass Turingmaschinen sehr komplizierte Berechnungen durchführen können, aber die Zeit erlaubt nicht, dass ich Ihnen wirklich Beispiele für komplizierte Berechnungen vorführe. Stattdessen werde ich eine Reihe von Beispielen von sehr einfachen aber sehr nützlichen Operationen angeben, und damit motivieren, dass eine Turingmaschine auch wesentlich kompliziertere Sachen machen kann.

Hier nun Beispielmachines, die einige wichtige Grundoperationen ausführen:

Beispiele 7.3 In diesen Beispielen besteht das Alphabet der Turingmaschine aus allen Zeichen, die erwähnt oder verwendet werden, und die Zustandsmenge besteht aus allen Zuständen, die im Programm vorkommen. Wir geben nur das Programm explizit an, weil die anderen Daten sich daraus ablesen lassen.

Außerdem weichen wir in einem Punkt von der obigen Beschreibung ab, um die Programme ein bisschen einfacher zu machen. Wenn die Beispielmachine eine bestimmte Datenart, etwa eine natürliche Zahl, als Eingabe erwartet, dann geht sie davon aus, dass tatsächlich Daten der gewünschten Form auf dem Band stehen (und bei Zahlentupeln, dass der Schreibkopf am rechten Ende der Daten steht). Sie prüft nicht nach, ob diese Vorgaben wirklich eingehalten wurden und verhält sich deshalb auch nicht normgerecht, wenn ihr „ungültige“ Daten präsentiert werden (d.h., sie geht dann nicht unbedingt in eine Endlosschleife).

- a) Diese Turingmaschine erwartet eine einzelne natürliche Zahl n auf dem Band, und erhöht sie um 1, berechnet also $n + 1$:

q_0	1	R	q_1
q_1	\square	1	q_1

Erläuterung: Die Maschine startet in Zustand q_0 und liest die rechte 1 der Zahl n . Der erste Befehl bewegt den Kopf um eine Stelle nach rechts und versetzt die Maschine in Zustand q_1 . Weil das neue Feld, dass der Lesekopf jetzt sieht, leer ist, kann der zweite Befehl ausgeführt werden, der eine 1 auf das Band schreibt aber den Zustand nicht verändert. Für Zustand q_1 gibt es aber keine Befehle, die ausführbar sind, wenn eine 1 gelesen wird; also stoppt die Maschine.

- b) Diese Turingmaschine erwartet eine einzelne natürliche Zahl n auf dem Band, und vermindert sie um 1, *wenn das innerhalb der natürlichen Zahlen möglich ist!* Sonst wird die Zahl nicht verändert.

Diese Maschine berechnet also die Funktion

$$f(n) := \begin{cases} 0, & \text{wenn } n = 0; \\ n - 1, & \text{wenn } n > 0. \end{cases}$$

Das Programm lautet

q_0	1	\square	q_0
q_0	\square	L	q_1
q_1	\square	1	q_1

Erläuterung: Die Maschine startet in Zustand q_0 und liest die rechte 1 der Zahl n . Der erste Befehl löscht diese 1 aber verändert den Zustand nicht. Der zweite Befehl ist dann anwendbar (weil jetzt eine leere Zelle gelesen wird) und bewegt den Kopf um eine Stelle nach links und versetzt die Maschine in Zustand q_1 .

Wenn das neue Feld, dass der Lesekopf jetzt sieht, leer ist, war die ursprüngliche Zahl $n = 0$ und durfte nicht vermindert werden. In diesem Fall ist der dritte Befehl anwendbar und schreibt wieder eine 1 auf das Band, um das Löschen rückgängig zu machen (es schadet nichts, dass die 1 nicht in die gleiche Zelle geschrieben wird, in der sie ursprünglich stand).

Sobald die Maschine aber in Zustand q_1 eine 1 liest, bleibt sie stehen, weil es keine zu dieser Situation passenden Befehle im Programm gibt.

- c) Diese Turingmaschine erwartet ein Paar (m, n) von natürlichen Zahlen auf dem Band, und berechnet ihre Summe $m + n$. Wir nummerieren die Befehle am rechten Rand.

$$q_0 \quad 1 \quad L \quad q_0 \quad (1)$$

$$q_0 \quad 0 \quad 1 \quad q_1 \quad (2)$$

$$q_1 \quad 1 \quad R \quad q_1 \quad (3)$$

$$q_1 \quad \square \quad L \quad q_2 \quad (4)$$

$$q_2 \quad 1 \quad \square \quad q_2 \quad (5)$$

$$q_2 \quad \square \quad L \quad q_3 \quad (6)$$

$$q_3 \quad 1 \quad \square \quad q_3 \quad (7)$$

$$q_3 \quad \square \quad L \quad q_4 \quad (8)$$

Erläuterung: Die Maschine startet in Zustand q_0 und liest die rechte 1 der Zahl n . Der erste Befehl bewegt den Kopf um eine Stelle nach links und ändert den Zustand nicht. Deshalb bleibt dieser Befehl immer wieder ausführbar und bewegt den Kopf immer weiter nach links, solange Einsen auf dem Band gelesen werden. Dieser Befehl sucht also die mit 0 beschriebene Trennstelle zwischen den Daten m und n .

