Inhaltsverzeichnis

Kapitel 4

Rechnerarchitektur

Im Prinzip ist auch ein heutiger Rechner (*Computer*) ein einziges grosses Schaltwerk, das von einer zentralen Taktfrequenz gesteuert wird. Die Komplexität eines derartigen Schaltwerks ist allerdings so gross, dass man schon zu Beginn der Konstruktion von Computern einige wesentliche Teilschaltwerke identifizierte, und diese als Teile des Aufbaus von Rechnern, i. e. der *Rechnerarchitektur* sah. Die Grundstruktur fast aller heutigen Computer basiert auf der *von Neumann–Architektur*, die der ungarische Mathematiker mit US–amerikanischem Pass *John von Neumann* erstmals 1945 formulierte.

Der offensichtliche, enorme Fortschritt der Rechnerleistung, der seither erzielt wurde, beruht zum grössten Teil auf rein technologischen Verbesserungen. Die prinzipielle Architektur blieb jedoch unverändert. So konnte mit Einführung der Halbleitertechnologie der Platzbedarf, die Geschwindigkeit und die Speicherkapazität von Computern fast *exponentiell* gesteigert werden, was auch in *Moore's Law* zum Ausdruck kommt:

> The density of transistors on a chip will double every 18 months, thus increasing the price performance of computing power by a factor of two every 1 1/2 years. – Gordon Moore (Co–Founder, INTEL Corp.), 1978

Hier tritt auch schon das Problem der Quantifizierung der *Computing Power* auf, das wir später noch näher behandeln werden.

4.1 Eine kleine Computergeschichte

Bevor wir uns der Frage zuwenden, was ein (komplexes) Schaltwerk zu einem Computer macht, noch kurz einige Meilensteine der Geschichte der Computer (INFOTEC, 1998).

3000 v. Ch.: Sandabakus (wahrscheinlich in Babylonien)

700–800 n. Ch.: Arabische Zahlen verbreiten sich in Europa (Konzept der Null und der stellengewichteten Ziffer)

1457: Gutenberg erfindet die Druckerpresse

1642: Blaise Pascal baut erste numerische Rechenmaschine (Paris)

1833: Charles Babbage plant die Analytical Machine mit Programmen auf Lochkarten

1844: Samuel Morse sendet Nachricht von Washington nach Baltimore

1936: Konrad Zuse baut ersten Relais–Rechner *Z–1* (Berlin)

1943: John William Mauchly und John Presper Eckert beginnen Entwicklung von ENIAC (Electronic Numerical Integrator And Calculator), dem ersten elektronischen Allzweck–Rechner (Masse: 30m lang, 3m hoch, 2m breit, 18,000 Röhren, 1 Addition 200μs, University of Pennsylvania)

1945: John von Neumann beschreibt die *stored–program*–Architektur

1948: William Bradford Shockley, John Bardeen und Walter H. Brattain erfinden den Transistor

1953: Erstes Magnetbandgerät *IBM 726* (40 Zeichen/cm, 7500 Zeichen/sec)

1954: John Backus entwickelt FORTRAN (IBM)

1958: Seymour Cray baut ersten volltransistorisierten Computer *CDC 1604* (Control Data Corp.)

1970: Gilbert Hyatt reicht Patent für "Single Chip Integrated Circuit Computer Architectur" ein (Mikroprozessor)

1970: ARPANET mit 15 Knotenrechnern (Start des Internet)

1971: Intel Corporation präsentiert ersten Mikroprozessor *Intel 4004*

1977: *Apple*, *Commodore* und *Tally* verkaufen erste *Personal Computers* (PCs)

1981: *IBM* bringt den *IBM 5150 PC* mit *DOS*–Betriebssystem von *Microsoft* (4.77 MHz *Intel 8088* CPU, 4KB RAM, 40KB ROM, 5.25 floppy drive, Grundpreis 3000 US$)

1988: DEC beginnt Entwicklung von 64 bit, 150 MHz Alpha Prozessor

1989: 100,000 Rechner am Internet

1995: *Intel* präsentiert *Pentium Pro* mit 150 MHz

1996: 13,000,000 Rechner am Internet

2000: Erste 1 GHz–Prozessoren (*AMD*, *Intel*) in PCs

2000: 93,000,000 Rechner am Internet [1]

2001: 126,000,000 Rechner am Internet [1]

4.2 Die von Neumann–Architektur

Was macht nun ein Schaltwerk zu einem Computer? Nehmen wir als Beispiel einen einfachen Taschenrechner als (auch historische) Vorstufe zu einem Computer. Jener besitzt ein *Rechenwerk*, das verschiedene einfache arithmetische Operationen ermöglicht. Über eine Eingabe (*Input*) werden Zahlen und Operationen eingegeben, deren Ergebnis (*Output*) am *Display* angezeigt wird. Addieren wir zwei Zahlen $A + B$ und danach $A + C$, so ist das Ergebnis der ersten Addition vergessen, wenn der Taschenrechner keinen *Speicher* aufweist. Wir müssen also wiederum die beiden Zahlen und die Operation "Addition" eingeben, um das erste Ergebnis neuerlich zu erhalten.

Hätte unser Rechner einen Speicher, so könnten wir nicht nur die Ergebnisse speichern, sondern auch die ursprünglichen Zahlen A, B und C. Um diese Zahlen allerdings aus dem Speicher zu holen, müssen wir wissen, wo welche Zahl gespeichert ist, um die richtige Zahl anzusprechen. Wir benötigen also eine Kennzeichnung des Speichers, die *Adressierung* genannt wird. Jede Zahl ist damit an einer definierten Addresse "beheimatet".

Wenn wir jetzt mehrere verschiedene Operationen mit unseren drei Zahlen ausführen möchten, dann müssten wir bei Veränderung von A, B und C die Abfolge von Operationen wiederum eingeben und die jeweiligen Zahlen aus dem Speicher mit der richtigen Adresse holen. Alternativ könnten wir aber auch die Abfolge von Operationen – ein *Programm* P – im Speicher ablegen. Wenn sich unsere Zahlen – die *Daten* D – ändern, so bräuchten wir dem Rechner nur mitteilen "Führe das Programm P aus". Das Programm würde dann die Daten D von den entsprechenden Speicherstellen holen, sie verarbeiten, und die neuen Ergebnisse wiederum abspeichern. Auch das Programm selbst muss aus dem Speicher geholt werden. Das Prinzip der Speicherung

[1] Internet Software Consortium (http://www.isc.org/)

von Daten und Programm in einem gemeinsamen Speicher ist die Grundidee der *von Neumann–Architektur*, die auch als *stored–program*–Architektur bezeichnet wird (Abb. 4.1).

