150 Jahre
Wissen für die Zukunft
Oldenbourg Verlag

Technische Informatik

Eine einführende Darstellung

von
Prof. Dr. Bernd Becker
Prof. Dr. Paul Molitor

Oldenbourg Verlag München Wien

Bernd Becker ist Inhaber des Lehrstuhls für Rechnerarchitektur an der Albert-Ludwigs-Universität Freiburg. Seine Hauptarbeitsgebiete sind Entwurf, Verifikation und Test von Schaltungen und Systemen. Seine Arbeiten werden unterstützt durch umfangreiche Drittmittelprojekte sowohl von DFG, BMBF als auch von Geldgebern direkt aus der Industrie. Zur Zeit ist er stellvertretender Sprecher des SFB Transregios der DFG "Automatic Verification and Analysis of Complex Systems".

Paul Molitor ist Professor für Technische Informatik an der Martin-Luther-Universität Halle-Wittenberg. Vor seiner Tätigkeit an der Universität Halle war er Professor für Schaltungstechnik an der Humboldt-Universität zu Berlin (1993/94) bzw. Projektleiter in dem an der Universität des Saarlandes und der Universität Kaiserslautern angegliederten Sonderforschungsbereich "VLSI Entwurfsmethoden und Parallelität" (1983–1992). Er studierte Informatik und Mathematik an der Universität des Saarlandes (Diplom 1982, Promotion 1986, Habilitation 1992).

Bibliografische Information der Deutschen Nationalbibliothek

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

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 4 50 51-0
oldenbourg.de

Lektorat: Dr. Margit Roth
Herstellung: Anna Grosser
Coverentwurf: Kochan & Partner, München
Gedruckt auf säure- und chlorfreiem Papier
Gesamtherstellung: Kösel, Krugzell

ISBN 978-3-486-58650-3

Vorwort

Zu dieser Auflage

Das vorliegende Buch ist eine Überarbeitung des Buches „Technische Informatik. Eine Einführung“, das vor drei Jahren bei Pearson Studium erschienen ist. Wenn auch die Entwicklung von Rechnern und die Bedeutung integrierter Schaltungen und eingebetteter Systeme in diesen drei Jahren weiter rasant voran geschritten ist, die für das Basisverständnis notwendigen Grundlagen haben sich nicht wesentlich verändert.

Nichtsdestotrotz war es sinnvoll, das Buch zu überarbeiten und an einigen Stellen zu modifizieren. Nur kleinere Änderungen, die bis auf wenige Ausnahmen eher kosmetischer Natur sind, erfuhren die Kapitel aus Teil I „Mathematische und elektronische Grundlagen“. Größere Änderungen finden sich in Teil II, der auf den Entwurf digitaler Hardware eingeht. So wurde Kapitel 8 um die Synthese mittels Funktionaler Dekomposition Boolescher Funktionen erweitert. Es werden bessere obere Schranken für die Komplexität Boolescher Funktionen als noch in der ersten Auflage enthalten eingeführt. Insbesondere im Kapitel 9 wurde die Darstellung zum Teil auch aus didaktischen Gründen modifiziert. Teil III, im Speziellen die Abschnitte 10.2, 10.3, 10.4 und 11.2, wurde grundlegend überarbeitet. Wir lehnen uns nunmehr bei der Vorstellung eines Lehrprozessors eng an den MIPS-Prozessor an, wenn auch der Befehlsaufbau sich an einigen Stellen stark unterscheidet. Hierbei beschränken wir uns bewusst auf einen sehr kleinen Befehlssatz, um die wesentlichen Aspekte der Funktionsweise eines Prozessors in Bezug auf die Befehlsabarbeitung zu erläutern. Auf Erweiterungen des Prozessors, wie zum Beispiel in Bezug auf die Möglichkeit von Unterprogrammaufrufen oder Ein-/Ausgabe-Routinen, wurde im Text verzichtet; sie wurden zum Teil in den Übungsteil verschoben. *Pipelining* wird in der vorliegenden Auflage nicht nur eher allgemein erklärt, sondern insbesondere auch in Bezug auf den Lehrprozessor selbst vorgestellt. Der Lehrprozessor wurde in einem Praktikum an der Albert-Ludwigs-Universität Freiburg auf einem Actel®-FPGA realisiert und als Rechnerplattform für ein bekanntes, relativ einfaches Computerspiel eingesetzt; eine Xilinx®-Realisierung des Lehrprozessors ist zur Zeit Gegenstand von Studenten-Praktika an der Martin-Luther-Universität Halle-Wittenberg. Die Kapitel zu Teil IV sind mehr oder weniger unverändert zu der ersten Auflage.

Die Kapitel wurden zum Teil durch eine Vielzahl von weiteren Übungsaufgaben – das Buch enthält nun um die 120 Übungsaufgaben – ergänzt, die es den Studierenden noch besser erlauben, ihr Wissen zu überprüfen und zu erweitern.

Zum Inhalt

Integrierte Schaltkreise haben in den vergangenen Jahren massiv unsere Umwelt verändert. Rechnersysteme in den verschiedensten „Ausprägungen“ sind integraler Bestandteil des täglichen Lebens geworden: Sie finden sich – um nur einige Beispiele zu nennen – als Laptops in Betrieben, Schulen, Universitäten und im privaten Bereich, als Großrechner im Banken- und Versicherungswesen, aber auch als Mikrocontroller zur Lösung komplexer Regelungs- und Steuerungsaufgaben in Autos, Eisenbahnen und Flugzeugen. Nicht zuletzt sei auf ihre Bedeutung in der Mobilkommunikation und Unterhaltungselektronik hingewiesen.

Unabhängig von der Anwendung gibt es gemeinsame Basiskomponenten, aus denen sich dann Rechner mit durchaus unterschiedlichen „Architekturen“ aufbauen lassen. Rechner „optimal“ für eine Zielanwendung zu konstruieren oder auch nur einzusetzen, wird in zunehmendem Maße in den Aufgabenbereich von Informatikern fallen. Ein tieferes Verständnis der Rechnerarchitektur ist damit unerlässlich. Die Vermittlung dieser Fähigkeiten ist eine Kernaufgabe der Technischen Informatik, die die Integration unterschiedlicher Bereiche erfordert. Das vorliegende Buch will hierzu einen Beitrag leisten.

Unter dem Begriff der *Rechnerarchitektur* verstand man bis vor einigen Jahren vielfach nur die *Software/Hardware-Schnittstelle* eines Rechners, also welcher Satz von *Maschinenbefehlen*, d. h. welche *Maschinensprache*, dem Menschen zur Verfügung steht, um mit dem Rechner „reden“ zu können. Wie die einzelnen Maschinenbefehle dann durch den Rechner letztendlich ausgeführt werden, wurde im Rahmen der Rechnerarchitektur nicht mehr betrachtet. Die Leistungsfähigkeit eines Rechners ergibt sich aber nicht (nur) aus den Möglichkeiten, die zu seiner Programmierung zur Verfügung stehen. Vielmehr hängt die Effizienz in großem Maße davon ab, wie schnell der Prozessor die verschiedenen Maschinenbefehle ausführen kann. Dieses Lehrbuch beschäftigt sich also nicht nur mit der Software/Hardware-Schnittstelle, sondern geht auch detailliert auf den Aufbau und den Entwurf einzelner Hardwarekomponenten, insbesondere auf die Entwurfsmöglichkeiten, sowie auf das Zusammenspiel dieser Komponenten und somit auf die prinzipielle Arbeitsweise eines Prozessors ein. Hierbei werden wir, wie oben bereits angedeutet, einen Bogen von den mathematischen und technischen Grundlagen bis hin zu algorithmischen Aspekten schlagen.

Rechner kann man als eine Hierarchie virtueller Rechner verstehen. Hierbei stehen die verschiedenen virtuellen Rechner für verschiedene Abstraktionsebenen. Eine zentrale Ebene ist die gerade schon angesprochene Maschinensprache-Ebene, auf der man den Rechner als eine Maschine sieht, die Programme in Maschinensprache einliest und gemäß der definierten Semantik der Maschinensprache korrekt ausführt. Die Maschinensprache hat entscheidenden Einfluss auf die Komplexität des eigentlichen, physikalischen Rechners und auf seine Performanz. Auf einen einfachen Nenner gebracht, kann man in etwa sagen, dass

- je größer der Satz der Maschinenbefehle ist, der von der Maschinensprache bereitgestellt wird, und
- je komplexer die Operationen sind, die diese Maschinenbefehle ausführen,

- desto komplexer wird die Realisierung des physikalischen Rechners sein, der als Maschinensprache gerade diese Sprache besitzt,
- desto schwieriger wird es sein, hohe Taktraten zu erreichen, da sich der Takt nach der langsamsten Operation richten muss,
- desto bequemer wird es der Mensch (und der Compiler) haben, Programme für den Rechner zu schreiben.

Im Rahmen dieses Buches werden wir die Maschinensprache-Ebene und die darunter liegenden Ebenen, die die Hardware eines Rechners „ausmachen", detailliert betrachten. Wir belassen es hierbei nicht bei einer reinen Vorstellung des Aufbaus und der Funktionsweise der einzelnen Teile eines Rechners, sondern versuchen darüber hinaus, dem Leser das mathematische und technologische Grundverständnis für die Funktionsweise der einzelnen Komponenten und des Gesamtsystems nahe zu bringen. Zudem geht das Buch auf algorithmische Aspekte des Schaltkreisentwurfes ein, d. h. stellt Datenstrukturen und Algorithmen vor, wie zu einer Spezifikation Hardware synthetisiert werden kann, die dieser Spezifikation genügt.

Wie ist nun das Buch aufgebaut?

- In Teil I werden mathematische und elektronische Grundlagen bereitgestellt, deren Kenntnis für den Entwurf von Rechnern, aber auch für das Verständnis der Funktionsweise eines Rechners unerlässlich ist.
- Darauf aufbauend gibt Teil II Einblicke in Entwurfsaufgaben wie die Optimierung digitaler Schaltkreise. Es werden spezielle Schaltkreise exemplarisch entworfen und der Aufbau ausgewählter, als Komponenten von Rechnern benötigter Hardwarebausteine (z. B. ALU) diskutiert. Damit werden die Voraussetzungen geschaffen, die benötigt werden, um sich konkret dem Entwurf von Rechnern zuwenden und diesen auch durchführen zu können.
- Ein konkreter Entwurf eines Mikroprozessors erfolgt in Teil III. Im Mittelpunkt dieses Kapitels steht die Definition der Maschinensprache, über die sich unser Prozessor definiert, deren Umsetzung in Hardware sowie Konzepte zur Beschleunigung dieses Rechners.
- Das Buch schließt mit Teil IV, in dem erläutert wird, wie die Kommunikation zwischen Rechnern (oder Hardware-Modulen) erfolgt, insbesondere wird dieser Teil auf die verschiedenen Bustypen und auf verschiedene Kodierungen der zu übermittelnden Daten eingehen.

Wir gehen nun etwas detaillierter auf die einzelnen Teile ein, um dem Leser eine Art Leitfaden an die Hand zu geben.

Das Einleitungskapitel

In der Einleitung (Kapitel 1) vertiefen wir die oben schon angesprochene Idee der virtuellen Rechner und ihrer Hierarchie. Abschnitt 1.1 ist, wenn auch einfach, für das allgemeine Verständnis der folgenden Kapitel unerlässlich. Das Kapitel schließt mit einem

historischen Rückblick auf die Entwicklung der Rechner, die sich eng an die Entwicklung der zu Grunde liegenden Technologien anlehnt.