Wenn diese Stelle erreicht wird, befindet sich die Maschine immer noch in Zustand q_0 und liest jetzt eine 0. Die beiden Zahlen werden mit Befehl 2 jetzt zu einer Zahl verschmolzen, indem die 0 mit einer 1 überschrieben wird; die Maschine geht über in Zustand q_1 . In diesem Moment steht nur noch eine Zahl auf dem Band, aber sie ist um zwei zu groß, denn wir wollen $m + n + 1$ Einsen auf dem Band haben, aber es gab zu Anfang schon $m + 1 + n + 1 = m + n + 2$ Einsen, und das Programm hat eine weitere 1 hinzugefügt!

Die Maschine bleibt in Zustand q_1 und bewegt mit Befehl 3 den Kopf nach rechts, solange Einsen gelesen wird. Sie sucht die leere Zelle, die die Zahl rechts begrenzt.

Sobald in Zustand q_1 die leere Zelle gelesen wird, bewegt sich der Kopf mit dem vierten Befehl nach links und geht in einen neuen Zustand q_2 über. Jetzt müssen zwei Einsen gelöscht werden.

Die beiden Befehle 5 und 6 für Zustand q_2 löschen die letzte 1 und bewegen den Kopf eine Stelle nach links; der Zustand wird zu q_3 . Der letzten beiden Befehle löschen auf die gleiche Weise die vorletzte 1 und bewegen den Kopf um eine weitere Stelle nach links; der Zustand wird zu q_4 . Jetzt sieht der Kopf die rechteste überlebende 1. Da es keine Befehle für Zustand q_4 gibt, stoppt die Maschine.

Hier die Folge von Konfigurationen für die Addition $3 + 2 = 5$:

Konfiguration	ausführbarer Befehl
1111011 q_0 1	1
111101 q_0 11	1
11110 q_0 111	1
1111 q_0 0111	2
1111 q_1 1111	3
11111 q_1 111	3
111111 q_1 11	3
1111111 q_1 1	3
11111111 q_1	4
11111111 q_2 1	5
11111111 q_2	6
1111111 q_3 1	7
1111111 q_3	8
11111 q_4 1	keiner

- d) Diese Turingmaschine verschiebt alle Daten auf dem Band, beginnend mit der gegenwärtigen Kopfposition bis zum ersten Leerfeld links (aber nicht einschließlich dieses Feldes), um eine Stelle nach links, und löscht das Feld, auf das der Kopf am Anfang stand. Die Maschine stoppt mit dem Kopf auf dem rechten Zeichen der bewegten Daten.

Das Alphabet Σ enthalte n Zeichen, die wir S_1, \dots, S_n nennen wollen. Die Maschine benötigt $2n+3$ Zustände q_0, \dots, q_{2n+2} , und ihr Programm enthält folgende Befehle:

- i) für jedes i mit $1 \leq i \leq n$ ein Befehl $q_0 S_i \sqcap q_i$
- ii) für jedes i mit $1 \leq i \leq n$ ein Befehl $q_i \sqcap L q_{n+i}$
- iii) für jedes i mit $1 \leq i \leq n$ ein Befehl $q_i S_i L q_{n+i}$
- iv) für jedes i mit $1 \leq i \leq n$ und für jedes j mit $1 \leq j \leq n$ ein Befehl $q_{n+i} S_j S_i q_j$
- v) für jedes i mit $1 \leq i \leq n$ ein Befehl $q_{n+i} \sqcap S_i q_{2n+1}$
- vi) für jedes i mit $1 \leq i \leq n$ ein Befehl $q_{2n+1} S_i R q_{2n+1}$
- vii) ein Befehl $q_{2n+1} \sqcap L q_{2n+2}$

Erläuterung: Die Befehle in Gruppe i) löschen das Anfangsfeld aber merken sich im neuen Zustand q_i , welches Zeichen dort stand.

Dann kann der entsprechende Befehl in Gruppe ii) ausgeführt werden, der den Kopf nach links bewegt und einen *neuen* Zustand einstellt, der aber auch dem zu bewegenden Zeichen zugeordnet ist und dieses Zeichen in Erinnerung behält. Ein neuer Zustand ist hier erforderlich, damit das leere Feld am Ende der Daten anders behandelt werden kann, als das am Anfang gelöschte Feld beim Lesen nach dem Löschen.