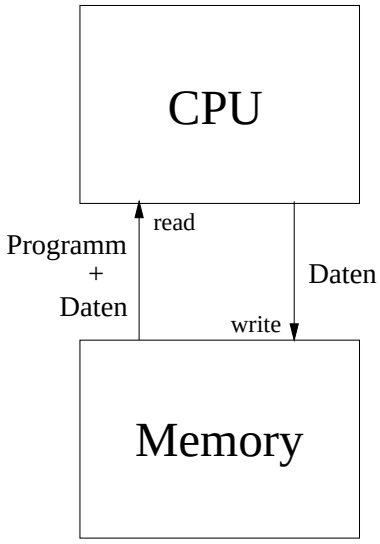

Abbildung 4.1: Prinzip der von Neumann–Architektur.

Hierbei holt eine *Central Processing Unit* (CPU) Programm und Daten aus dem gemeinsamen Speicher (*Memory*), führt die einzelnen Anweisungen (Operationen, *Instructions*) aus und schreibt Ergebnisdaten wieder in den Speicher zurück. Etwas genauer formuliert führt die CPU folgende Schritte aus:

1) Hole Anweisung aus dem Speicher

2) Hole die Daten für diese Anweisung

3) Führe die Anweisung aus

4) Speichere das Ergebnis der Anweisung

5) Gehe zu 1) oder Programmende

Die Leitungen, die die Verbindung zwischen CPU und Speicher herstellen, werden als *Bus* bezeichnet. Dabei unterscheidet man prinzipiell zwischen *Adress–* und *Datenbus*. Um ein Datum oder eine Anweisung aus dem Speicher zu holen, muss die CPU zuerst die Adresse des jeweiligen Speicherplatzes angeben. Die Binärdarstellung der Adresse wird an den Adressbus *angelegt*, was voraussetzt, dass die *Busbreite* (Anzahl der parallelen Leitungen)

diese Adresse aufnehmen kann. Die Busbreite bestimmt also den adressierbaren Speicherraum. Hat die CPU eine Adresse ausgewählt, so werden die Daten/Instruktionen über den Datenbus zurück an die CPU gesendet. Will man also z. B. ein Byte übertragen, so sollte der Datenbus aus 8 parallelen Leitungen bestehen. Ein 16–Bit–Wort müsste dann in zwei Schritten (von zwei verschiedenen Adressen) geholt werden, was doppelt so lange dauert. Ein (aufwendigerer) Datenbus aus 16 Leitungen würde für diesen Fall eine Verdopplung der Geschwindigkeit bewirken. Wir können hier schon die ganz allgemeine Erkenntnis ableiten, dass alle Hardwareverbesserungen die Geschwindigkeit eines Computersystems erhöhen.

4.3 Der Universalrechenautomat

Im folgenden werden wir uns immer weiter in das Innere der CPU, des Speichers, und des Bussystems vorarbeiten. Eine erste Verfeinerung der prinzipiellen von Neumann–Architektur ist der *Universalrechenautomat* nach Händler (Volkert, 1999), dessen Struktur in Abb. 4.2 gezeigt ist.

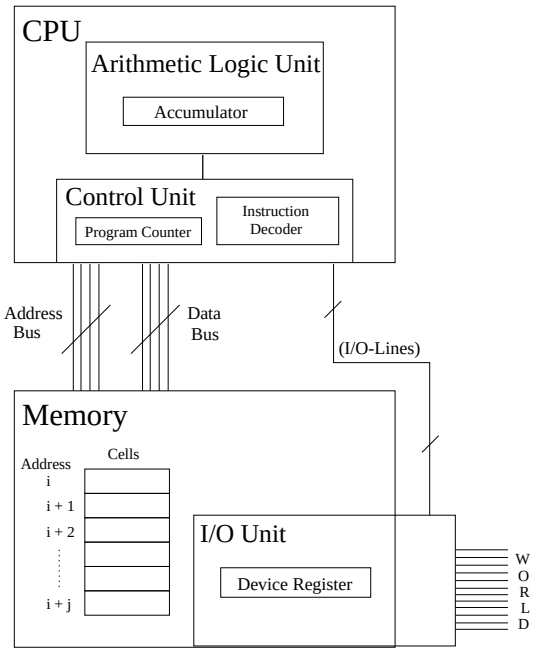

Abbildung 4.2: Ein Universalrechenautomat (URA).

Die Hauptbestandteile eines Computers – CPU und Speicher – werden hier folgendermassen unterteilt:

- CPU
  - Control Unit: Steuerwerk für Programmsteuerung und Befehlsdecodierung
  - Arithmetic Logical Unit: Rechenwerk für arithmetische und logische Operationen
- Speicher
  - Adressierbare Speicherzellen
  - I/O Unit: Ein/Ausgabe von Daten und Programmen (Tastatur, Monitor, Maus, Drucker, Scanner etc.)

Bei der Ansteuerung von I/O–Geräten (*Devices*) unterscheidet man zwischen *memory mapped I/O* und I/O Units, die über spezielle Leitungen mit der CPU verbunden sind. Heute wird fast nur mehr memory mapped I/O verwendet, bei dem die einzelnen Ein/Ausgabe–Geräte Teil des Adressraums des Speichers sind. Ein Gerät wird dann über eine Speicheradresse "beschrieben" und "gelesen", d.h. man schreibt entsprechende Befehlssequenzen in die Register, die an dieser Speicheradresse liegen, um das Gerät zu steuern. Vielfach übernehmen dann eigene Prozessoren in den Geräten die weitere Verarbeitung. Das Schreiben und Lesen einzelner *Device Register* unterscheidet sich aus Sicht der CPU nicht von dem Umgang mit einfachen Speicherzellen.

Die Funktionsweise des URA kann zusammenfassend beschrieben werden:

1) Der URA wird logisch und räumlich in die Teile CPU (Control Unit, ALU), Memory und I/O Unit zerlegt.

2) Das Programm des URA wird über die I/O Unit eingegeben und im Speicher abgelegt. Das Programm ermöglicht die Bearbeitung eines Problems, das von der Struktur des Rechners *unabhängig* ist.

3) Programm und Daten sind im selben Speicher.