Teil I: Grundlagen

Der erste Teil des Buches beschäftigt sich mit verschiedenen Grundlagen, deren Kenntnis eine notwendige Voraussetzung für das Verständnis der späteren, weiterführenden Betrachtungen ist. Auch wenn man die Inhalte dieser Kapitel auf den ersten Blick statt der Technischen Informatik zunächst eher den Bereichen der Mathematik und Elektrotechnik zuordnen würde, so spielen diese bei eingehender Betrachtung in unserem Kontext eine wichtige Rolle.

In Kapitel 2 werden die grundlegenden mathematischen Begriffe eingeführt, wie zum Beispiel Boolesche Algebren, Boolesche Funktionen und Boolesche Ausdrücke, also Begriffe, die im Zusammenhang der Schaltungstheorie eine zentrale Rolle spielen. Es stellt damit einen wichtigen „Werkzeugkasten“ für die weiteren Kapitel bereit.

Kapitel 3 beschreibt und diskutiert, wie Zeichen und Zahlen im Rechner zu repräsentieren sind. Da die Darstellung von Zeichen und Zeichenfolgen in den Kapiteln 13 und 14 nochmals detailliert aufgegriffen wird, insbesondere in Bezug auf Fehlertoleranz und Längen-Optimalität, steht die Darstellung von Zahlen im Mittelpunkt dieses Kapitels. Es werden verschiedene Darstellungen vorgestellt und ihre Vor- und Nachteile diskutiert.

Die Kapitel 4 und 5 geben eine kurze Einführung in die elektronischen Grundlagen von Schaltungen. Diese sind notwendig, um verstehen zu können, wie Signale in einem Rechner auf der physikalischen Ebene verarbeitet werden. Auch wenn die restlichen Teile des Buches, bis auf wenige Abschnitte, nicht auf diese beiden Kapitel zurückgreifen, sind die hier vermittelten Grundlagen notwendig, um die Funktionsweise von Rechnern und ihr Zusammenspiel mit der „Umwelt“ wirklich verstehen zu können. Es werden Fragen wie „Wie funktioniert eine Taste?“, „Wie ist ein Tastenfeld (Tastatur) aufgebaut?“ oder „Wie arbeitet ein Analog/Digital-Wandler?“ behandelt. Die letzte Frage hat eine zentrale Bedeutung, denkt man zum Beispiel an sogenannte „eingebettete Systeme“, bei denen oft ein analoger Wert (Temperatur, Luftfeuchtigkeit, Säuregehalt usw.) gemessen werden muss, um ihn dann durch einen (digitalen) Rechner verarbeiten zu lassen.

Teil II: Entwurf digitaler Hardware

Teil II veranschaulicht, wie ausgehend von einer Spezifikation einer Schaltung, die durch eine funktionale Beschreibung gegeben ist, eine Realisierung als Schaltkreis erzeugt werden kann. Die Kapitel 7 und 8 stellen Entwurfsmethoden für zwei- und mehrstufige kombinatorische Schaltungen vor und besprechen automatische Verfahren, wie zu einer gegebenen Booleschen Funktion eine kostengünstige Realisierung gefunden werden kann. Diese Verfahren basieren zum Teil auf effizienten Repräsentationsformen für Boolesche Funktionen, die eine effiziente Bearbeitung erlauben. Eine solche Datenstruktur, die sogenannten „Decision Diagrams“, die „mächtiger“ als die in Kapitel 2 vorgestellten ist, wird in Kapitel 6 eingeführt. Für spezielle Funktionen, wie die Addition oder Multiplikation, die in der Praxis sehr häufig vorkommen, werden in Kapitel 9 spezielle Schaltkreise entwickelt und analysiert. Dieses Kapitel stellt zudem vor, wie eine

arithmetisch-logische Einheit (ALU) mithilfe der entwickelten Methoden konstruiert werden kann.

Teil III: Architektur eines Prozessors

In Teil III wird ein kleiner Mikroprozessor „gebaut“. Hierbei geht das Buch schrittweise vor.

In Kapitel 10 wird ein Ein-Zyklus-Mikroprozessor entworfen, d. h. ein Prozessor, der jeden seiner Befehle in einem Takt vollständig ausführt. Damit vermeiden wir, schon an dieser Stelle über Konzepte wie Befehlspipelining sprechen zu müssen. Das Kapitel versucht zuerst, dem Leser ein Gefühl für den grundsätzlichen Aufbau eines Rechners zu vermitteln. Eine Menge von Befehlen wird vorgestellt, über welche die Maschinensprache unseres Prozessors definiert wird. Es wird gezeigt, wie der Prozessor, welcher das „Herz“ eines jeden Rechners darstellt, die verschiedenen Maschinenbefehle ausführen kann. Hierdurch wird das Zusammenwirken der einzelnen Komponenten des Rechners detailliert und anschaulich erklärt. Das Kapitel widmet sich zum Schluss dem Steuerwerk des Prozessors.

Nachdem in Kapitel 10 der grundsätzliche Aufbau eines Prozessors und seiner Maschinensprache diskutiert wurde, dabei aber von zahlreichen Details abstrahiert wurde, wird in Kapitel 11 untersucht, wie der eben vorgestellte Prozessor zu einem leistungsfähigeren Mehr-Zyklen-Rechner ausgebaut werden könnte. Das Kapitel beginnt mit einer prinzipiellen Diskussion der CISC- und RISC-Philosophie. Hier wird die oben angesprochene Frage des Einflusses der Architektur der Maschinensprache auf die Leistungsfähigkeit eines Prozessors detailliert, wenn auch stark vereinfacht, diskutiert. Daran schließt sich die Vorstellung von Ansätzen zur Beschleunigung der Befehlsabarbeitung: Abarbeitung durch Befehlspipelining und effiziente Realisierung von Speicherzugriffen mithilfe einer Speicherhierarchie. Integriert in dieses Kapitel ist eine Vorstellung des Aufbaus der verschiedenen Speichertypen wie zum Beispiel statische und dynamische Speicherzellen.

Teil IV: Kommunikation

Teil IV beschäftigt sich abschließend mit der Frage, wie Informationen und Daten zwischen Komponenten bzw. zwischen Rechnern übertragen werden können.

In Kapitel 12 werden die Grundlagen der Kommunikation zwischen Hardwarekomponenten von einem logischen Standpunkt aus erklärt. Es setzt sich mit seriellen und parallelen Bussystemen und dazugehörigen Protokollen, insbesondere mit Bus-Arbitrierung auseinander.

Da bei dem Übertragen von Daten Fehler auftreten können, ist es zum Teil wichtig, Codes zu benutzen, die fehlertolerant sind. Fehlertolerante Codes sind Codes, die es einem Empfänger erlauben zu erkennen, ob ein Fehler bei der Übertragung erfolgt ist, beziehungsweise Fehler zu erkennen *und* zu korrigieren. Dieser Thematik ist das Kapitel 13 gewidmet.

Neben fehlertoleranten Codes sind sogenannte Häufigkeitscodes ebenfalls von zentraler Bedeutung. Das Ziel dieser Codes besteht darin, die Zeichen eines Codes so zu kodieren,

dass die Übertragung einer Sequenz von Zeichen möglichst wenig Aufwand zur Folge hat. Neben einer kurzen Einführung in die Informationstheorie, die die mathematische Grundlage längenoptimaler Codes bildet, stellt Kapitel 14 drei Kodierungen vor, die versuchen, die mittlere Codewortlänge zu minimieren, die Shannon-Fano-Kodierung, die Huffman-Kodierung und die sogenannte arithmetische Kodierung.

An wen richtet sich das Buch?

Das Buch richtet sich insbesondere an Studierende der Informatik im Grundstudium bzw. in Bachelor-Studiengängen. Spezielle Vorkenntnisse werden nicht vorausgesetzt. Wir haben versucht, alle benötigten Grundlagen in den einführenden Kapiteln bereitzustellen und in diesem Sinne ein geschlossenes Buch zu erarbeiten. Wir sind der Überzeugung, dass das Buch sowohl als vorlesungsbegleitende Literatur benutzt werden kann, als auch als Grundlage zum Selbststudium geeignet ist.

Danksagung

Das vorliegende Buch ist aus Vorlesungen zur Technischen Informatik entstanden, die die beiden Autoren und Rolf Drechsler, Mitautor der 2005 bei Pearson Studium erschienenen ersten Auflage des Buches, seit 1993 in Bremen, Freiburg und Halle gehalten haben. Insofern sind wir zahlreichen Kollegen, (ehemaligen) Mitarbeitern und Studierenden zu Dank verpflichtet, insbesondere natürlich Rolf Drechsler. Sie alle haben durch konstruktive Kritik und Mitarbeit wesentlich zu dem Buch in seiner jetzigen Form beigetragen. Besonders nennen – natürlich in der Hoffnung möglichst niemand vergessen zu haben – möchten wir: Changxing Dong, Nicole Drechsler, Rüdiger Ebendt, Julian Eichstätt, Piet Engelke, Thomas Eschbach, Görschwin Fey, Riccardo Forth, Daniel Große, Wolfgang Günther, Harry Hengster, Mark Herbstritt, Andreas Hett, Martin Keim, Marcus Kocum, Matt Lewis, Christoph Löffler, Christian Mariner, Tobias Nopper, Ilia Polian, Jörg Ritter, Christoph Scholl, Frank Schmiedle, Tobias Schubert, Jürgen Schüle, Kelley Strampp, Lisa Teuber, Sandro Wefel, Martina Welte und Ralf Wimmer. Natürlich ist das Buch auch beeinflußt durch die Lehrveranstaltungen, die wir selbst während unserer Studienzeit besucht haben. In diesem Sinne möchten wir insbesondere unserem akademischen Lehrer Günter Hotz danken.

Freiburg im Breisgau / Halle an der Saale, im Januar 2008

Bernd Becker / Paul Molitor

Inhaltsverzeichnis

1 Einleitung

Dasselbe, was Du auf rechnerischem Weg gemacht hast, habe ich kürzlich mechanisch versucht und eine aus elf vollständigen und sechs verstümmelten Rädchen bestehende Maschine gebaut, welche gegebene Zahlen im Augenblick automatisch zusammenrechnet: addiert, subtrahiert, multipliziert und dividiert. Du würdest hell auflachen, wenn Du da wärest und sehen könntest, wie sie, so oft es über einen Zehner oder Hunderter weggeht, die Stellen zur Linken ganz von selbst erhöht oder ihnen beim Subtrahieren etwas wegnimmt.

[Brief von Schickard an Kepler am 20. September 1623]

1.1 Was ist überhaupt ein Rechner?

Die Beantwortung dieser in der heutigen Zeit scheinbar einfachen Frage hängt stark von der Sichtweise der befragten Person auf den Rechner ab. Es wird insbesondere von Bedeutung sein, wie weit der/die Befragte bei ihren Überlegungen von dem eigentlichen physikalischen Rechner abstrahiert. So bekommt man sicherlich recht unterschiedliche Antworten auf diese Frage, wenn man z. B. einen Physiker, einen Informatiker und einen Verwaltungsangestellten fragt.

Diese Sichtweisen spiegeln sich in verschiedenen Abstraktionsebenen eines Rechners wider, die bei der Beschreibung der Rechner unterschieden werden können:

- Anwendungsebene
- Ebene der höheren Programmiersprachen
- Assembler-Ebene
- Betriebssystem-Ebene

- Maschinensprache-Ebene
- Mikroprogramm-Ebene
- Hardware-Ebene

Wir geben im Folgenden erste „Definitionen“ dieser verschiedenen Ebenen.