Im neuen Zustand ist ein Befehl aus Gruppe iv) ausführbar, falls das neu unter dem Kopf erscheinende Feld ein Zeichen aus dem Alphabet trägt. Dieser Befehl schreibt das in Erinnerung gebliebene vorher rechts gestandene Zeichen in das Feld, merkt sich aber wieder durch Übergang in einen der Zustände q_j , welches Zeichen vorher auf dem jetzigen Feld stand und jetzt überschrieben wird.

Jetzt ist ein Befehl aus Gruppe iii) ausführbar und der Schreib-Lese-Kopf bewegt sich nach links zum nächsten Feld.

Beim ersten Lesen jedes neuen Feldes ist die Turingmaschine in einen der Zustände q_{n+i} . Steht auf dem neuen Feld ein Zeichen, werden die oben beschriebenen Operationen noch einmal ausgeführt, aber wenn das neue Feld leer ist, dann muss nur noch das richtige von rechts kommende Zeichen in das Feld geschrieben werden; die Datenbewegung ist dann beendet. Das Beschreiben dieses letzten Feldes erledigt ein Befehl aus Gruppe v), der die Maschine in den vorletzten Zustand q_{2n+1} versetzt.

In diesem Zustand werden Befehle aus Gruppe vi) ausgeführt und der Kopf nach rechts bewegt, solange Zeichen aus dem Alphabet gelesen werden. Wenn das anfangs gelöschte leere Feld erreicht wird, wird der Kopf mit dem letzten Programmbefehl nach links gerückt und die Maschine stoppt in Zustand q_{2n+2} , für den keine Befehle existieren.

- e) Diese Turingmaschine erwartet ein Tupel von natürlichen Zahlen auf dem Band und links von diesem Tupel, getrennt von dem Tupel durch mindestens ein leeres Feld, eine Markierung *. Andere Zeichen als 0, 1, * und leere Felder befinden sich nicht auf dem Band.

Es wird eine *Kopie* des Zahlentupels auf das Band geschrieben, wobei die rechteste Eins der Kopie das markierte Feld überschreibt. Andere Zeichen als 0, 1, * und leere Felder befinden sich (zu Beginn) nicht auf dem Band.

Anschließend bewegt sich der Kopf an den Anfang der ursprünglichen Daten (mit leichten Abänderungen des Programms könnte man ihn auch an den Anfang der Kopie bewegen).

Wir benutzen zusätzliche Zeichen, um den Arbeitsablauf zu markieren.
Das Programm lautet

q_0	1	A	q_0	q_0	0	B	q_0
q_0	A	L	q_1	q_0	B	L	q_6
q_0	\square	L	q_9	q_1	0	L	q_1
q_1	1	L	q_1	q_1	\square	L	q_1
q_1	*	A	q_2	q_1	A	1	q_3
q_1	B	0	q_3	q_2	A	R	q_5
q_2	B	R	q_5	q_3	0	L	q_4
q_3	1	L	q_4	q_4	\square	A	q_2
q_4	0	A	q_2	q_4	1	A	q_2
q_4	*	A	q_2	q_5	\square	R	q_5
q_5	0	R	q_5	q_5	1	R	q_5
q_5	A	L	q_0	q_5	B	L	q_0
q_6	0	L	q_6	q_6	1	L	q_6
q_6	\square	L	q_6	q_6	A	1	q_7
q_6	B	0	q_7	q_7	0	L	q_8
q_7	1	L	q_8	q_8	\square	B	q_2
q_8	0	B	q_2	q_8	1	B	q_2
q_8	*	B	q_2	q_9	0	L	q_9
q_9	1	L	q_9	q_9	\square	L	q_9
q_9	A	1	q_{10}	q_9	B	0	q_{10}
q_{10}	\square	R	q_{10}	q_{10}	0	R	q_{10}
q_{10}	1	R	q_{10}	q_{10}	A	A	q_{11}
q_{10}	B	B	q_{11}	q_{11}	A	1	q_{12}
q_{11}	B	0	q_{12}	q_{11}	0	L	q_2
q_{11}	1	L	q_2	q_{11}	\square	L	q_2
q_{12}	0	R	q_{11}	q_{12}	1	R	q_{11}

Aus Platzgründen stehen zwei Befehle auf jeder Zeile, und die Befehle sind nach Eingangszuständen sortiert.

Wir beschreiben in groben Zügen, was das Programm macht. Die ersten vier Befehle ersetzen im ursprünglichen Zahlentupel Einsen durch A und Nullen durch B , und übergeben die Kontrolle an die q_1 -Befehle oder q_6 -Befehle, die nach links den Rest des Zahlentupels und die Leerstellen überspringen, bis sie in der Kopie entweder die Markierung $*$ finden oder ein A oder ein B , das die letzte kopierte Stelle kennzeichnet.