4) Der Speicher ist eine numerierte Folge von Speicherzellen. Die Nummer einer Speicherzelle ist deren Adresse, über die der Inhalt der Speicherzelle gelesen oder geschrieben werden kann.

5) Ein Programm ist eine Folge von Befehlen, die in aufeinanderfolgenden Speicherzellen abgelegt sind. Wird ein Befehl von der Adresse a geholt und abgearbeitet, dann wird der nächste Befehl (im Normalfall) von der Adresse $a + 1$ geholt.

6) *Sprungbefehle* bewirken eine Änderung der Reihenfolge der Befehlsarbeitung, die bewirkt, dass der nächste Befehl nicht von der Adresse $a + 1$, sondern von $a \pm n$ geholt wird.

7) *Bedingte Sprünge* sind Sprungbefehle, die nur ausgeführt werden, wenn eine (programmierte) Bedingung erfüllt wird. Ist die Bedingung nicht erfüllt, wird der nächste Befehl (wie üblich) von der Adresse $a + 1$ geholt.

8) Programm und Daten sind binär codiert.

Von diesen acht wesentlichen Punkten ist vor allem Punkt 7) für die Flexibilität des URA verantwortlich. Diese Eigenschaft ermöglicht nämlich, dass sich das Programm in Abhängigkeit von den (zunächst unbekannten) Eingabedaten verhält. Ohne diese Eigenschaft wären z. B. Benutzeroberflächen (Interaktion mit dem Anwender) nicht möglich, und der Rechner damit nicht universell verwendbar.

4.4 Die Architektur des Befehlssatzes

Die primäre Schnittstelle zwischen einem Programmierer und einem Computer ist der Befehlssatz der verwendeten CPU. Verwendet die Programmiererin eine *Hochsprache*, so übernimmt ein *Compiler* die Übersetzung der Anweisungen in Maschinenbefehle. Ein Compiler (*Übersetzer*) ist also ein Hilfsmittel, das dem Programmierer erlaubt, eine CPU (*Maschine*) zu programmieren, ohne deren intrinsischen Befehlssatz zu kennen. Trotz der fortschreitenden Compilertechnologie gibt es auch heute noch Einsatzgebiete, die eine direkte Programmierung in *Maschinensprache* nötig machen (zeit–, und speicherkritische Anwendungen).

Die wichtigste Frage für den Entwickler einer CPU ist daher die Entscheidung über Art und Anzahl der verschiedenen Befehle einer Maschinensprache. Einerseits sollten die Befehle Programme aus allen möglichen Bereichen unterstützen, andererseits würden zuviele verschiedene Befehle die Komplexität einer CPU, und damit die Geschwindigkeit und Programmierbarkeit, negativ beeinflussen. Ganz wesentlich für das Aussehen des Befehlssatzes ist auch das Zusammenspiel zwischen CPU und Speicher.

4.4.1 Maschinenarchitektur

Die wesentlichste Unterteilung des Aufbaus einer CPU in *Maschinenarchitekturen* rührt vom Aufbau des *internen* Speichers einer CPU (Hennessy and

Patterson, 1996). Daten und Befehle, die vom Speicher geholt werden, müssen zwischengespeichert werden, um die Befehle abarbeiten zu können. Aufbau, Manipulierbarkeit und Kapazität des internen Speichers bestimmen somit wesentlich die Architektur des Befehlssatzes.

Zur Illustration der folgenden Maschinenarchitekturen werden wir unser erstes (sehr kleines) Maschinenprogramm schreiben. Unsere Aufgabe ist es zwei Zahlen A, B aus dem Speicher zu addieren und das Ergebnis in die Speicherstelle C abzulegen. Wir verwenden also symbolische Namen und keine Zahlen für die Speicherstellen, was eigentlich schon den Abstraktionsschritt von der Maschinenprogrammierung zur *Assemblerprogrammierung* darstellt. Ein *Assembler* übersetzt die *Mnemonics* der einzelnen Befehle und die symbolischen Adressen in Maschinensprache, was für den menschlichen Programmierer eine wesentliche Erleichterung darstellt, die mit Namen mehr anfangen kann als mit binären Zahlen.

Die Stackmaschine

Der interne Speicher einer *Stack–Maschine* wird durch einen *Stapelspeicher* (Stack) gebildet, dessen prinzipieller Aufbau in Abb. 4.3 dargestellt ist.

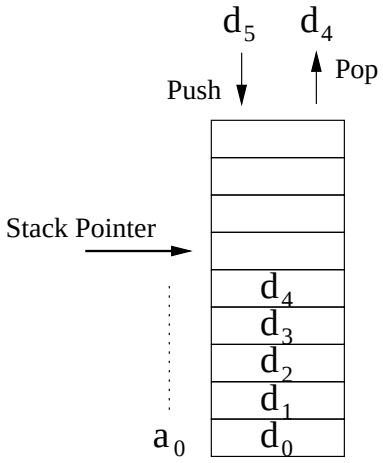

Abbildung 4.3: Ein Stack (Stapelspeicher).

Der *Stack Pointer* (ein internes Register) zeigt dabei immer auf die nächste freie Speicherstelle, in die ein neues Datum aus dem Memory mit dem Befehl `PUSH` geholt wird. Der Befehl `POP` holt das "oberste" Datum vom Speicher, was dieser Datenstruktur die Bezeichnung *LIFO*–Speicher (Last In First Out) gibt, der auch in der Softwaretechnik eine wichtige Rolle spielt. Diese Speicherorganisation unterstützt ganz elementar die *Umgekehrte Polnische Notation* (UPN), bei der arithmetische Operationen in einer speziellen Reihenfolge

ausgeführt werden. So wird die Addition $A + B$ als $AB+$ geschrieben, was auch zum Wegfall von eventuellen Klammern führt ($(A + B) * (C + D)$ wird in UPN zu $AB + CD + *$).

Unser Additionsprogramm für die Stackmaschine würde dann folgendermassen aussehen (mit erfundenen aber durchaus üblichen Assemblerbefehlen):

```
PUSH A     ; hole Datum von Speicherstelle A und lege es auf den Stack
PUSH B     ; hole B auf den Stack
ADD        ; addiere und speichere das Ergebnis am Stack
POP C      ; hole Datum vom Stack und lege es in Speicherstelle C
```

Während die ersten CPUs vielfach Stackmaschinen waren, ist diese Architektur bei den meisten heutigen CPUs nicht mehr anzutreffen. Der Hauptgrund dafür ist, dass immer nur das oberste Datum am Stack explizit angesprochen werden kann, und Daten oft mehrmals aus dem Speicher geholt werden müssen (in unserem Beispiel wäre A und B nach Addition nicht mehr in der CPU). Andererseits ist das Ausführen von Operationen sehr kompakt, da man keinerlei Adressen von Operanden mehr angeben muss, sobald diese auf dem Stack sind (siehe `ADD`). Die einzelnen Befehle produzieren daher einen *kompakten* Code, da nur die Operation, nicht aber die Adresse der Operanden codiert werden muss. Letzteres ist auch der Grund warum man die Stackmaschine auch als *Null–Adressmaschine* bezeichnet.