1.1.1 Die verschiedenen Abstraktionsebenen

Auf der Hardware-Ebene betrachten wir die Hardware des Rechners, speziell wie der *Prozessor*, d. h. die Steuer- und die Recheneinheit des Rechners, über Transistoren und Grundbausteine hardwaremäßig aufgebaut ist. „Wie kann eine Addition oder eine Multiplikation von Zahlen hardwaremäßig realisiert werden?“ ist zum Beispiel eine Frage, die man auf dieser Ebene zu beantworten hat.

Die nächsthöhere Ebene betrachtet, wie die einzelnen Bausteine der Steuer- und Recheneinheit eines Prozessors in Abhängigkeit von den abzuarbeitenden *Maschinenbefehlen* angesteuert werden müssen, damit der Rechner die vom Programmierer vorgelegten Programme korrekt abarbeitet. Das allgemein angewendete Arbeitsprinzip beruht auf dem sogenannten *Fetch-Decode-Execute*-Zyklus (siehe Abbildung 1.1). Die im Prozessor enthaltenen Bausteine müssen über Steuersignale so angesteuert werden, dass in der ersten Phase des Arbeitszyklus der nächste abzuarbeitende Maschinenbefehl aus dem Speicher, in der Regel dem *Hauptspeicher*, geholt[1] wird. In der zweiten Phase wird der Maschinenbefehl durch die Steuereinheit dekodiert. Die *execute*-Phase führt dann die dekodierte Anweisung aus, nachdem gegebenenfalls für die Ausführung notwendige Operanden aus dem Speicher geladen worden sind. Die Ansteuerung der Steuersignale erfolgt bei CISC (*Complex Instruction Set Computer*) über sogenannte *Mikrobefehle*, die in Form eines Programms, des sogenannten *Mikroprogramms*, in einem Nurlesespeicher (ROM, *Read Only Memory*) gespeichert sind.

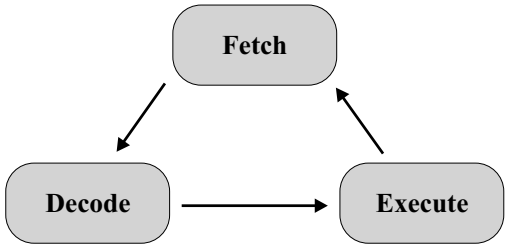


Abbildung 1.1: *Fetch-Decode-Execute-Arbeitszyklus eines Prozessors*

Die *Maschinensprache-Ebene* ist die unterste dem Programmierer frei zugängliche Sprache, in der er Programme für den Rechner schreiben kann. Diese Ebene wird aus

[1] *fetch* = holen

diesem Grunde auch *Software/Hardware-Schnittstelle* eines Prozessors genannt. Der Satz der verfügbaren Maschinenbefehle eines Prozessors bestimmt zu großen Teilen die „Architektur“ des Rechners. Sind zum Beispiel nur wenige verschiedene Maschinenbefehle in der Maschinensprache verfügbar, die zudem noch alle einfacher Natur sind und die (fast) alle ungefähr die gleiche Ausführungszeit haben, so kann die Mikroprogramm-Ebene entfallen und die Maschinenbefehle können direkt, unter Verwendung einer *Befehlspipeline*, durch die Hardware ausgeführt werden, ohne ein Mikroprogramm als Interpreter zwischenzuschalten. Man spricht in diesem Zusammenhang von einer RISC-Architektur (RISC = *Reduced Instruction Set Computer*), andernfalls von einer CISC-Architektur. Die Maschinenbefehle selbst sind Folgen über 0 und 1, also kaum verständlich und recht unbequem für den menschlichen Programmierer. Ein Maschinenprogramm ist im Hauptspeicher abgespeichert.

Die *Betriebssystem-Ebene* liegt direkt oberhalb der Maschinensprache-Ebene. Das Betriebssystem (engl.: *Operating System*) ist in unserer vereinfachten Sicht[2] ein in Maschinensprache geschriebenes Programm, welches die Betriebsmittel wie Speicher, Ein- und Ausgabegeräte verwaltet, die Ausführung von Programmen steuert und die Kommunikation (Interaktion) zwischen Mensch und Computer ermöglicht. Auf dieser Ebene kommt der Benutzer zum Beispiel nicht mehr direkt mit der Ansteuerung des Speichers und der Peripherie in Verbindung. Es werden durch das Betriebssystem die Dateien verwaltet und dem Benutzer zum Lesen und Schreiben zur Verfügung gestellt, ohne dass dieser die physikalische Adresse der Dateien auf der *Festplatte* kennen muss. Zudem gaukelt das Betriebssystem dem Nutzer vor, dass ihm der volle *adressierbare Hauptspeicher* zur Verfügung steht, auch wenn es sich zum Beispiel um ein Mehrnutzer-System handelt oder der Rechner über weniger physikalischen Hauptspeicher verfügt als prinzipiell adressierbar ist. Der englische Begriff *operating system* kennzeichnet den Sinn und Zweck eines Betriebssystems. Die in den Anfängen der Computer stark mit fehlerträchtigen Arbeiten beschäftigte Bedienmannschaft eines Rechners, die *Operator-Mannschaft* genannt wurde, schrieb sich Programme, um sich die Arbeit zu erleichtern; diese wurden nach und nach zum *operating system* zusammengefasst [38].[3]

Oberhalb der Betriebssystem-Ebene befindet sich die *Assembler-Ebene*, die eine symbolische Notation der auf der Maschinensprache- und Betriebssystem-Ebene vorhandenen Befehle zur Verfügung stellt. Zudem stellt die Assembler-Ebene dem Programmierer sogenannte *Pseudoinstruktionen* zur Verfügung, die sich sehr einfach durch nur wenige Maschinenbefehle realisieren lassen, um so dem Menschen die Arbeit beim Programmieren zu erleichtern.

Auf den *Ebenen der höheren Programmiersprachen*, wie zum Beispiel C, C++, Java, C#, Pascal oder Fortran, versteht man den Rechner als einen „schwarzen Kasten“, der Programme in der gegebenen höheren Programmiersprache einliest und direkt auf der Hardware ausführt.

Auf der *Anwendungsebene* sieht der Benutzer des Rechners nur die Anwendung – man

[2] In der Praxis stimmt dies so nicht! Vielmehr wird heutzutage ein Betriebssystem in einer höheren Programmiersprache geschrieben und wird auf einer existierenden Hostplattform durch einen sogenannten *Cross-Compiler* in ein Maschinenprogramm für die neue Hardwareplattform übersetzt.

[3] Im Rahmen des vorliegenden Buches werden wir nur auf wenige einzelne Aspekte eines Betriebssystems zu sprechen kommen. Weiterführende Informationen zu Betriebssystemen findet man z. B. in [33].

denke zum Beispiel an Informationssysteme, Programme zur Textverarbeitung oder zur Tabellenkalkulation.

1.1.2 Hierarchie virtueller Rechner

Jede Ebene mit Ausnahme der untersten, d. h. der Hardware-Ebene, ist durch eine Sprache definiert, die der Rechner auf dieser Ebene zu verstehen hat, d. h. deren Anweisungen er ausführen kann. Hierbei interessiert nicht, wie die Ausführung auf der eigentlichen Hardware letztendlich erfolgt. Vielmehr sehen wir die Rechner auf einer jeden Abstraktionsebene als „schwarzen Kasten“ an, der Programme in der entsprechenden Sprache als Eingabe akzeptiert und, entsprechend der Semantik dieser Sprache, das Programm ausführt. Wir sprechen in diesem Zusammenhang von *virtuellen Rechnern*.

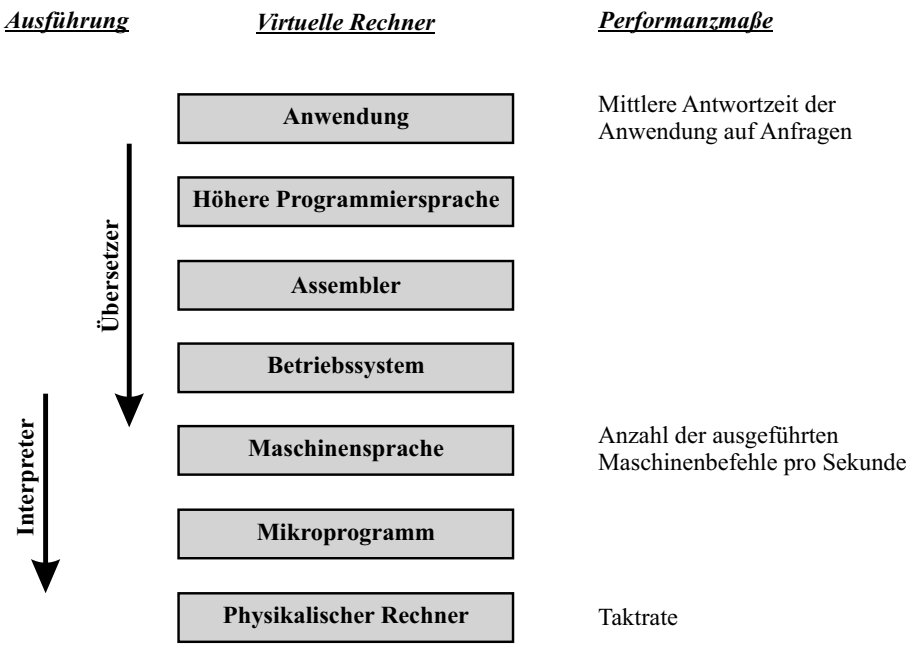


Abbildung 1.2: *Hierarchie virtueller Rechner. Links: Ausführungsart eines Programms des entsprechenden virtuellen Rechners. Rechts: die verschiedenen Performanzmaße der einzelnen Ebenen.*

Unsere Sichtweise führt uns zu einer *Hierarchie virtueller Rechner* beginnend bei dem virtuellen Rechner, der über eine höhere Programmiersprache definiert ist, bis hin zu der eigentlichen Hardware. Abbildung 1.2 zeigt die hier besprochene Hierarchie virtueller Rechner. Wie bereits oben erwähnt, ist die Mikroprogramm-Ebene nur in der CISC-Architektur vorzufinden, RISC kennt diese Ebene nicht. Links in der Abbildung ist vermerkt, wie – ob mit Übersetzung oder Interpretation – in der Regel die Aus-

führung eines Programms des entsprechenden virtuellen Rechners erfolgt. Rechts sind verschiedene Performanzmaße angegeben, die auf den entsprechenden Ebenen verwendet werden können, um die Leistung des virtuellen Rechners zu quantifizieren.

Wie oben implizit schon angedeutet, muss letztendlich jedoch jede Ebene auf die unterste Ebene, d. h. den eigentlichen physikalischen Rechner, abgebildet werden. Dies erfolgt teils durch Interpretation, teils durch Übersetzung.

Exkurs: Übersetzer und Interpreter ▷ ▷ ▷

Unter einem L_i-*Compiler* bzw. L_i-*Übersetzer* versteht man ein Maschinenprogramm, das ein Programm P_i der Sprache L_i in ein Programm P_j der Sprache L_j transformiert, das äquivalent zu P_i ist, d. h. das gleiche Ein-/Ausgabeverhalten wie P_i hat. Ist die Ausgabe eines L_i-Compilers jeweils ein Maschinenprogramm, so spricht man von einem L_i-*Vollcompiler*.

Unter einem L_i-*Interpreter* versteht man ein Maschinenprogramm, das ein Programm P_i der Sprache L_i Anweisung für Anweisung ausführt. Bevor die nächste Anweisung durch den Interpreter betrachtet wird, wird die aktuelle Anweisung auf dem physikalischen Rechner ausgeführt. Dies bedeutet insbesondere, dass während der Interpretation, im Unterschied zur Übersetzung, zu keinem Zeitpunkt ein zu P_i äquivalentes Maschinenprogramm erzeugt und abgespeichert wird. Insbesondere werden mehrfach auszuführende Programmteile immer wieder neu betrachtet und auf die Ebene der Maschinensprache abgebildet.