Der Stern kann nur beim ersten Mal gefunden werden und er wird mit A überschrieben. Wenn ein vorher kopiertes A oder B gefunden wird, wird es mit dem richtigen Zahlenwert 0 oder 1 überschrieben und

links davon wird die jetzt zu kopierende Zahl in der Form A oder B geschrieben, um die neue letzkopierte Stelle zu markieren.

In allen Fällen geht die Maschine dann in Zustand q_5 über, für den es Befehle gibt, die den Kopf wieder nach rechts bewegen, bis er auf die zuletzt kopierte Originalstelle stößt (gekennzeichnet durch ein A oder ein B). Der Kopf wird dann ein Feld nach links bewegt und dieses Verfahren wiederholt sich, wieder beginnend in Zustand q_0 .

Die Wiederholungen werden abgebrochen, wenn in Zustand q_0 eine Leerstelle gelesen wird, die das Ende des zu kopierenden Tupels markiert. Anschließend ist die Maschine in Zustand q_9 und der Kopf wird nach links bewegt, bis das zuletzt geschriebene A oder B am linken Ende der Kopie gefunden wird. Dieser Buchstabe wird auf den entsprechenden Zahlenwert korrigiert.

In Zustand q_{10} läuft der Kopf nach rechts, bis die A s und B s der als kopiert markierten ursprünglichen Daten gefunden werden. Diese werden in den ursprünglichen Zahlenwert zurückverwandelt, bis in Zustand q_{11} keine A s und B s mehr gefunden werden.

Der Kopf wird dann eine Stelle nach links bewegt auf das rechte Ende des ursprünglichen Tupels. Der Kopf liest dort mit Sicherheit eine 1, aber die Maschine ist in Zustand q_2 versetzt worden, für den es keine zu einer gelesenen 1 passenden Befehle gibt. Also stoppt die Maschine in der gewünschten Stellung.

- f) Diese Turingmaschine erwartet ein Paar von natürlichen Zahlen auf dem Band, getrennt durch ein Nullfeld. Links und rechts vom Paar stehen Leerfelder.

Die Maschine vergleicht die beiden Zahlen und hinterlässt die natürliche Zahl 0, wenn sie verschieden sind, und 1, wenn sie gleich sind.

Das Programm finden Sie in Tabelle 7.1 auf der nächsten Seite

Die Methode dieses Programms besteht darin, die Einsen abwechselnd an beiden Enden des Zahlenpaares zu löschen. Dazu wird der Kopf hin- und hergefahren, und das Ende der Daten wird daran erkannt, dass der Kopf ein leeres Feld liest. Er wird dann um ein Feld zurückgefahren, in die Richtung, aus der er gekommen ist. Wenn dort eine 1 steht, wird sie gelöscht, aber wenn dort die trennende 0 steht, gibt es auf einer Seite der 0 keine Einsen mehr zum Löschen.

Jetzt kann geprüft werden, ob die Anfangszahlen gleich waren. Wenn sie es waren, kann immer rechts von der 0 und dann links von der 0

q_0	1	\square	q_1
q_0	0	L	q_6
q_1	\square	L	q_2
q_2	1	L	q_2
q_2	0	L	q_2
q_2	\square	R	q_3
q_3	0	R	q_{10}
q_3	1	\square	q_4
q_4	\square	R	q_5
q_5	1	R	q_5
q_5	0	R	q_5
q_5	\square	L	q_0
q_6	\square	1	q_7
q_6	1	\square	q_8
q_7	\square	R	q_7
q_7	1	R	q_7
q_7	0	1	q_{12}
q_8	\square	L	q_9
q_9	1	\square	q_8
q_9	\square	R	q_7
q_{10}	0	1	q_{12}
q_{10}	1	\square	q_{11}
q_{10}	\square	L	q_{10}
q_{11}	\square	R	q_{10}

Tabelle 7.1: Ein Turingmaschinenprogramm zum Vergleich zweier Zahlen

eine Eins gelöscht werden, bis irgendwann festgestellt wird, dass *rechts* von der 0 alle Einsen weg sind. Wenn die Einsen auf der linken Seite zuerst ausgehen, waren die Anfangszahlen ungleich (weil rechts eine Eins gelöscht werden kann, die links keinen Partner hat). Wenn die Einsen rechts zuerst ausgehen, ist nur noch zu prüfen, dass links auch keine mehr vorhanden sind; nur in diesem Fall waren die Anfangszahlen wirklich gleich.