Eine *Virtual Machine* (VM) ist eine CPU, die nur aus Software besteht, d. h. es ist die Nachbildung einer physikalischen CPU. Die VM "lebt" daher im Speicher, wodurch der Nachteil des wiederholten Ladens von Daten in die CPU wegfällt, die effiziente Befehlsabarbeitung jedoch erhalten bleibt. Dies ist ein wesentlicher Grund dafür, dass die *Java Virtual Machine*, auf der Java–Programme exekutiert werden, als Stackmaschine implementiert ist.

Die Akkumulatormaschine

Viele frühe Maschinen hatten ein einziges internes Register, das vom Programmierer angesprochen werden konnte. Ein solches Register bezeichnet man als *Akkumulator*. Unser Beispielprogramm würde hier folgende Gestalt annehmen:

```
LOAD A     ; hole Datum von Speicherstelle A in den Akkumulator
ADD B      ; addiere B zum Akkumulatorinhalt
STORE C    ; schreibe Akkumulatorinhalt nach C
```

Der Code (Maschinenprogramm) für die Akkumulatormaschine ist zwar auch recht kompakt, da implizit jeder Datentransfer und jede Operation über den Akkumulator läuft, der nicht adressiert werden muss. Wie bei der Stackmaschine bleibt aber das Problem, dass Daten immer wieder geladen werden müssen, wenn sie mehrmals verwendet werden. Die Befehle für die Akkumulatormaschine weisen immer nur eine Adresse auf, daher wird sie auch als *Ein–Address–Maschine* bezeichnet.

Die Registermaschinen

Mit fortschreitender Technologie wurde es immer einfacher und billiger dem Programmierer einen Satz von allgemein verwendbaren Registern zur Verfügung zu stellen. Dies hat den wesentlichen Vorteil, dass oft verwendete Daten in der CPU zwischengespeichert werden können, was gleich mehrere Vorteile bringt: die Befehle für das Nachladen von Daten entfallen (damit das Holen und Ausführen dieser Befehle *und* das Nachladen). Die Bearbeitung dieser Daten in der CPU erfolgt schneller, da die *General Purpose Registers* in der CPU auf Geschwindigkeit optimiert werden können.

Die Registermaschinen können noch in *Register–Memory–*, und *Register–Register*–Maschinen unterteilt werden. Das Beispielprogramm für eine Register–Memory–Maschine lautet:

```
LOAD R1, A     ; hole Datum von Speicherstelle A nach Register R1
ADD R1, B      ; addiere B zu R1
STORE C, R1    ; speichere R1 nach C
```

Auch hier müsste man die Variablen A und B bei mehrmaliger Verwendung nachladen, doch durch einen zusätzlichen Befehl, der die Addition von Registern erlaubt (`ADD R1, R2`) könnte man Variablen und Teilergebnisse in der CPU zwischenspeichern. Allerdings muss der Programmierer explizit Befehle einfügen, damit dies auch wirklich geschieht. Da bei all diesen Operationen immer zwei Operanden addressiert werden müssen, heisst diese Maschine auch *Zwei–Adress–Maschine.*

Unsere kleine Addition auf einer Register–Register–Maschine:

```
LOAD R1, A      ; hole Datum von Speicherstelle A nach Register R1
LOAD R2, B      ; hole Datum von B nach R2
ADD R3, R2, R1  ; addiere Inhalt von R1 und R2 nach R3
STORE C, R3     ; speichere R3 nach C
```

Trotz der etwas aufwendigeren Befehle und Adressierungen stehen nach diesen Anweisungen alle Variablen und das Ergebnis auch nach dem Abspeichern in der CPU zur Verfügung. Da bei dieser Maschinenarchitektur

alle Daten in Register geladen werden, dort dann verarbeitet, und von den Registern wieder gespeichert werden, nennt man diesen Maschinentyp auch *Load–Store–Maschine*. Bei Registeroperationen müssen drei Register adressiert werden, daher heisst sie auch *Drei–Adress–Maschine*.

Theoretisch könnte man natürlich auch eine *Memory–Memory*–Maschine bauen, bei der man zwei Operanden "direkt" im Speicher addieren könnte. Trotz kompakten Codes wäre diese Maschine aber langsam, da bei *jeder* Addition zwei Operanden in den Speicher geholt, addiert, und das Ergebnis rückgespeichert werden müsste. Die meisten heutigen Maschinen sind daher vom Typ Register–Register.

Ein Vorteil der *Load–Store*–Architektur ist auch, dass alle Befehle ähnliche Ausführungszeiten haben. Die Ausführungszeit wird meist in *Clock Cycles* (Taktzyklen) angegeben. Geringe Unterschiede in den Ausführungszeiten ermöglichen ein effizientes *Pipelining* von Befehlen, das wir später noch ausführlich behandeln werden. Daher wählen wir für die hypothetische Maschine *DLX* (Kapitel 5) die *Load–Store*–Architektur.

4.4.2 Speicheradressierung

Wir wissen bereits, dass Speicherzellen über Nummern (Adressen) angesprochen werden, doch verbergen sich hinter dieser einfachen Feststellung viele wichtige Details. Zuerst stellt sich einmal die Frage wieviele Bits an einer Speicheradresse stehen. Aus historischen Gründen sind die meisten heutigen Prozessoren immer noch *byte–adressiert*, d. h. an jeder Adresse wird ein Byte gespeichert. Zusätzlich können heute auch Halbwörter (16 Bits), Wörter (32 Bits, *word*) und Doppelwörter (64 Bits) im Speicher angesprochen werden, die sich dann allerdings über entsprechend viele Adressen erstrecken.