◁ ◁ ◁

Wir werden in den folgenden Kapiteln sehen, dass ein Maschinenprogramm nicht mehr übersetzt werden kann. Befehl nach Befehl des Maschinenprogramms muss durch das Mikroprogramm bzw. die Hardware aus dem Hauptspeicher in den Prozessor geholt (Fetch-Phase), dekodiert (Decode-Phase) und letztendlich ausgeführt (Execute-Phase) werden, bevor der nächste Maschinenbefehl durch den Prozessor betrachtet und verarbeitet werden kann.

1.2 Historischer Rückblick

Wir wollen im Folgenden die Rechner im Wandel der Zeit kurz Revue passieren lassen. Wir beginnen mit dem Jahr 1938, in dem erstmalig ein Rechner gebaut wurde, der unseren heutigen Vorstellungen entspricht. Auf frühere, zum Teil nur theoretische Ansätze, wie zum Beispiel die von Wilhelm Schickard (1592-1635), Blaise Pascal (1623-1662), Gottfried Wilhelm Leibniz (1646-1716), Samuel Morland (1625-1695) und Charles Babbage (1791-1871), der als Großvater heutiger Rechner gesehen werden kann, wollen wir hier nicht eingehen. Einen entsprechenden historischen Rückblick findet man zum Beispiel in [39].

Unser historischer Rückblick lehnt sich sehr eng an die technologischen Fortschritte an, die über die Zeit gemacht worden sind, beginnend bei einem rein mechanischen Rechner über mit Relais, Vakuumröhren oder Transistoren aufgebauten Rechnern bis

hin zu Mikroprozessoren, die durch integrierte Schaltungen realisiert sind.

1938-1941: Die Z1, Z2 und Z3 von Konrad Zuse

Das Computer-Zeitalter begann im Jahre 1938 mit den Arbeiten von Konrad Zuse (1910-1995), einem damals jungen Wissenschaftler aus Berlin. Die von ihm konzipierten Rechner unterschieden sich grundsätzlich von den bisher konstruierten oder angedachten Rechenmaschinen. Das vielleicht einschneidenste Merkmal der Arbeiten von Konrad Zuse ist die konsequente Anwendung des binären Zahlensystems, das die heutigen Rechner kennzeichnet und das, wie wir in diesem Buch auch sehen werden, einen entscheidenden Anteil an der Leistungsfähigkeit von heutigen Rechnern hat. Das zweite Merkmal der Zuse-Maschinen bestand darin, dass es im Gegensatz zu den früheren Rechenmaschinen eine klare Trennung zwischen dem Programm und den ausführenden Teilen gab, also eine klare Trennung der Komponenten zum Speichern und zum Rechnen, wie wir das auch heute noch kennen.

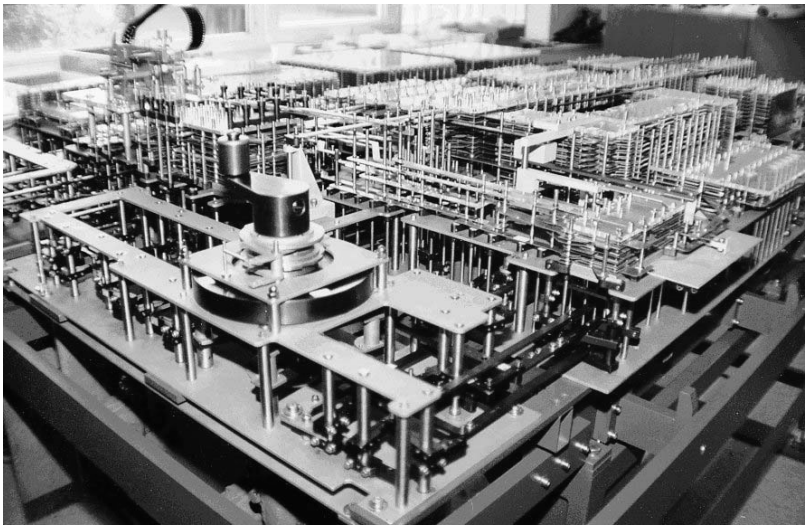


Abbildung 1.3: *Nachbau der 1938 von Konrad Zuse fertig gestellten Z1*
[http://www.alliedbytes.de/Info_Referat/, 2007]

Der erste von Zuse konstruierte Rechner, die Z1, wurde vollständig aus privaten Mitteln finanziert. Als Werkstatt diente das Wohnzimmer des damals 28-jährigen Zuse. Die Z1 war ein vollständig mechanischer, im Wesentlichen aus Blech bestehender Rechner mit einer Taktfrequenz von 1 Hertz [4]. Der Elektromotor zur Taktgebung verbrauchte circa 1.000 Watt. Die Z1 war eine halbe Tonne schwer. Sie gilt heute als erster frei programmierbarer Rechner der Welt (auch wenn es noch keine Speichermöglichkeit für

[4] *Hertz* ist die Einheit für die Frequenz. Sie gibt die Anzahl der Zyklen pro Sekunde an.

das Programm gab) und enthielt wie auch ihre Nachfolger Z2 und Z3 (siehe das Blockschaltbild in Abbildung 1.4) alle Bausteine eines modernen Computers. So verwenden die Zuse-Rechner einen Speicher, ein Steuerwerk (Leitwerk) und eine arithmetische Einheit für Gleitkommaarithmetik.

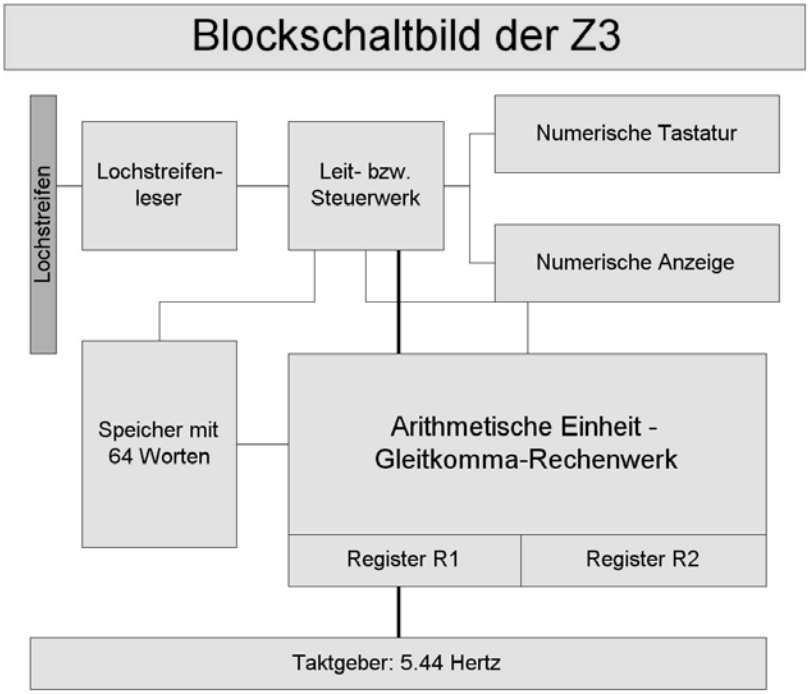


Abbildung 1.4: *Blockschaltbild der Z3*
[http://irb.cs.tu-berlin.de/~zuse/Konrad_Zuse/de/index.html, 2007]

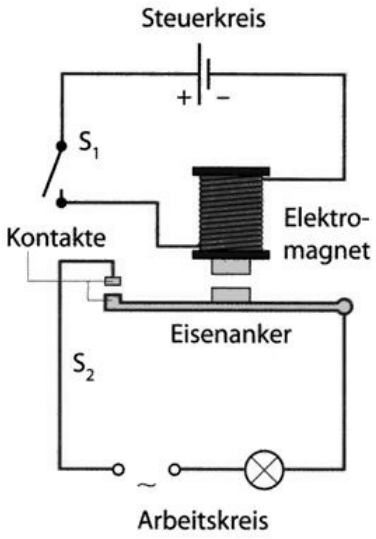
Die mechanische Konstruktion der Z1 war sehr aufwändig und recht fehleranfällig, sodass Zuse bereits vor Fertigstellung der Z1 mit dem Entwurf einer auf Relais basierenden arithmetischen Einheit und eines auf Relais basierenden Steuerwerkes begann. Die Z2, die Konrad Zuse 1939 fertig stellte, vereinigte diese beiden Module mit dem mechanisch arbeitenden Speicher der Z1. Hierdurch konnte er die Taktfrequenz seines Rechners auf 3 Hertz erhöhen.

Auf Grund der positiven Erfahrungen, die Zuse mit der Z2 gemacht hatte, konstruierte er die Z3, bei der nun auch der Speicher, der aus 64 Speicherzellen der Bitbreite 22 bestand, mit Relais aufgebaut war. Die Z3 bestand aus ungefähr 2.600 Relais – der Speicher war mit 1.400 Relais und die arithmetische Einheit und das Steuerwerk mit jeweils 600 Relais aufgebaut. Die Leistungsaufnahme war mit ungefähr 4.000 Watt recht hoch. Die Multiplikation erforderte 16 Takte bei einer Taktfrequenz von 5 bis 10 Hertz,

die Addition benötigte 3 Takte, was zu einer Laufzeit von ungefähr 3 Sekunden für eine Multiplikation und von circa 0, 6 Sekunden für eine Addition führte.

Exkurs: Das Relais ▷ ▷ ▷

Ein *Relais* ist ein Schalter, der nicht von Hand, sondern mithilfe eines Elektromagneten betätigt wird. Es besteht aus zwei getrennten Stromkreisen. Der erste Stromkreis wird als *Steuerstromkreis* und der zweite als *Arbeitsstromkreis* bezeichnet. Wird der Steuerstromkreis über einen Schalter (S_1) geschlossen, dann zieht der Elektromagnet (bestehend aus einer Spule mit Eisenkern) den Schalter (S_2) im Arbeitskreis an und der zweite Stromkreis ist ebenfalls geschlossen. Wird der erste Schalter S_1 geöffnet, dann lässt der Magnet den Schalter S_2 los, und der Arbeitsstromkreis ist unterbrochen. Das Relais bietet somit die Möglichkeit, mit kleinen Spannungen, z. B. Batteriespannungen, Stromkreise mit hohen Spannungen und Strömen zu steuern.