Nach dieser Prüfung werden eventuell noch verbleibende Einsen gelöscht und die trennende 0 wird zu einer 1 gemacht und steht am rechten Ende des Ergebnisses. Auf ihrer linken Seite steht eine zweite 1 genau dann, wenn die ursprünglichen Zahlen gleich waren. In diesem Zustand stoppt die Maschine.

Damit man diesen Plan in seiner Ausführung besser verfolgen kann, nennen wir kurz die Funktion der dreizehn Zustände.

In Zustand q_0 wird die rechteste 1 gelöscht, oder erkannt, dass es keine mehr gibt, die gelöscht werden könnte. Die gleiche Funktion am linken Ende erfüllt Zustand q_3 .

In Zuständen q_1 und q_4 wird der Kopf nach der Löschung einer Eins am Ende wieder auf Daten gerückt.

In Zustand q_2 fährt der Kopf ganz nach links, bis er ein leeres Feld sieht, und dann fährt er ein Feld nach rechts, damit er auf eine zu löschende Eins ausgerichtet ist, falls es eine gibt. Zustand q_5 erfüllt die gleiche Funktion für die Kopffahrt bis ans rechte Ende.

Wenn in Zustand q_0 eine 0 gelesen wird, gibt es am rechten Ende keine 1 mehr, und Zustand q_6 untersucht links von der 0, ob die linke Zahl des Paares auch schon ganz gelöscht ist. Wenn ja, ist die Stelle links von der 0 leer; dort wird eine 1 geschrieben, und in Zustand q_7 wird auch die 0 zu einer 1 gemacht, und die Maschine stoppt anschließend in Zustand q_{12} . Auf dem Band stehen zwei Einsen, also die Zahl 1.

Wenn die Zelle links von der 0 nicht leer ist, dann arbeiten q_8 und q_9 zusammen, um alle Einsen links von der 0 zu löschen. Wenn ein leeres Feld als nächste Löschstelle auftaucht, ist man damit fertig, und Zustand q_7 fährt den Kopf auf die 0 zurück, ersetzt sie durch eine 1, und stoppt die Maschine in Zustand q_{12} mit der Zahl 0 auf dem Band.

Das gleiche Ergebnis ist erwünscht, wenn in Zustand q_3 eine 0 gelesen wird. Dann gibt es links von der 0 nichts mehr zu löschen, obwohl rechts von der 0 gerade eine 1 gelöscht wurde. Also müssen die ursprünglichen Zahlen ungleich gewesen sein.

Zustände q_{10} und q_{11} arbeiten dann zusammen, um alle Einsen rechts von der trennenden 0 zu löschen. In Zustand q_{10} fährt der Kopf dann auf diese 0 zurück, wandelt sie in eine 1, und stoppt die Maschine.

- g) Die letzte Beispielmachine erwartet auf dem Band eine Bitfolge aus Nullen und Einsen, die als eine Binärzahl zu interpretieren ist, und der Schreib-Lese-Kopf steht am rechten Ende dieser Zahl. Die Turingmaschine erhöht die Binärzahl um 1.

Obwohl diese binäre Bitfolge nicht unser Standardzahlenformat ist, haben wir tatsächlich eine Verwendung für diese Maschine im Auge.

Das Programm ist relativ einfach:

q_0	1	0	q_1
q_0	0	1	q_2
q_0	\square	1	q_2
q_1	0	L	q_0
q_2	0	R	q_2
q_2	1	R	q_2
q_2	\square	L	q_3

Hier die Funktionsweise: Zustand q_0 bearbeitet Einsen am rechten Ende der Zahl. Sie werden alle zu 0 gemacht und anschließend wird in Zustand q_1 der Schreib-Lese-Kopf nach links bewegt und die Maschine kommt wieder in Zustand q_0 .

Sobald in Zustand q_0 aber eine 0 oder ein leeres Feld gelesen wird, muss dieses Feld auf 1 erhöht werden. Anschließend befindet sich die Maschine in Zustand q_2 , und in diesem Zustand fährt der Kopf ganz bis ans rechte Ende der Bitfolge und darüber hinaus auf die erste Leerstelle.

Hier wird der Kopf auf den Anfang der Daten zurückgebracht und die Maschine stoppt in Zustand q_3 .

Bemerkung 7.4 Wir wollen aus diesen Beispielen und aus anderen auf Grund der Beispiele plausibel gewordenen Turingmaschinen, die wir nicht im Detail konstruiert haben, immer kompliziertere Turingmaschinen zusammensetzen. Dabei ist folgende Tatsache von großem Nutzen:

Seien $D_1 \subseteq \mathbf{N}^k$ und $D_2 \subseteq \mathbf{N}^m$ nichtleere Teilmengen, und seien

$$f: D_1 \longrightarrow \mathbf{N}^m \quad \text{und} \quad g: D_2 \longrightarrow \mathbf{N}^n$$

partiell definierte Funktionen, die durch Turingmaschinen T_f bzw. T_g berechenbar sind.