Beispiel: In einem Speicher stehen ab Speicheradresse 1000 16 Bytes in hintereinanderliegenden Speicherzellen. Wir könnten dann von Adresse 1007 ein Byte lesen. Von Adresse 1004 könnten wir ein Wort lesen, wobei das nächste Wort an Adresse 1008 stehen würde. Von 1008 können wir aber auch ein Doppelwort (8 Bytes) lesen. (Was würde passieren, wenn wir das nächste Doppelwort lesen würden?) ◊

In obigem Beispiel stehen die einzelnen Daten an speziellen Speicherstellen. Man sagt, die Daten sind *ausgerichtet* (*aligned*). Für ausgerichtete Daten muss gelten $A \ mod \ s = 0$, wobei A die Adresse eines Datums und s die Grösse des Datums in Bytes ist. Viele Prozessoren sehen aus Hardwaregründen vor, dass die Daten ausgerichtet sein müssen. Ein Lesen/Schreiben eines Halbworts an einer ungeraden Adresse würde dann zu einem *Address Error* führen. Der Programmierer muss also dafür Sorge tragen, dass solche inkorrekten Zugriffe nicht auftreten.

Ein weiteres einfaches Problem, über das aber schon unzählige Programmierer gestolpert sind, ist die Ordnung der Bytes (*Byte Order*). Ist ein Halbwort an einer Adresse gespeichert, so gibt es genau zwei Möglichkeiten die beiden Bytes abzuspeichern. Entweder steht das MSByte an der Adresse, oder das LSByte. Wenn das höherwertige Byte (Bits 15–8) an der Adresse steht, so nennt man dies *Big Endian* (grosses Ende). Im anderen Fall sind die Bytes im *Little Endian*–Format (kleines Ende) gespeichert. Innerhalb einer Maschine bereiten diese Unterschiede meist kein Problem, da die Hardware auf das entsprechende Format vorbereitet ist. Tauschen zwei verschiedene Maschinen jedoch Daten aus (Netzwerk), dann muss die Byteordnung *unbedingt* berücksichtigt werden.

Wenn jede Adresse über eine eigene Nummer angesprochen werden kann, ist der gesamte Speicher für ein Programm zugänglich. Allerdings wird die Nummer grösser, je grösser der Speicher wird. Damit wird aber auch ein Befehl, der diese Speicherstelle anspricht grösser, da das *Adressfeld* mehr Bits bereitstellen muss (überlegen Sie wieviele Bits zur Adressierung eines 64 *MByte*–Speichers reserviert sein müssen). Dies würde aber wieder den Speicherbedarf von Programmen unnötig erhöhen. Ausserdem stehen Daten oft miteinander in Beziehung (Arrays, Matrizen), was man durch effiziente *Adressierungsmodi* unterstützen kann. Die gebräuchlichsten Möglichkeiten der Adressierung wollen wir nun näher betrachten.

Direkte Adressierung

Dieser Adressierungsmodus wird auch als *Absolute Adressierung* bezeichnet, da die absolute Speicheradresse (Nummer) angegeben wird. Als Beispiel für die verschiedenen Modi werden wir einen `MOVE`–Befehl betrachten, der ein Datum in das Register $R1$ transferiert.

```
MOVE R1, 1000     ; hole Inhalt der absoluten Speicheradresse 1000
```

Die Grösse des Datums hängt von der Spezifikation des `MOVE`–Befehls ab. Mit diesem Modus werden z. B. I/O–Devices angesprochen, die an einer systemabhängigen Speicherstelle angesiedelt sind. Ein Nachteil dieses Modus ist der Platzbedarf des Befehls (siehe oben). Viel günstiger ist es, wenn das Datum schon in einem Register steht.

```
MOVE R1, R2     ; hole Inhalt des Registers R2
```

Indirekte Adressierung

Dieser Modus entspricht dem *Pointer*–Konzept in Hochsprachen.

```
MOVE R1, (1000)     ; hole Inhalt der Adresse, die in 1000 steht
```

Es wird also das Datum an Adresse 1000 als weitere Adresse (Pointer) interpretiert, von der das Datum für `R1` geholt wird. Damit ist es möglich durch einen einzigen Befehl (Änderung des Pointers in 1000) auf verschiedene Datensätze zuzugreifen. Dieser spezielle Modus wird *Memory Deferred* bezeichnet, da der Pointer im Speicher abgelegt ist. Alternativ kann der Pointer in einem Register abgelegt sein (*Register Deferred*).

```
MOVE R1, (R2)     ; hole Inhalt der Adresse, die in R2 steht
```

Da man Register schneller und einfacher als Speicherstellen verändern kann, wird dieser Modus sehr häufig verwendet. Eine Spezialform dieser indirekten Adressierung sind *Autoincrement* und *Autodecrement*.

```
MOVE R1, (R2)+    ; hole Inhalt der Adresse in R2 und inkrementiere R2
MOVE R1, -(R2)    ; dekrementiere R2 und hole Inhalt der Adresse in R2
```

Mit diesen Anweisungen lassen sich sehr elegant Datenzugriffe in Schleifen programmieren. Eine sehr typische Anwendung dieser Anweisungen wäre auch die Verwendung von `R2` als Stackpointer (warum?). Erweiterungen der indirekten Adressierung über Register sind *Displacement*

```
MOVE R1, 100(R2) ; hole Inhalt der Adresse R2 + 100
```

und *Indexed*.

```
MOVE R1, (R2+R3) ; hole Inhalt der Adresse R2 + R3
```

Beide Varianten erlauben effiziente Programmierung in zweierlei Hinsicht. Die Befehlslänge (Anzahl der Bits eines Befehls) kann klein gehalten werden, da einige wenige Register sehr effizient zu adressieren sind. Ausserdem erlauben diese Modi die Generierung von sehr kompaktem Programmcode (Qualität des Programmierers/Compilers vorausgesetzt).

Immediate Adressierung

Es gibt auch Daten, die weder im Speicher, noch in einem Register stehen müssen, sondern direkt (*immediate*) im Programmcode angegeben sind.

```
MOVE R1, #100     ; lade R1 mit der Zahl 100
```

Diese Zahl muss dann natürlich explizit im Befehl angegeben werden. Der Platzbedarf hängt dann von der maximalen Grösse der Zahl ab, die vom Architekten des Befehlssatzes zugelassen wird. Stehen Wort–Register zur Verfügung (32 Bit), dann würde das Immediate Datum allein 32 Bit beanspruchen. Allerdings stellt sich die Frage, ob man derart grosse Zahlen jemals direkt eingeben wird.