[http://www.laurentianum.de/physikmuseum, 2007]

◁ ◁ ◁

1946: Die ENIAC – Vakuumröhren anstelle von Relais

Das ENIAC-Projekt (***E**lectronic **N**umerical **I**ntegrator **A**nd **C**omputer*) stellt einen weiteren Meilenstein in der Geschichte der Computer dar. Es wurde von der *Moore School of Electrical Engineering* der *University of Pennsylvania* in Kooperation mit dem *U.S. Army Ordnance Department Ballistic Research Laboratory* zu ballistischen Untersuchungen kurz vor Ausbruch des Zweiten Weltkrieges ins Leben gerufen. John Mauchly (1907-1980), der am *Ursinus College* als Professor tätig war und während des Zweiten Weltkriegs Kurse in Elektrotechnik an der *Moore School* gab, überlegte sich 1942 in dem kurzen Aufsatz „*The Use of High Speed Vacuum Tube Devices for Calculating*“, dass ein Rechner durch Einsatz von Vakuumröhren im Vergleich zu einem auf Relais-Technik

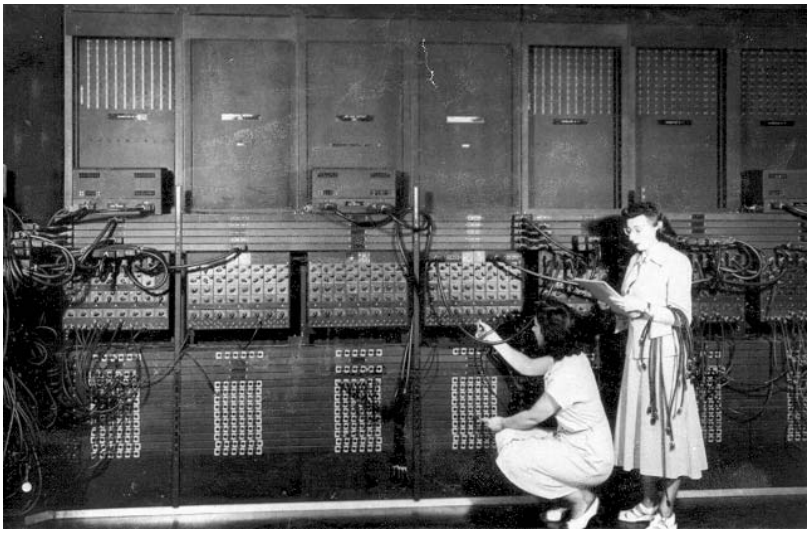
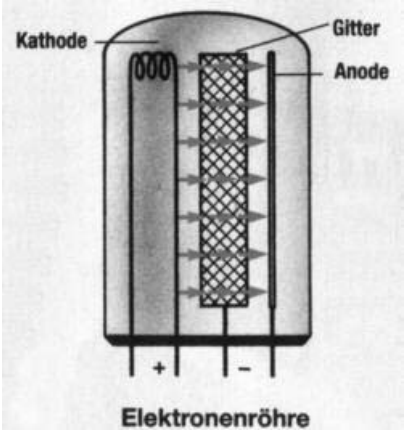


Abbildung 1.5: *Die in Philadelphia gebaute ENIAC (Ausschnitt). Die ENIAC konnte durch Verdrahten der einzelnen Komponenten programmiert werden.*
[http://www.yesterdaystomorrows.org/newimages/Eniac.jpg, 2004]

aufgebauten Rechner um Größenordnungen beschleunigt werden kann.

Exkurs: Die Vakuumröhre ▷ ▷ ▷

Bei einer *Vakuumröhre* ist zwischen einer Elektronen-abgebenden Kathode und einer Elektronen-aufnehmenden Anode ein Gitter. Durch das Spannungsgefälle zwischen der Kathode und der Anode werden Elektronen von der Kathode zur Anode emittiert. Strom fließt, wenn diese an der Anode ankommen. Über das Potential des Gitters kann der zwischen Kathode und Anode fließende Strom gesteuert werden.
Wie das Relais ist auch die Vakuumröhre ein Schalter. Die Vakuumröhre ist etwas kleiner, aber wesentlich schneller (bis zu einem Faktor 1.000) als ein Relais. Nachteil von Vakuumröhren sind der sehr hohe Stromverbrauch und die Anfälligkeit der Röhren, die im Mittel etwa eine Lebensdauer von zwei Jahren haben.



[http://www.alliedbytes.de/Info_Referat/, 2007]

◁ ◁ ◁

Basierend auf dieser Idee baute John Mauchly zusammen mit einem seiner Studen-

ten, Presper Eckert (1919-1995), die aus 17.468 Vakuumröhren bestehende ENIAC. Sie konnte eine Multiplikation (der Bitbreite 10) in einer Laufzeit von drei Millisekunden ausführen. Sie war also um einen Faktor 1.000 schneller als die Z3 von Zuse.

Die ENIAC war in gewissem Sinne ein „Monster“. Neben den 17.468 Vakuumröhren bestand sie aus ungefähr 1.500 Relais, 70.000 Widerständen und 10.000 Kondensatoren. Sie war ungefähr 24 Meter breit, drei Meter hoch und ein Meter tief, bei einem Gewicht von 30 Tonnen. Das durch die *U.S. Army* bewilligte Budget von ursprünglich 150.000 US$ erhöhte sich im Laufe der Arbeiten auf eine Gesamthöhe von 485.000 US$.

Eines der großen Probleme bei der ENIAC lag nicht nur in dem hohen Stromverbrauch, sondern auch in der mühsamen Programmierung – die Programmierung der ENIAC erfolgte über explizites Neuverdrahten der einzelnen Module – und in der Anfälligkeit der Vakuumröhren. Geht man von einer mittleren Lebensdauer einer Vakuumröhre von ungefähr zwei Jahren aus, so kann man sich leicht überlegen, dass bei der ENIAC mit ihren 17.468 Röhren rein rechnerisch jede Stunde eine Röhre ausfiel.

Der Nachfolger der ENIAC, die UNIVAC, die im Jahre 1951 durch die Firma *Remington Rand* fertig gestellt wurde, war mit 46 verkauften Maschinen zu einem Kaufpreis von mehr als 1.000.000 US$ der erste kommerzielle Rechner.

1952: Die IAS, der von-Neumann-Rechner

Schon während dem Bau der ENIAC war den Entwicklern klar, dass nicht nur die Daten, mit denen gerechnet wurde, sondern auch die Programme im Rechner gespeichert werden müssten. So äußerte John Mauchly 1948, dass *Rechnungen nur dann bei hoher Geschwindigkeit ausgeführt werden können, wenn der Maschine auch mit hoher Geschwindigkeit Befehle erteilt werden können.* Ideen für ein Programmspeicher-Prinzip, wie zum Beispiel die, dass Befehle und numerische Daten in gleicher Weise im Speicher abgelegt werden sollten, wurden während des ENIAC-Projektes nur rudimentär formuliert. Ihnen wurde erst durch John von Neumann (1903-1957), einem in Ungarn geborenen und am *Institute for Advanced Study* an der *Princeton University* arbeitenden Mathematiker, zum Durchbruch verholfen.

Von wirklich zentraler Bedeutung für den Rechnerbau waren die Arbeiten von John von Neumann zu den logischen Grundlagen seines in Princeton entwickelten IAS-Rechners, der 1952 der Öffentlichkeit vorgestellt wurde. Dieser unter *von-Neumann-Architektur* bekannte logische Aufbau ist während der letzten 50 Jahre von wenigen Ausnahmen abgesehen unverändert für Rechner benutzt worden. Die wesentlichen Punkte des von-Neumann-Prinzips, die zum Teil bereits in den Rechnern von Konrad Zuse zu finden waren, sind die folgenden:

- *Trennung zwischen Datenpfad und Speicherkomponenten.* Dies hatte eher technische Gründe, da dadurch der Entwurf sowohl von den Speicherzellen als auch von dem Datenpfad einfacher wurde;
- *Interne Speicherung der Programme in einer Speicherkomponente*, da durch Verwendung eines schnellen Speichers der Prozessor ohne große Wartezeiten auf die nächsten abzuarbeitenden Befehle zugreifen kann;

Abbildung 1.6: *Der am **I**nstitute for **A**dvanced **S**tudies in Princeton entwickelte IAS-Rechner. Auf dem Bild sind John von Neumann (links) zusammen mit dem damaligen Direktor des Instituts, Robert Oppenheimer, zu sehen.*
[http://www.admin.ias.edu/pr/images/winter03cover.jpg, 2004]

- *Gemeinsamer Speicher für Daten und Befehle*, da sich die Größen der Programme und der benötigte Speicherplatz für Daten von Programm zu Programm ändern können;
- *Fetch-Decode-Execute-Arbeitszyklus*, d. h. die grundsätzliche Arbeitsweise setzt sich aus drei Phasen zusammen. In einem ersten Schritt, der *Fetch*-Phase, wird der nächste abzuarbeitende Befehl aus der Speicherkomponente geholt. Der eingelesene Befehl wird in der *Decode*-Phase dekodiert und schließlich in der *Execute*-Phase ausgeführt.

1953: Die TRADIC – die Transistoren halten Einzug

TRADIC ist die Abkürzung von ***TRA**nsistor **DI**gital **C**omputer*. Wie der Name es schon andeutet, handelt es sich bei der TRADIC von *AT&T Bell Labs* um den ersten Rechner, der nur Transistoren und Dioden und keine Vakuumröhren mehr benutzte. Die

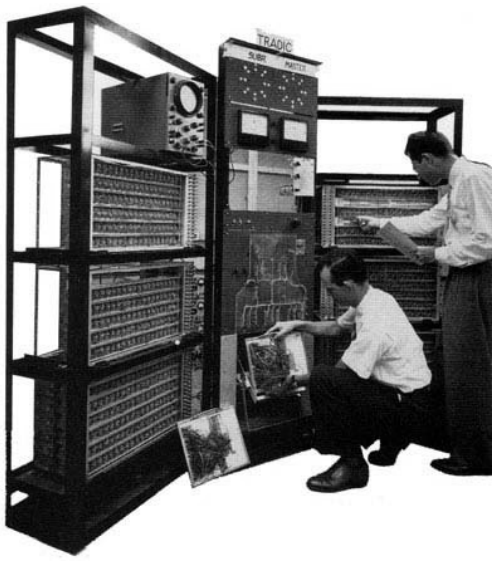
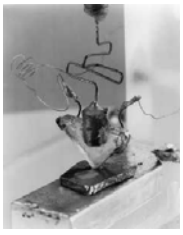


Abbildung 1.7: *Die TRADIC von AT&T Bell Labs*
[http://www.cedmagic.com/history/tradic-transistorized.html, 2007]

TRADIC wurde im Auftrag der *U.S. Air Force* entwickelt. Sie bestand aus ungefähr 700 Transistoren und etwa 10.000 Dioden. Die Leistungsaufnahme war mit knapp 100 Watt um ein Vielfaches (Faktor 10 bis 12) kleiner als die der mit Vakuumröhren arbeitenden Maschinen.

Exkurs: Der Transistor ▷ ▷ ▷

Der *Transistor* wurde in den *Bell Laboratories* von John Bardeen (1908-1991), Walter Brattain (1902-1987) und William Shockley (1910-1989) erfunden – die Abbildung rechts zeigt den ersten Transistor. Die Entdeckung resultierte im Wesentlichen aus den Anstrengungen, bessere Verstärker herzustellen und einen Ersatz für mechanische Relais und Vakuumröhren zu finden.



[http://www.digicamhistory.com/, 2007]

Ein Transistor ist ein elektronisches Bauelement mit drei Anschlüssen. Über einen dieser Anschlüsse, das *Gate*, kann der Strom zwischen den anderen beiden Anschlüssen, die *Drain* oder *Senke* beziehungsweise *Source* oder *Quelle* genannt werden, gesteuert werden, sofern diese beiden Anschlüsse auf unterschiedlichen Potentialen liegen.

Transistoren können sehr kompakt auf Siliziumchips gebaut werden. Mehrere Millionen Transistoren passen heutzutage auf eine Fläche von 1 cm^2 – seit 1960 hat sich die Anzahl der Transistoren, die auf einem Quadratzentimeter Silizium integriert werden können, ungefähr alle 18 Monate verdoppelt. Die Schaltgeschwindigkeit liegt inzwischen im Nanosekundenbereich.