Dann ist auch die Verknüpfung $g \circ f$ durch eine Turingmaschine berechenbar.

Man erhält eine solche Turingmaschine T , wenn man die Zustände q_i von T_g so zu neuen Zuständen q'_i umbenennt, dass T_f und T_g keine gemeinsamen Zustände haben. Als das Alphabet, die Zustandsmenge, und das Programm von T nimmt man jeweils die Vereinigung der entsprechenden Mengen für T_f und T_g , und man fügt für alle Situationen in den Zuständen von T_f , in denen T_f stoppt, neue Programmbefehle hinzu, die den Terminalzustand von T_f in den Initialzustand q'_0 von T_g überführen aber sonst nichts verändern.

Dadurch ist gewährleistet, erstens dass die Programme von T_f und T_g nicht durcheinander kommen, weil kein Zustand aus dem ersten Programm auch im zweiten Programm vorkommt, und zweitens, durch die Überleitung in den Anfangszustand von T_g , dass nachdem T_f seine Berechnung beendet hat, T_g mit dem Ergebnis weiterrechnet und seinen Teil der Berechnung auf diesen Daten vollständig durchführt.

Bei Eingabedaten (n_1, n_2, \dots, n_k) entsteht auf diese Weise tatsächlich als Endergebnis

$$g(f(n_1, n_2, \dots, n_k)).$$

Diese Bemerkung besagt nichts anderes, als dass man Turingmaschinen hintereinander schalten kann, um neue leistungsfähigere Turingmaschinen zu bauen.

Wir wollen jetzt basierend auf dieser Bemerkung und den Beispielen einige Aufgaben nennen, die durch eine Turingmaschine ausgeführt werden können, auch wenn wir diese nicht vollständig angeben können.

Bemerkung 7.5 a) Ein Turingmaschinenprogramm ist eine Folge von Quadrupeln, in die Zustände q_i , Schriftzeichen S_i und Bewegungen L oder R eingehen. Alle diese Daten sind nur endlich oft vorhanden, und deshalb kann man die vorhandenen Zustände, Zeichen und Bewegungen durch natürliche Zahlen aufzählen und diese Zahlen auch als Bezeichnung für die Daten verwenden.

Wenn wir das machen, wird ein Turingmaschinenprogramm mit n Befehlen zu einer Folge von Zahlenquadrupeln oder zusammengefasst, zu einem einzigen $4n$ -Tupel, das man in Standardformat auf das Band einer Turingmaschine schreiben kann.

Eine Turingmaschine kann eine beliebige Bitfolge aus Nullen und Einsen untersuchen und feststellen, ob sie ein gültiges Turingmaschinenprogramm darstellt. Dabei könnte die vergleichende Turingmaschine aus Beispiel 7.3 f) zum Einsatz kommen.

b) Mit der Turingmaschine aus Beispiel 7.3 g) und auf Grund von Teil a) von dieser Bemerkung ist es möglich, mit einer Turingmaschine *alle* Turingmaschinenprogramme effektiv aufzuzählen.

Wenn man gleichzeitig einen Zähler auf dem Band führt und für jedes gültige Programm hochzählt, kann man für jede natürlichen Zahl n mit einer Turingmaschine das n -te gültige Turingmaschinenprogramm in der obigen binären Zählreihenfolge effektiv erzeugen und auf dem Band hinterlassen.

Die eindeutig bestimmte Turingmaschine, die dieses Programm hat, nennen wir T_n . Da unsere Aufzählung garantiert alle Turingmaschinen erfasst, haben wir so eine effektiv berechenbare Bijektion zwischen der Menge der natürlichen Zahlen und der Menge aller Turingmaschinen.

Die Zahl n , die dabei einer Turingmaschine T zugeordnet wird, in anderen Worten, die Zahl n mit $T = T_n$, nennen wir die **Gödel-Nummer** der Turingmaschine T , und wir nennen T_n die **Turingmaschine mit Gödelnummer n** .

- c) Da wir unser Alphabet vergrößern dürfen, ist es möglich, sowohl ein Turingmaschinenprogramm wie auch Daten für die Turingmaschine gleichzeitig auf erkennbare Weise auf das Band zu schreiben. Dazu schreibe man an beiden Enden des Programms auf dem Band ein Sonderzeichen, das den Programmbereich anzeigt. Außerhalb dieses Bereichs schreibe man Daten für die Turingmaschine, und zwar rechts vom Programmbereich die Bandkonfiguration, die unter und rechts vom Schreib-Lese-Kopf stehen soll, und links vom Programmbereich die Bandkonfiguration, die links vom Schreib-Lese-Kopf stehen soll.
- d) Da wir in den Beispielen gesehen haben, dass man mit Turingmaschinen Daten auf dem Band verschieben und kopieren und vergleichen kann, leuchtet es hoffentlich ein, dass es möglich ist, eine Turingmaschine M zu programmieren, die im Format von Teil c) dieser Bemerkung ein Turingmaschinenprogramm mit zugehörigen Daten auf dem Band erkennen kann, zusätzlichen Platz auf dem Band für einen Zustandszähler durch Verschiebung von anderen Daten freimachen kann, und dann unter Verwendung dieses Zustandszählers das auf dem Band befindliche Turingmaschinenprogramm lesen und auf den Daten *ausführen* kann.