In einem Befehlssatz einer konkreten Maschine sind selten alle möglichen Adressmodi unterstützt. Man muss daher Grundlagen für die Entscheidung über die Wahl einzelner Parameter erarbeiten. Dies geschieht, indem man *Benchmark*–Programme unter genau definierten Bedingungen auf Maschinen laufen lässt, die einen möglichst grossen Befehlssatz haben. Daraus lassen sich dann Statistiken erstellen, die Anhaltspunkte für die Architektur des Befehlssatzes geben.

Benchmarkmessungen

Für die Architektur unserer DLX–Maschine werden wir Displacement und Immediate als Adressmodi vorsehen, da diese $75\% - 99\%$ aller verwendeten Adressierungsmodi ausmachen. Die Grösse des Displacement wird auf 16 Bit gesetzt werden, da dies in 99% der gemessenen Fälle ausreicht. Auch die Grösse der Immediate–Adressierung wird auf 16 Bit gesetzt, was 80% aller Fälle abdeckt (Hennessy and Patterson, 1996).

4.4.3 Operationen des Befehlssatzes

Die Operationen eines Befehlssatzes können in Gruppen eingeteilt werden. Die wichtigsten davon, die auf jedem heutigen Prozessor implementiert sind, werden wir hier kurz betrachten.

Operationen für den *Datentransfer* sind elementare Voraussetzung, um Daten aus dem und in den Speicher zu bewegen. Operationen aus dieser Gruppe haben wir schon in unserem ersten kleinen Assemblerprogramm kennengelernt.

Operationen für *Arithmetik* und *Logik* umfassen die bekannten Integeroperationen wie Addition (siehe `ADD`), Multiplikation, oder logische Verknüpfungen der Operanden.

Floating Point–Operationen werden in einem speziellen Rechenwerk der CPU ausgeführt. Die Operanden für Addition, Subtraktion, Multiplikation, und Division sind auf den meisten heutigen Maschinen im IEEE–Format codiert. Manchmal gibt es Hardwareimplementierungen für das Berechnen von Quadratwurzeln, oder trigonometrischen Funktionen.

Control–Operationen sind jene Befehle, die den Programmablauf steuern. Man unterscheidet dabei zwischen

- Jump (unbedingter Sprung)
- Branch (bedingter Sprung)
- Call (Prozeduraufruf)
- Return (Rücksprung aus Prozedur)

Alle diese Befehle aus der Control–Gruppe veranlassen, dass das Programm *nicht* mit dem nächsten Befehl, sondern an einer anderen Stelle fortgeführt wird. Anders augedrückt verändern diese Befehle den Program Counter (PC), also jenes Register in der Control Unit, das die Adresse des nächsten auszuführenden Befehls beinhaltet.

Speziell bei Branches kann man beobachten, dass diese sehr oft nur einige Instruktionen überspringen (kleine Schleifen). Daher genügt es zur Adressierung der Sprünge meist die Distanz relativ zum PC anzugeben. In vielen Fällen genügen dafür 8 Bits. Mit diesen lassen sich Sprünge von ± 127 relativ zum PC realisieren.

Im Unterschied zu Jumps und Branches, sind die Sprungadressen von Call und Return nicht vor Programmausführung bekannt, d. h. sie müssen *dynamisch* bestimmt werden.

Beispiel: Wir betrachten drei Prozeduren A, B und C eines Programms. Die Prozedur C kann sowohl von A, als auch von B aufgerufen werden. Am Prozedurende von C steht die Anweisung `RETURN`. Ein Compiler (Assembler) kann hier nicht entscheiden, an welche Adresse zu springen ist, da dies davon abhängt, ob A oder B die Prozedur C aufgerufen hat. Die Rücksprungadresse kann daher erst zur Laufzeit des Programms gebildet werden. $\Diamond$

Die Rücksprungadresse wird je nach Maschine meist in einem speziellen Register oder auch auf einem Stack abgelegt, dessen Struktur sich hier als sehr vorteilhaft erweist. Die Rücksprungadresse ist jene Adresse, an der die Befahlsarbeitung nach Prozedurende fortgesetzt wird.

Branches sind Sprünge, die von einer Bedingung abhängig sind. Die Bedingung wird im Programm definiert, und muss dann zur Laufzeit interpretiert werden. Allerdings ist der Sprung trotzdem vor Programmablauf festlegbar, denn es gibt nur zwei Möglichkeiten: die Bedingung ist erfüllt (Sprung), oder sie ist nicht erfüllt (kein Sprung). Es gibt nun verschiedene Möglichkeiten, eine Bedingung zu prüfen, wobei meist nur eine dieser Möglichkeiten in einem Prozessor realisiert ist.

- Condition Code Register (in einem speziellen Register werden Statusflags gesetzt, die nach jedem Befehl aktualisiert werden)
- Register (ein beliebiges Register wird für Vergleiche benutzt, Condition Code muss vom Programm(ierer) erzeugt werden)
- Compare and Branch (ein einziger Befehl prüft die Bedingung *und* verzweigt)

Ein Nachteil des Condition Code Register ist, dass der Condition Code (wie Zero Flag, Overflow, etc.) bei jeder Operation gebildet wird, auch wenn kein Branch Befehl eine Bedingung prüft. Ein Compare and Branch ist relativ zeitaufwendig (im Vergleich zu anderen Befehlen, Probleme beim Pipelining). Die häufigste Bedingung für Branches ist das Prüfen auf Gleichheit/Ungleichheit zweier Zahlen (in Registern oder Speicherstellen).

Programmtechnisch ist bei Prozeduraufrufen zu beachten, dass Register oft als Zwischenspeicher verwendet werden. Wird z. B. `R1` für eine Variable x verwendet, dann darf eine aufgerufene Prozedur dieses Register nicht verändern. Eine Möglichkeit wäre dann, dieses Register gar nicht zu verwenden, was allerdings oft eine erhebliche Einschränkung darstellt. Daher ist entweder die aufrufende Prozedur (*Caller Save*) oder die aufgerufene Prozedur (*Callee Save*) dafür verantwortlich, dass dieses Register temporär gespeichert wird und nach dem Prozeduraufruf restauriert wird.

Wieviele und welche Befehle sollte nun ein Befehlssatz haben? Bechmarkmessungen für einen *Intel 80x86*–Prozessor ergaben Befehlshäufigkeiten, die in Tabelle 4.1 angeführt sind.

Befehl	Häufigkeit
load	0.22
branch	0.20
compare	0.16
store	0.12
add	0.08
and	0.06
sub	0.05
move	0.04
call	0.01
return	0.01
total	0.96

Tabelle 4.1: Befehlshäufigkeiten für Intel 80x86 bei Integer Benchmarks.