◁ ◁ ◁

1971: Der Intel 4004, der erste Mikroprozessor

Computers in the future may weigh no more than 1,5 tons.
[Popular Mechanics 1949]

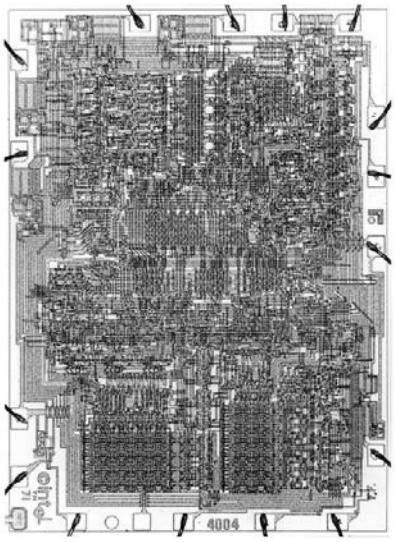


Abbildung 1.8: *Intel-4004-Prozessor*
[http://www-vlsi.stanford.edu/group/chips_micropro_body.html, 2007]

Der erste Prozessor, der auf einem einzigen Chip integriert war, ist der *Intel 4004*.[5] Die Entwicklung des *Intel 4004* geht auf einen Auftrag der Firma *Busicom* aus Japan an die damals noch sehr kleine Firma *Intel Corporation* zurück. *Busicom* verkaufte zur damaligen Zeit Tischrechner und beauftragte *Intel Corporation*, eine integrierte Schaltung mit der für einen Tischrechner notwendigen Funktionalität zu entwerfen und zu

[5] Jack Kilby (*1923), Mitarbeiter von *Texas Instruments*, baute 1958 den ersten integrierten Schaltkreis aus Silizium und zeigte damit, dass Widerstände, Kondensatoren und Transistoren auf dem gleichen Stück Halbleiter nebeneinander funktionieren können. Seine Schaltung bestand aus fünf untereinander verbundenen Komponenten.

fertigen. Ted Hoff, der sich bei *Intel Corporation* der Aufgabe annahm, ignorierte die von *Busicom* bereitgestellte Spezifikation und begann stattdessen mit der Entwicklung eines *general purpose*-Mikroprozessors, d. h. eines universellen Rechners, der über Software auf die Bedürfnisse des Kunden einstellbar ist und somit einen wesentlich größeren Markt erreichen konnte. Der von Ted Hoff gemachte Entwurf wurde von Federico Faggin zu dem ersten, auf einem Chip integrierten Mikroprozessor umgesetzt. Es handelt sich um einen 4-Bit-Prozessor, d. h. die von ihm verarbeiteten Daten hatten eine Wortbreite von 4 Bit mit der Besonderheit, dass Maschinenbefehle Wortbreite 8 Bit hatten – was bedeutete, dass Programm und Daten in unterschiedlichen Speichern abgelegt wurden. Die Maschinensprache des *Intel 4004* bestand aus 46 Befehlen. Der Mikroprozessor war aus 2.300 Transistoren aufgebaut und lief mit einer Taktfrequenz von 108 kHz.

1986: Der MIPS R2000, Beginn des RISC-Zeitalters

Der erste RISC (***R**educed **I**nstruction **S**et **C**omputer*) wurde 1986 der Öffentlichkeit vorgestellt. Es handelte sich um den MIPS R2000, der aus dem MIPS-Projekt der *Stanford University* hervorging. Die Abkürzung MIPS steht für ***M**icroprocessor without **I**nterlocked **P**ipeline **S**tages*.

Der Grundgedanke von RISC ist, dass man versucht, die Rechner zu beschleunigen, indem man in die Maschinensprache nur sehr einfache Maschinenbefehle aufnimmt, die alle ungefähr die gleiche Ausführungszeit haben. Hierdurch gewährleistet man, dass eine Verarbeitung der Maschinenprogramme unter Verwendung einer Befehlspipeline effizient möglich wird.

Die RISC-Philosophie hat zunächst Eingang im Bereich der wissenschaftlichen Arbeitsplatzrechner gefunden. Sie ist heute in allen Bereichen anzutreffen.

Wir wollen es bei diesen ersten Anmerkungen zu RISC belassen. Im weiteren Verlauf des Lehrbuches werden wir noch detailliert auf RISC eingehen (siehe Teil III).

2000: Der Intel Pentium 4

I think there is a world market for maybe five computers.
[Thomas Watson, *IBM Corporation*, 1943]

Wir schließen unseren Rundgang durch die Geschichte mit dem *Intel Pentium 4* ab. Wir wollen nicht auf Details eingehen, sondern nur zwei Kennzahlen[6] nennen und diese denen des oben schon kurz vorgestellten *Intel 4004* gegenüberstellen.

- Der *Intel Pentium 4* bestand ursprünglich aus 42.000.000 Transistoren, der Intel 4004 aus 2.300 Transistoren.
- Die Taktfrequenz des *Intel Pentium 4* lag bei 1,5 GHz, die des *Intel 4004* bei 108 KHz.

[6] Die Kennzahlen der heute verfügbaren Prozessoren haben sich weiter erhöht. So ist z. B. der Montecito Chip von Intel aus mehr als einer Milliarde Transistoren aufgebaut (1,72 Milliarden Transistoren).

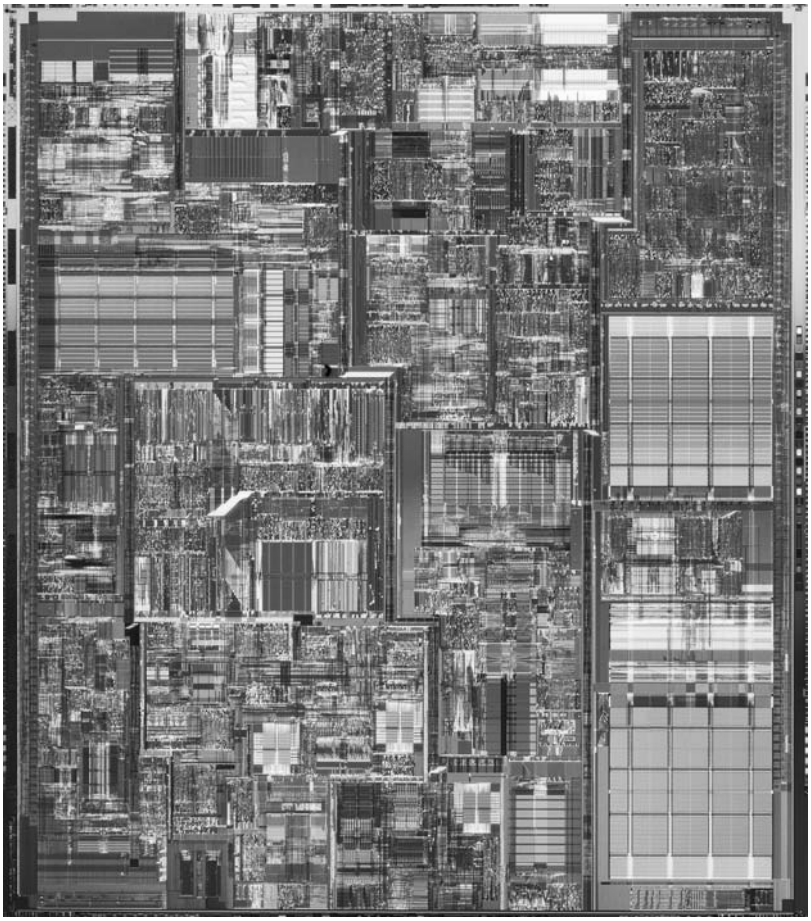


Abbildung 1.9: *Intel-Pentium-4-Prozessor*
[http://www-vlsi.stanford.edu/group/chips_micropro_body.html, 2007]

Der Vergleich zeigt eindrucksvoll, wie rasant sich die Technologie in den letzten 30 Jahren weiterentwickelt hat. Um ein Gefühl dafür zu bekommen, wie gewaltig die seit dem *Intel 4004* gemachten Fortschritte wirklich sind, wird oft ein Vergleich mit der Automobilindustrie gezogen:

Hätte sich die Geschwindigkeit der Kraftfahrzeuge in ähnlicher Art und Weise erhöht, wie dies bei der Taktfrequenz der Mikroprozessoren erfolgt ist, so könnte man heute in gerade mal 13 *Sekunden mit dem Auto von San Francisco nach New York gelangen.*

Teil I

Grundlagen

Im ersten Teil dieses Buches wollen wir uns mit verschiedenen Grundlagen beschäftigen, deren Kenntnis eine notwendige Voraussetzung für das Verständnis der späteren, weiterführenden Betrachtungen ist. Auch wenn man die Inhalte dieser Kapitel auf den ersten Blick statt der Technischen Informatik zunächst eher den Bereichen der Mathematik und Elektrotechnik zuordnen würde, so spielen diese bei eingehender Betrachtung dennoch auch in unserem Kontext eine wichtige Rolle. So hätten wir beispielsweise sicherlich Probleme, den Aufbau eines Addierer-Schaltkreises oder gar die Funktionsweise einer ALU zu verstehen, wenn wir nichts über Stellenwertsysteme und insbesondere über das Binärsystem wüssten.

Aus diesem Grund führen wir in Kapitel 2 Boolesche Algebren, Boolesche Funktionen sowie Boolesche Ausdrücke ein und stellen damit einen wichtigen „Werkzeugkasten“ bereit, auf den wir später ständig zurückgreifen werden. Ziel des anschließenden Kapitels 3 ist es, zu beschreiben und zu diskutieren, auf welche Weise es möglich ist, Zeichen und vor allem Zahlen im Rechner zu repräsentieren, und welche Vor- und Nachteile die einzelnen Darstellungen aufweisen.

In den Kapiteln 4 und 5 geben wir eine Einführung in die elektronischen Grundlagen von Schaltungen, die notwendig ist, um verstehen zu können, wie Signale in einem Rechner auf der physikalischen Ebene verarbeitet werden. Insbesondere ist die Welt, in der wir leben, zu großen Teilen eine analoge Welt. Wollen wir zum Beispiel die Temperatur, also einen analogen physikalischen Wert, messen, um sie durch einen (digitalen) Rechner verarbeiten zu lassen, so muss der analoge Wert in einen digitalen Wert gewandelt werden. Wie erfolgt diese Umwandlung? Solche und ähnliche Fragen wollen wir in diesem Teil behandeln. Hierbei werden wir die Funktionsweisen der einzelnen Bauelemente und elementarer Schaltungen nicht bis ins letzte Detail diskutieren. Dies würde den Rahmen des Buches bei weitem sprengen. Im Vordergrund wird das prinzipielle Verständnis des Aufbaus und der Funktionsweise elektronischer Schaltungen stehen.

2 Grundlegende mathematische Begriffe

Eine der wichtigsten Abstraktionsebenen eines Rechners ist die in Kapitel 1 eingeführte Hardware-Ebene. In einer feineren Einteilung kann diese in weitere Unterebenen gegliedert werden. Eine dieser Unterebenen ist die *digitale Ebene*, auf der ausschließlich die beiden Signale 0 und 1 unterschieden und verarbeitet werden. Es wird also von elektronischen Eigenschaften, Spannungspegeln, Anstiegszeiten und dergleichen abstrahiert – stattdessen wird ein Signal als *binäre Informations-Einheit* interpretiert. Trotzdem handelt es sich bei dieser Sichtweise um eine relativ niedrige Abstraktionsebene, die für den Entwurf von Rechnern allerdings eine ganz wichtige Rolle spielt. Oftmals besteht die Entwurfsaufgabe genau darin, komplexere Bausteine wie beispielsweise Steuerwerke, über die auf höheren Beschreibungsebenen (System-Ebene, Register-Transfer-Ebene) gesprochen wird, einzig und allein mithilfe der auf der digitalen Ebene zur Verfügung stehenden Mittel und Methoden zu entwerfen.