Dazu muss die Turingmaschine M nur die Quadrupel des Bandprogramms durchsuchen und den ersten Zustand im Quadrupel mit dem gegenwärtig gespeicherten Zustand vergleichen, sowie das gelesene Zeichen im Quadrupel mit dem gegenwärtig an der Lesestelle stehenden Zeichen vergleichen.

Bei Übereinstimmung ist es leicht, den neuen Zustand in den Zustandszähler zu kopieren und das zu schreibende Zeichen an die Schreibstelle zu kopieren, oder die Daten zu verschieben, um eine Kopfbewegung zu simulieren.

Weil die Turingmaschine M für die Simulation einen Zustandszähler auf dem Band benutzt, kann sie selber mit einer festen Anzahl von

eigenen internen Zuständen auskommen, die unabhängig von dem Turingmaschinenprogramm auf dem Band ist.

Wir haben hier begründet, dass es eine Turingmaschine gibt, die das Programm einer beliebigen anderen Turingmaschine interpretieren und emulieren kann.

- e) Die Turingmaschine aus Teil d) kann verfeinert werden zu einer Turingmaschine U , die als Eingabe ein $k + 1$ -Tupel (n_0, n_1, \dots, n_k) von natürlichen Zahlen verlangt (für beliebiges k), und die dann genau das macht, was T_{n_0} , die Turingmaschine mit Gödelnummer n_0 , mit den Daten (n_1, \dots, n_k) machen würde. Sie produziert nicht nur die gleichen Ergebnisse, sondern hält auch genau dann nie an, wenn T_{n_0} nie anhalten würde.

Um das zu erreichen muss man der Maschine M aus Teil d) nur eine andere Turingmaschine vorschalten, die das n_0 -te Turingmaschinenprogramm erzeugt und an der richtigen Stelle auf dem Band hinterlässt. Wir wissen schon, dass diese Aufgabe von einer Turingmaschine ausgeführt werden kann.

Die Maschine U , die wir hier beschrieben haben, nennt sich eine **universelle Turingmaschine**.

Allmählich mag es so erscheinen, dass man *jede* nur erdenkbare Rechenaufgabe mit einer Turingmaschine berechnen kann. Das ist leider nicht der Fall, und es gibt tatsächlich **nichtberechenbare Funktionen**, sogar von \mathbf{N} nach \mathbf{N} .

Satz 7.6 Das **Haltproblem** ist das Problem, durch eine Berechnung zu bestimmen, ob eine gegebene Turingmaschine bei gegebenen Eingangsdaten auf dem Band irgendwann anhält oder in einer Endlosschleife ewig weiterrechnet.

Es stellt sich heraus, dass das Haltproblem unlösbar, d. h., nicht effektiv entscheidbar ist. Genauer, sogar folgender Spezialfall ist nicht entscheidbar:

Sei $f: \mathbf{N} \rightarrow \mathbf{N}$ die Funktion

$$f(n) := \begin{cases} 0, & \text{wenn } T_n \text{ mit Eingabe } n \text{ nach endlicher Zeit anhält.} \\ 1, & \text{wenn } T_n \text{ mit Eingabe } n \text{ nie anhält.} \end{cases}$$

Es gibt keine Turingmaschine, die f berechnet.

Beweis. Angenommen, es gäbe eine Turingmaschine T , die f berechnet. Sei

M die Turingmaschine mit Programm

q_0	1	L	q_1
q_1	\square	R	q_0
q_1	1	R	q_2

Wenn diese Maschine die Zahl 0 auf dem Band vorfindet, dann geht sie in eine Endlosschleife, aber wenn sie eine natürliche Zahl ≥ 1 vorfindet, dann hält sie in Zustand q_2 am rechten Ende dieser Zahl.

Wenn man diese Maschine *hinter* T schaltet, dann erhält man eine Turingmaschine G , die bei Eingabe einer natürlichen Zahl n genau dann anhält, wenn $f(n) = 1$, aber nicht anhält, wenn $f(n) = 0$.

Wenn man die Definition von f berücksichtigt, dann sieht man, dass G bei Eingabe von n genau dann anhält, wenn T_n mit Eingabe n *nicht* anhält.