Wir sehen, dass 96% aller Instruktionen von 10 Befehlen abgedeckt werden! Daraus ergibt sich auch ein zentraler Leitsatz für die Architektur des Befehlssatzes: *Make the common case fast.* Die Befehle, die oft vorkommen, sollten also schnellstmöglich ausgeführt werden (Hardware). Überdies erkennt man, dass ein erstaunlich kleiner Befehlssatz genügt (überlegen Sie Gründe, warum in obiger Tabelle kein Multiplikationsbefehl vorkommt) .

Für die DLX ziehen wir also den Schluss, dass wir wenige elementare Befehle zur Verfügung stellen werden. Branches sollten zumindest mit 8 Bit relativ zum PC ermöglicht werden. Neben dieser Sprungadressierung muss diese Architektur aber auch indirekte Adressierung über Register für Rücksprungadressen vorsehen.

4.4.4 Operandentypen

Die Grösse der einzelnen Operanden von Instruktionen hängt elementar von der Rechnerarchitektur insbesondere von der Datenbusbreite ab. Operanden, deren Grösse der Breite des Datenbusses entsprechen können am einfachsten transferiert werden (in einem Taktzyklus). Beim Transfer Operanden unterschiedlicher Grösse erhöht sich der Hardwareaufwand. Es ist unmittelbar einsichtig, dass ein 32–Bit Datenbus mindestens zwei Taktzyklen braucht, um ein Doppelwort zu übertragen.

Der Typ des Operanden wird ganz einfach durch die Instruktion festgelegt. So gibt es eigene *Opcodes* (der Teil eines Befehls, der den Befehl selbst codiert) für Ganzzahl–, und Floating Point–Multiplikation. Dadurch wird sofort klar, welchen Typ die jeweiligen Operanden haben (müssen). Integers werden heute meist als Wort und Doppelwort repräsentiert. Für negative Zahlen wird fast ausschliesslich das 2–Komplement verwendet. Ähnliches Konvergenzverhalten kann man bei der Floating Point–Repräsentation beobachten, bei der durchwegs die beiden IEEE–Formate (32 und 64 Bit) verwendet werden. Manche Maschinen haben noch spezielle Befehle für String/Characteroperationen oder BCD–Zahlen.

Daraus folgt für die DLX–Maschine eine klare, einfache 32–Bit Architektur, die beide IEEE–Formate unterstützen wird.

4.4.5 Befehlssatzcodierung

Ein codierter Befehl kann in drei wesentliche Teile (*Felder*) unterschieden werden. Das eigentliche Befehlsfeld (*Opcode*) gibt den speziellen Befehl an (z. B. `ADD`), das Adressfeld (*Adress Field*) enthält eine Adresse, die in Abhängigkeit eines eventuellen *Address Specifier* interpretiert wird. Der Address Specifier kann entfallen, wenn die Spezifizierung der Adresse im Opcode erfolgt (z.

B. `LW` für "load word" enthält schon die Information darüber, dass ein Wort adressiert wird).

Die wesentliche Frage bei der Codierung des Befehlssatzes ist die nach der Länge der einzelnen Codewörter, die uns schon einmal in Kapitel 1.2 beschäftigt hat. Dort haben wir Codes fester und variabler Länge unterschieden, dessen Bedeutung für einen codierten Befehl in Abb. 4.4 gezeigt ist.

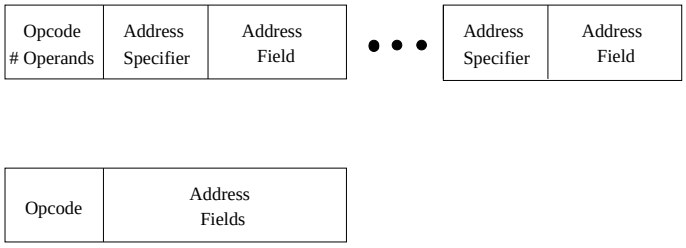

Abbildung 4.4: Befehlscode variabler (oben) und fester Länge (unten).

Wenn wir den Befehlssatz hinsichtlich Programmgrösse optimieren wollen, müssen wir einen Befehlscode variabler Länge vorsehen, da dann für jeden Befehl nur die jeweils nötige Zahl von Bytes gebraucht werden. Diese Variante stellt allerdings höhere Ansprüche an die Control Unit der CPU, da diese bei jedem Befehl aus dem Opcode die Anzahl der Operanden bestimmen muss und die entsprechende Anzahl von Bytes aus dem Speicher holen muss.

Die Programmsteuerung ist bei Befehlscodes fester Länge naturgemäss einfacher, da jedes Befehlswort gleiche Länge aufweist, was wiederum den Programmcode vergrössern wird, da es immer Befehle geben wird, die eigentlich nicht die vorgesehene Länge brauchen würden.

Für die DLX werden wir Befehlscodes fester Länge verwenden und dabei möglichst alle Bits des Codeworts verwenden.

4.4.6 Befehlssatz und Compiler

Da fast alle heutigen Programme in einer Hochsprache geschrieben werden, muss der Befehlssatz einer Maschine nicht nur auf die jeweilige Hardwarearchitektur, sondern auch auf grundlegende Compilertechnologien abgestimmt werden. Heutige Compiler versuchen Programmcode in vielerlei Hinsicht zu optimieren. Eine der wichtigsten Optimierungen betrifft die Verwendung von Registern für Variablen einer Hochsprache. Insbesondere innerhalb von Prozeduren wird versucht, lokale Variable in Registern zu halten, da dies die Ausführungsgeschwindigkeit erhöht. Die Entscheidung welche Variablen in Register kommen, um die Performance zu optimieren ist ein sehr komplexes

Problem. Generell gilt natürlich, dass bei steigender Zahl der Register immer mehr Variablen dort gehalten werden können, allerdings ist auch auf heutigen Maschinen die Anzahl der Register beschränkt. Eine häufig verwendete Methode in Compilern zur Registerallokierung ist das *Graph Coloring*. Diese Methode ist aber nur robust, wenn dem Compiler zumindest 16 Register zur Verfügung stehen.

Auch für den Compilerbau gilt ein ähnliches Prinzip wie in der Maschinenarchitektur: *Make the frequent cases fast and the rare case correct.* Die häufig auftretenden Codesequenzen sollten also auf Geschwindigkeit optimiert werden, während die seltenen "nur" korrekt sein sollten. Daraus ergeben sich einige prinzipielle Anforderungen an die Architektur des Befehlssatzes.