Um solche Aufgaben angemessen beschreiben und lösen zu können, bedarf es einer formalen mathematischen Basis. Da statt des gewohnten Dezimalsystems nur die Ziffern 0 und 1 zur Verfügung stehen, weist dieses Fundament gegenüber dem Körper der reellen Zahlen, in dem wir uns bei den Rechenproblemen des Alltags üblicherweise bewegen, zwangsläufig gravierende Unterschiede auf. In diesem Grundlagenkapitel wollen wir uns mit diesen zunächst etwas ungewohnten Gesetzmäßigkeiten vertraut machen, um diese dann in späteren Teilen des Buches beim Entwurf digitaler Schaltungen in geeigneter Weise einzusetzen.

In Abschnitt 2.1 wird mit der *Booleschen Algebra* sozusagen der als Grundausstattung für weitere Aufgaben benötigte „Werkzeugkasten“ zur Verfügung gestellt. Auch wenn sich die Notwendigkeit der formalen Einführung dieses mathematischen Gebildes nicht auf den ersten Blick erschließt, so wird doch bereits beim Durcharbeiten der folgenden Abschnitte 2.2 und 2.3 rasch klar, dass die Boolesche Algebra als Fundament notwendig ist, um mit den direkt anschließend eingeführten *Booleschen Funktionen* bzw. mit einer ihrer Darstellungsformen, den *Booleschen Ausdrücken*, umgehen zu können.

2.1 Boolesche Algebra

Eine Boolesche Algebra ist eine spezielle algebraische Struktur, die nach dem englischen Mathematiker und Philosophen George Boole benannt ist, der sie 1847 erstmals vorstellte. Wir beginnen unsere Diskussion mit der formalen Definition einer Booleschen Algebra und betrachten anschließend Beispiele, die teilweise für das weitere Vorgehen wichtig sind und überdies dem besseren Verständnis dienen.

Definition 2.1: *Boolesche Algebra*

Sei M eine nicht leere Menge, auf der die zwei binären Operatoren $\cdot$ und $+$ sowie der unäre Operator $\sim$ definiert sind, d. h.:

$$
\begin{aligned}
+ &: M \times M \to M \\
\cdot &: M \times M \to M \\
\sim &: M \to M
\end{aligned}
$$

Das Tupel $(M, +, \cdot, \sim)$ heißt *Boolesche Algebra (BA)*, wenn für beliebige $x, y, z \in M$ die folgenden *Axiome* gelten:

$$
\begin{aligned}
&\textit{Kommutativität:} && x \cdot y = y \cdot x \\
& && x + y = y + x \\
&\textit{Assoziativität:} && (x \cdot y) \cdot z = x \cdot (y \cdot z) \\
& && (x + y) + z = x + (y + z) \\
&\textit{Absorption:} && (x \cdot y) + x = x \\
& && (x + y) \cdot x = x \\
&\textit{Distributivität:} && x \cdot (y + z) = (x \cdot y) + (x \cdot z) \\
& && x + (y \cdot z) = (x + y) \cdot (x + z) \\
&\textit{Auslöschung:} && x + (y \cdot (\sim y)) = x \\
& && x \cdot (y + (\sim y)) = x
\end{aligned}
$$

Durch die Axiome haben wir nun also ein Grundgerüst an Rechenregeln, die in der Booleschen Algebra grundsätzlich immer gelten und bei Rechnungen/Umformungen ohne weiteres verwendet werden dürfen. Einige davon wie z. B. die Kommutativität sind vom Dezimalsystem her bekannt, andere wie die Absorption gelten im Körper der reellen Zahlen nicht. Aus den als Axiome gegebenen Regeln lassen sich noch weitere Gesetze herleiten, die sich im praktischen Umgang mit konkreten Booleschen Algebren ebenfalls als sehr nützlich erweisen.

Lemma 2.1: *Gesetze der Booleschen Algebra*

Sei $(M, +, \cdot, \sim)$ eine Boolesche Algebra und $x, z, y \in M$ beliebig. Dann gelten die folgenden Gesetze:

$$
\begin{aligned}
&\textit{Neutrale Elemente}: && \exists n \in M : x + n = x \textit{ und } x \cdot n = n \\
& && \exists e \in M : x \cdot e = x \textit{ und } x + e = e
\end{aligned}
$$

Allgemein wird n auch als *Nullelement* (0) und e als *Einselement* (1) bezeichnet.

Idempotenz :	$x + x = x$
	$x \cdot x = x$
Regeln von de Morgan :	$\sim (x + y) = (\sim x) \cdot (\sim y)$
	$\sim (x \cdot y) = (\sim x) + (\sim y)$
Consensus :	$(x_1 \cdot x_2) + ((\sim x_1) \cdot x_3)$
	$\quad = (x_1 \cdot x_2) + ((\sim x_1) \cdot x_3) + (x_2 \cdot x_3)$
	$(x_1 + x_2) \cdot ((\sim x_1) + x_3)$
	$\quad = (x_1 + x_2) \cdot ((\sim x_1) + x_3) \cdot (x_2 + x_3)$

Beweis: siehe Übungen. ◁

Um herauszufinden, ob eine Menge mit entsprechenden Operatoren eine Boolesche Algebra darstellt, muss bewiesen werden, dass die Axiome aus Definiton 2.1 für den Verbund allgemeine Gültigkeit besitzen. Ist der Nachweis geglückt, so können beim Rechnen im jeweiligen Verbund automatisch auch die Regeln aus Lemma 2.1 verwendet werden, was natürlich sehr hilfreich sein kann.

Um aber überhaupt ein Gefühl für Boolesche Algebren und die für sie geltenden Gesetze bekommen zu können, ist es wichtig, konkrete Beispiele zu betrachten, was wir im Folgenden auch tun wollen.

Beispiel 2.1: *Boolesche Algebra der Teilmengen*

Sei S eine beliebige, nichtleere Menge. Die Potenzmenge von S sei durch 2^S bezeichnet. Weiterhin seien $\cup$ und $\cap$ die aus der Mengenlehre bekannten Vereinigungs- bzw. Schnittoperatoren und

$$\sim : 2^S \to 2^S,$$
$$M \to S \setminus M$$

die Abbildung, die jeder beliebigen Teilmenge M von S ihr Komplement bezüglich S zuordnet. Dann ist

$$BA_1 := (2^S, \cup, \cap, \sim)$$

eine Boolesche Algebra. Sie wird auch als *Boolesche Algebra der Teilmengen* bezeichnet.

Beispiel 2.2: *Zweielementige Boolesche Algebra*

Es sei $\mathbb{B} := \{0, 1\}$. Weiterhin seien folgende Operatoren definiert:

$$\begin{aligned} +_b &: \mathbb{B} \times \mathbb{B} \to \mathbb{B} \text{ mit } x +_b y := x + y - x \cdot y \\ \cdot_b &: \mathbb{B} \times \mathbb{B} \to \mathbb{B} \text{ mit } x \cdot_b y := x \cdot y \\ \sim &: \mathbb{B} \to \mathbb{B} \quad \text{mit } \sim x := 1 - x \end{aligned}$$

Dann ist

$$BA_2 := (\mathbb{B}, +_b, \cdot_b, \sim)$$

eine Boolesche Algebra. Sie wird auch als *zweielementige Boolesche Algebra* bezeichnet. Die Operatoren $+_b$ und $\cdot_b$ wurden hier mit einem indizierten b versehen, um Verwechslungen mit den in der Definition ebenfalls vorkommenden Additions- bzw. Multiplikationsoperatoren der reellen Zahlen zu vermeiden. Zukünftig wird der Index weggelassen, wann immer Mehrdeutigkeiten ausgeschlossen sind.

Während die Boolesche Algebra BA_1 der Teilmengen vor allem demonstriert, dass sich das abstrakte Gerüst einer Booleschen Algebra mit konkreten Strukturen „füllen“ lässt, die weit von dem entfernt sind, was wohl den meisten Lesern bei diesem Begriff intuitiv in den Sinn kommt, entpuppt sich BA_2 bei genauerem Hinsehen als alte Bekannte. Interpretiert man die Werte 0 und 1 als *falsch* und *wahr*, so stellen die Operatoren $+_b$ und $\cdot_b$ die logischen *ODER*- und *UND*-Verknüpfungen dar, während $\sim$ dem Negations-Operator entspricht. Insgesamt ist BA_2 dann nichts anderes als die bereits aus der Schulzeit bekannte Aussagenlogik.

Mit dieser Interpretation lässt sich nun auch endlich intuitiv der Sinn der Gesetze der Booleschen Algebra erschließen. Man versteht nun zum Beispiel, warum anders als beim Körper der reellen Zahlen $x+x = x$ gilt: Das liegt daran, dass die Aussage „x *ODER* x“ genau dann gilt, wenn x mit dem Wert *wahr* belegt ist. Somit liefert also die Struktur der Booleschen Algebra die für die Aussagenlogik bekannten Rechenregeln.

In späteren Kapiteln dieses Buches wird die Boolesche Algebra der Teilmengen kaum weiter von Bedeutung sein, jedoch spielt die zweielementige Boolesche Algebra an zahlreichen Stellen eine wichtige Rolle. Eine weitere in unserem Kontext sehr wichtige Boolesche Algebra, nämlich die Boolesche Algebra der Funktionen, werden wir in Abschnitt 2.2 kennen lernen. Vorher soll aber noch auf einen ebenfalls interessanten Aspekt der Booleschen Algebra eingegangen werden: das Dualitätsprinzip der Operatoren + und ·. Durch dieses lässt sich aus einer gültigen Gleichung unmittelbar eine zweite gültige Gleichung herleiten.

Definition 2.2: *Duale Gleichung*

Sei p eine aus den Gesetzen einer Booleschen Algebra abgeleitete Gleichung. Die *duale Gleichung* p' erhält man aus p durch gleichzeitiges Vertauschen von + und · sowie 0 und 1.

Lemma 2.2: *Prinzip der Dualität*

Sei p eine aus den Gesetzen einer Booleschen Algebra abgeleitete Gleichung. Gilt p, so gilt auch die zu p duale Gleichung p'.

Beweis: Für jedes Axiom a gilt: Auch die duale Gleichung a' ist ein Axiom. Da jede allgemeingültige Gleichung direkt oder indirekt aus den Axiomen hergeleitet ist, lässt sich die duale Gleichung auf die gleiche Art und Weise durch Verwendung der dualen Axiome herleiten. ◁

2.2 Boolesche Funktionen

Wie bereits zuvor erwähnt, werden durch die Boolesche Algebra Werkzeuge bereitgestellt, die beim Entwurf von Rechnern sehr wichtig und hilfreich sind. Doch was genau ist denn nun eine Entwurfsaufgabe bzw. wie lässt sich diese formal korrekt beschreiben?

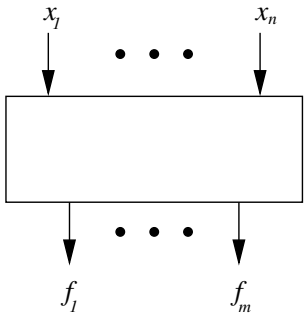


Abbildung 2.1: *Ausgangssituation beim Schaltkreisentwurf*

Für den Moment genügt es davon auszugehen, dass für einen geplanten Entwurf eine Ausgangssituation vorliegt, wie sie in Abbildung 2.1 skizziert ist. Die zu entwerfende Komponente verfügt über eine vorgegebene Menge von Ein- und Ausgängen, wobei die Eingänge jeweils mit binären Signalen (also 0 oder 1) belegt werden können. Abhängig von diesen Eingangssignalen sollen bestimmte Ausgangssignale generiert werden, die wiederum aus der Menge $\mathbb{B} = \{0, 1\}$ stammen. Mit anderen Worten: Die Komponente soll jeder möglichen Kombination von Eingangssignalen eine bestimmte Kombination von Ausgangssignalen zuordnen. Das Ziel des Entwurfs kann also als Realisierung einer Abbildung einer Menge binärer Signale auf eine andere Menge ebenfalls binärer Signale angesehen werden. Eine solche Abbildung bezeichnen wir als *Boolesche Funktion*.