Die Turingmaschine G hat aber auch eine Gödel-Nummer, sagen wir m . Was eben gesagt wurde bedeutet, dass G mit Eingabe m genau dann anhält, wenn $T_m = G$ bei Eingabe m *nicht* anhält.

Das ist ein Widerspruch und zeigt die Unmöglichkeit einer Turingmaschine, die f berechnet. ■

Es gibt viele andere unentscheidbare Probleme, die sich aber in den meisten Fällen letztendlich auf dieses Problem zurückführen lassen.

Literatur

Für Hörer, die sich einzelne Quellen selber besorgen wollen, ist in den sinnvollen Fällen die ISBN Nummer in der Literaturliste mit angegeben.

Literatur

- [1] Ken Arnold, James Gosling and David Holmes. *The Java Programming Language. Third Edition.* Addison-Wesley, Boston-San Francisco-New York-Toronto-Montreal, 2000. ISBN 0-201-70433-1
- [2] Noam Chomsky. *Syntactic Structures.* Mouton & Co., London-The Hague-Paris, Fifth printing, 1965. Das klassische Werk über formale Grammatiken, Erstveröffentlichung 1957.
- [3] Edgar W. Dijkstra. *Over seinpalen.* <http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD74.PDF> (18.07.2006)
- [4] Heinz-Peter Gumm und Manfred Sommer. *Einführung in die Informatik.* Oldenbourg Wissenschaftsverlag, München-Wien, 4., überarbeitete Auflage, 2000. ISBN 3-486-25050-7
- [5] John P. Hayes. *Computer Architecture and Organization.* International Student Edition. McGraw-Hill Book Company, Singapore, 1979. ISBN 0-07-Y66319-X
- [6] Petra Hornstein. *PetriEdiSim. Tutorial und Editor für Petri-Netze.* <http://olli.informatik.uni-oldenburg.de/PetriEdiSim/> (01.02.01)
- [7] Stephen Cole Kleene. *Introduction to Metamathematics.* D. Van Nostrand Company, Inc., Princeton-Toronto-New York, 1950.
- [8] Lance A. Leventhal. *8080A/8085 Assembly Language Programming.* OSBORNE/McGraw-Hill, Berkeley, California, 1978. ISBN 0-931988-10-1

- [9] Walter Oberschelp und Gottfried Vossen. *Rechneraufbau und Rechnerstrukturen*. R. Oldenbourg Verlag, München-Wien, 4., verbesserte Auflage, 1990. ISBN 3-486-21743-7
- [10] Adam Osborne. *An Introduction to Microcomputers: Volume 1 — Basic Concepts*. OSBORNE/McGraw-Hill, Berkeley, California, 2nd edition, 1980. ISBN 0-931988-34-9
- [11] Lutz Richter. Geräte- und Datei-Verwaltung. <http://www.ifi.unizh.ch/~riedl/lectures/B-7.pdf> (01.03.2007)
- [12] Helmut Rohlfing. *PASCAL. Eine Einführung*. BI-Hochschultaschenbücher Bd. 756. Bibliographisches Institut, Mannheim-Wien-Zürich, 1978. ISBN 3-411-00756-7
- [13] David Spuler. *Comprehensive C*. Prentice Hall, New York-London-Toronto-Sydney-Tokyo-Singapore, 1992. ISBN 0-13-156514-1
- [14] Andrew S. Tanenbaum. *Structured Computer Organization*. Pearson Prentice Hall, Upper Saddle River, New Jersey, 5th edition, 2006. ISBN 0-13-148521-0
- [15] Andrew S. Tanenbaum. *Computerarchitektur*. Pearson Studium, München, 5. Auflage, 2006. ISBN 3-8273-7151-1. (Die deutsche Übersetzung von [14])

Die Literaturliste ist alphabetisch sortiert. Bis auf die englische Ausgabe [14] des Buches von Tanenbaum wurden die angegebenen Quellen nur für einzelne, sehr beschränkte Themen während der Vorlesung verwendet.

Neben den aufgeführten Monographien wurden im Laufe des Semesters sehr hilfreiche Informationen von vielen Quellen aus dem Internet bezogen, darunter [6] über Petri-Netze, die Vorlesungsfolien [11] von Prof. Dr. Lutz Richter in Zürich über parallele Verarbeitung und Semaphore in Bezug auf Gerätetreiber, E. W. Dijkstras ursprüngliche maschinengeschriebene Arbeit über Semaphore unter [3] als .pdf-Datei im Internet archiviert, und natürlich allgemeine Quellen wie die Wikipedia

<http://de.wikipedia.org/wiki/Hauptseite>

für kurze Antworten zu allen möglichen Fragen.