Definition 2.3: *Boolesche Funktionen*

Es sei wie gewohnt $\mathbb{B} = \{0, 1\}$ und $D \subseteq \mathbb{B}^n$ beliebig. Dann definieren wir:

1. Eine Abbildung $f : \mathbb{B}^n \to \mathbb{B}^m$ heißt *(totale* bzw. *vollständig definierte) Boolesche Funktion* in n Variablen mit m Ausgängen. $\mathbb{B}_{n,m} := \{ f ;\ f : \mathbb{B}^n \to \mathbb{B}^m \}$ bezeichnet die *Menge aller totalen Booleschen Funktionen* von $\mathbb{B}^n$ nach $\mathbb{B}^m$.

2. Eine Abbildung $f : D \to \mathbb{B}^m$ heißt *partielle Boolesche Funktion* über D. Die *Menge aller partiellen Booleschen Funktionen über D* wird durch $\mathbb{B}_{n,m}(D) = \{ f ;\ f : D \to \mathbb{B}^m \}$ bezeichnet.

Von besonderer Bedeutung sind zunächst Funktionen mit genau einem Ausgang, weshalb im entsprechenden Sonderfall $m = 1$ statt $\mathbb{B}_{n,1}$ oft auch abkürzend einfach $\mathbb{B}_n$ geschrieben wird. Außerdem ist es sinnvoll, über bestimmte Bild- und Urbildmengen reden zu können, weshalb wir uns für das Weitere auf folgende Schreibweisen verständigen wollen:

- Für $f \in \mathbb{B}_n$ bezeichnet
  - $ON(f) := \{ x \in \mathbb{B}^n ;\ f(x) = 1 \}$ die *Erfüllbarkeitsmenge* von f,
  - $OFF(f) := \{ x \in \mathbb{B}^n ;\ f(x) = 0 \}$ die *Nichterfüllbarkeitsmenge* von f.
- Für $D \subseteq \mathbb{B}^n$ und $f \in \mathbb{B}_n(D)$ bezeichnet
  - $ON(f) := \{ x \in D ;\ f(x) = 1 \}$ die Erfüllbarkeitsmenge von f
  - $OFF(f) := \{ x \in D ;\ f(x) = 0 \}$ die Nichterfüllbarkeitsmenge von f
  - $DEF(f) := D$ den *Definitionsbereich* von f,
  - $DC(f) := \{ x \in \mathbb{B}^n ;\ x \notin D \}$ den *Don't-Care-Bereich* von f.

Eine totale Boolesche Funktion $f \in \mathbb{B}_{n,m}$ ist bereits allein durch ihre Erfüllbarkeitsmenge vollständig bestimmt. Da $ON(f) \cup OFF(f) = \mathbb{B}^n$, gilt dies natürlich genauso für die Nichterfüllbarkeitsmenge. Bei einer partiellen Booleschen Funktion $g \in \mathbb{B}_{n,m}(D)$ mit $D \subseteq \mathbb{B}^n$ benötigt man zwei der drei Mengen $ON(g)$, $OFF(g)$ und $DC(g)$, da $\mathbb{B}^n$ in diesem Fall die disjunkte Vereinigung aus Erfüllbarkeitsmenge, Nichterfüllbarkeitsmenge und Don't-Care-Bereich darstellt. Darüber hinaus gilt dann auch noch die Beziehung $ON(g) \cup OFF(g) = DEF(g)$.

Bevor wir mit unserer Untersuchung Boolescher Funktionen fortfahren, betrachten wir nun ein Beispiel.

Beispiel 2.3: *Repräsentation einer Booleschen Funktion durch eine Tabelle*

Es sei $f \in \mathbb{B}_{2,4}$, d. h. f besitzt zwei Eingänge und vier Ausgänge. Beide Eingänge, für die wir die Bezeichnungen x_1 und x_2 wählen, können die Werte 0 oder 1 annehmen, wodurch sich $2^2 = 4$ mögliche Eingangskombinationen ergeben. Jeder Eingangskombination muss nun für jeden der Ausgänge, welche wir mit f_1, f_2 ,f_3, f_4 bezeichnen, eine Belegung aus $\mathbb{B}$ zugewiesen werden. Dies lässt sich durch eine Tabelle bewerkstelligen, wie sie im Folgenden gegeben ist:

x_1	x_2	f_1	f_2	f_3	f_4
0	0	0	0	0	0
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	1	0	1

Werden die einzelnen Ausgänge unabhängig voneinander betrachtet, dann kann ganz allgemein festgestellt werden, dass eine Funktion aus der Menge $\mathbb{B}_{n,m}$ nichts anderes ist als m einzelne Funktionen aus $\mathbb{B}_n$, also mit jeweils nur einem Ausgang. In diesem Sinne stellt f hier vier einzelne Funktionen aus $\mathbb{B}_2$ dar. Man kann sich leicht davon überzeugen, dass f_1 der *Und-Funktion*, f_2 der *Oder-Funktion* und f_3 der *Exklusiv-Oder-Funktion* entspricht. Die Funktion f_4 hingegen stellt die sogenannte *Projektionsfunktion nach der Booleschen Variablen* x_1 dar, denn der Funktionswert ist in jedem Fall mit dem Wert des Eingangs x_1 identisch. Eine solche Projektionsfunktion nach einer Variablen x_i wird meist abkürzend und vereinfachend durch das Symbol x_i selbst bezeichnet.

Als Nächstes wenden wir uns dem Vergleichen und Verknüpfen von Booleschen Funktionen zu. Offensichtlich sind zwei Boolesche Funktionen $f, g \in \mathbb{B}_{n,m}$ genau dann als gleich anzusehen, wenn sie sich für jede Belegung der Booleschen Variablen „gleich verhalten“. Formal ausgedrückt bedeutet dies:

$$f = g \quad \Leftrightarrow \quad \forall x \in \mathbb{B}^n : f(x) = g(x)$$

Neben der Gleichheit kann analog auch der Vergleichsoperator „$\leq$“ auf Funktionen definiert werden:

$$f \leq g \quad \Leftrightarrow \quad \forall x \in \mathbb{B}^n : f(x) \leq g(x)$$

Boolesche Funktionen lassen sich aber nicht nur vergleichen, sondern mithilfe von geeigneten Operatoren auch verknüpfen. Dies ist z. B. dann von großer Bedeutung, wenn ein Entwurf hierarchisch durchgeführt wird und infolgedessen mehrere bereits entworfene und formal durch Boolesche Funktionen repräsentierte Komponenten zu weiteren Komponenten verknüpft oder zusammengefasst werden. Wir beschränken uns hierbei auf Funktionen aus $\mathbb{B}_n$ und führen die bereits aus Abschnitt 2.1 bekannten Operatoren nun für Boolesche Funktionen ein:

$$\begin{aligned}
+ &: \mathbb{B}_n \times \mathbb{B}_n \to \mathbb{B}_n \text{ mit } (f+g)(x) := f(x) +_b g(x) && \forall x \in \mathbb{B}_n \\
\cdot &: \mathbb{B}_n \times \mathbb{B}_n \to \mathbb{B}_n \text{ mit } (f \cdot g)(x) := f(x) \cdot_b g(x) && \forall x \in \mathbb{B}_n \\
\sim &: \mathbb{B}_n \to \mathbb{B}_n \text{ mit } (\sim f)(x) :=\sim f(x) && \forall x \in \mathbb{B}_n
\end{aligned}$$

Dabei bezeichnen f und g beliebige Boolesche Funktionen aus $\mathbb{B}_n$ und $+_b$, $\cdot_b$ sowie $\sim$ die Operatoren aus der zweielementigen Booleschen Algebra BA_2. Mit diesen Operatoren lässt sich nun die bereits in Abschnitt 2.1 angesprochene Boolesche Algebra der Funktionen gewinnen.

Satz 2.1: *Boolesche Algebra der Booleschen Funktionen in n Variablen*

Durch $(\mathbb{B}_n, +, \cdot, \sim)$ ist eine Boolesche Algebra gegeben.

Beweis: Durch Zurückführen der Operatoren der Booleschen Algebra der Funktionen auf die Operatoren von BA_2. ◁

Die Erkenntnis, dass die Booleschen Funktionen zusammen mit den oben eingeführten Operatoren eine Boolesche Algebra bilden, versetzt uns in die Lage, alle für Boolesche Algebren zur Verfügung stehenden Gesetze zu verwenden und somit letzten Endes mit Funktionen genauso zu rechnen wie mit Booleschen Variablen. Dies stellt eine enorme Erleichterung dar, und wir werden auch immer und immer wieder Gebrauch davon machen.

Bevor aber wirklich mit Booleschen Funktionen „gerechnet“ werden kann, muss man sich Gedanken um die Darstellung bzw. die Beschreibung von Funktionen machen. Bisher wurden die Darstellungen durch die Erfüllbarkeits- bzw. die Nichterfüllbarkeitsmenge angesprochen sowie anhand eines Beispiels die Darstellung in Tabellenform vorgeführt. Schon hier lässt sich leicht erkennen, dass die Vorgehensweise und der rechnerische Aufwand bei der Berechnung des Ergebnisses der Verknüpfung zweier Funktionen stark von der gewählten Darstellungsform abhängt – in der Tat, bei den bisher angesprochenen Darstellungen Boolescher Funktionen müssen jeweils alle Elemente der Erfüllbarkeitsmenge, der Nichterfüllbarkeitsmenge oder der Tabellenform „angefasst“ werden. Es gibt aber auch noch einige weitere Möglichkeiten der Beschreibung Boolescher Funktionen. Eine sehr gebräuchliche Darstellung, die es erlaubt, Funktionen in einfacher Weise durch die gegebenen Operatoren zu verknüpfen und die sich zudem hervorragend dazu eignet, die Gesetze der Booleschen Algebra auszunutzen, sind die Booleschen Ausdrücke. Mit diesen befasst sich der nun folgende Abschnitt 2.3. Weitere Darstellungsformen Boolescher Funktionen werden wir in Kapitel 6 kennen lernen.

2.3 Boolesche Ausdrücke

Boolesche Ausdrücke sind eine weitere Darstellungsform Boolescher Funktionen, die an die „algebraische Struktur“ der Funktionen angelehnt ist. Die Konstruktion Boolescher Ausdrücke sorgt dafür, dass es trivial ist, Boolesche Ausdrücke durch die Operatoren der Booleschen Algebra miteinander zu verknüpfen. Darüber hinaus können Funktionen durch Boolesche Ausdrücke meist sehr viel effizienter dargestellt werden, als dies in Form einer Funktionstabelle (siehe Beispiel 2.3) möglich ist.

Doch bevor wir uns von den Eigenschaften und Vorzügen, aber auch von den Nachteilen Boolescher Ausdrücke ein genaueres Bild verschaffen können, müssen zunächst ihre Syntax und ihre Semantik definiert werden. Hierzu sei im Folgenden $\mathbb{X}_n := \{x_1, \ldots, x_n\}$ eine Menge von n Booleschen Variablen. Ferner sei vorbereitend das Alphabet A durch $A := \mathbb{X}_n \cup \{0, 1, +, \cdot, \sim, (,)\}$ gegeben. Unter A^* verstehen wir die Menge aller beliebig langen endlichen Zeichenketten über A. Die Zeichenkette mit der Länge null ist ebenfalls enthalten und wird mit ϵ bezeichnet.