1) Orthogonalität des Befehlscodes (Operationen, Datentypen und Adressierungsmodi sollten beliebig kombinierbar sein)

2) Einfache Grundoperationen (manche Architekturen haben sehr spezielle Befehle, die aber selten von Compilern verwendet werden)

3) Einfache Abhängigkeiten (heute genügt es nicht mehr die Codegrösse zu optimieren, pipelines, branch prediction etc. beeinflussen den Code)

4) Statische Instruktionen (Befehle sollten keine Laufzeitabhängigkeiten haben, um sie zur Compilezeit exakt definieren zu können)

Aus obigen Punkten ergibt sich eindeutig die Forderung nach einem einfachen, klar strukturierten Befehlssatz. Interessanterweise setzte sich diese Erkenntnis aber erst ab 1980 durch. Vorher versuchte man nämlich durch sehr komplexe Befehlssätze (z. B. *VAX*), Maschinensprache an Hochsprache anzunähern. Heute ist aber die Ansicht weit verbreitet, dass eine RISC (*Reduced Instruction Set Computer*)–Architektur beträchtliche Geschwindigkeitsvorteile bringt. Diese gehen zwar auf Kosten der Codegrösse eines Programms, doch zeigt die heutige Entwicklung auf dem Speichersektor, dass dieses Problem an Bedeutung verliert.

Der DLX–Maschine wird daher eine RISC–Architektur zugrundegelegt werden.

4.5 Architekturperformance

Wir haben in diesem Kapitel oft von Verbesserungen von Komponenten der Rechnerarchitektur gesprochen, nun wollen wir versuchen, diese Verbesserungen zu quantifizieren und in Masszahlen zu fassen.

Ganz global betrachtet führt ein Computer(system) ein Programm in einer messbaren Zeit t_P aus. Führt man nun eine oder mehrere Verbesserungen, (z. B. schnellerer Speicher, schnellere Befehle, höhere Taktfrequenz, etc.) aus, so wird t_P (hoffentlich) verringert. Bezeichnen wir die Ausführungszeit auf der ursprünglichen Maschine mit t_P, old und jene auf der verbesserten Maschine mit t_P, new so ergibt sich der *Speedup* s zu

$$s = \frac{t_{P,old}}{t_{P,new}}. \tag{4.1}$$

Kennen wir den Anteil f_e am Gesamtprogramm von den Befehlen, auf die sich die Verbesserung auswirkt, und den Speedup s_e dieser speziellen Verbesserung, dann kommen wir über *Amdahl's Law* zum Speedup

$$s = \frac{1}{(1 - f_e) + \frac{f_e}{s_e}}. \tag{4.2}$$

Beispiel: Wir betrachten eine Verbesserung einer Maschine, die 40% aller Befehle betrifft. Diese Verbesserung erhöht die Geschwindigkeit dieser Befehle um einen Faktor 10. Wie gross ist der Speedup der Maschine? Wir setzen ein in Gleichung 4.2 und erhalten

$$s = \frac{1}{0.6 + \frac{0.4}{10}} \simeq 1.56.$$

$\Diamond$

Eine weitere wichtige Masszahl zur Beurteilung der Performance einer Rechnerarchitektur sind die *Clocks per Instruction*

$$CPI = \frac{n_C}{n_I}, \tag{4.3}$$

wobei n_C die Anzahl der Taktzyklen ist, die ein Programm zur Ausführung braucht, und n_I die Anzahl der Instruktionen des Programms ist. über die CPI-Rate kommt man einfach zur Ausführungszeit

$$t_P = CPI \times n_I \times t_C, \tag{4.4}$$

mit t_C als der Periodendauer eines Taktzyklus. Die CPI–Rate ist unabhängig von der Taktfrequenz und der Anzahl der Instruktionen eines Programms, was wesentlich zu einer Vergleichbarkeit der CPI–Raten beiträgt. Allerdings ist auch diese Masszahl von mehreren Parametern abhängig. Generell kann man folgende Abhängigkeiten identifizieren:

- Taktzykluszeit – Technologie der Hardware

- CPI – Organisation und Befehlssatzarchitektur
- n_C – Befehlssatzarchitektur und Compilertechnologie

Oft ist es auch nützlich, die CPI–Raten einzelner Befehle oder Befehlsgruppen zu kennen. Die gesamte CPI–Rate ergibt sich dann zu

$$CPI = \sum_{i=1}^{n} CPI_i \; f_i, \tag{4.5}$$

mit CPI_i als CPI–Rate einer Teilmenge des Befehlssatzes und f_i der Anteil dieser Teilmenge an der Anzahl der Gesamtinstruktionen. Die Summe ist dann über alle n Teilmengen zu bilden, d. h., es muss gelten $\sum_{i=1}^{n} f_i = 1$.

Eine andere oft gebrauchte Masszahl sind die *Million Instructions Per Second*

$$MIPS = \frac{n_I}{t_P \times 10^6} = \frac{1}{CPI \times t_C \times 10^6}. \tag{4.6}$$

Die MIPS–Rate ist jedoch eine sehr ungenügende Masszahl, da sie sogar bei derselben Rechnerarchitektur und demselben Programm unterschiedliche (compilerabhängige) Ergebnisse bringen kann. Es ist sogar möglich, dass die MIPS–Rate indirekt proportional zur Perfomance eines Systems ist. Ähnlich ist die Situation bei den *Million Floating Point Operations Per Second* (MFLOPS), bei denen nicht die Gesamtzahl der Instruktionen n_I, sondern nur die Anzahl der Floating Point–Operationen betrachtet wird.

Literaturverzeichnis

Ash, R. (1965). *Information Theory.* Wiley, New York, 1st edition.

Broy, M. (1993). *Rechnerstrukturen und maschinennahe Programierung.* Informatik, Eine grundlegende Einführung, Teil II. Springer, Berlin.

Carlson, A. B. (1975). *Communication Systems.* McGraw–Hill, Tokyo, 2nd edition.

Dworatschek, S. (1970). *Schaltalgebra und digitale Grundschaltungen.* de Gruyter, Berlin.

Hennessy, J. L. and Patterson, D. A. (1996). *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, San Francisco, 2nd edition.

Huffman, D. A. (1952). A Method for the Construction of Minimum Redundancy Codes. *Proceedings of the IRE*, 40(10):1098–1101.

INFOTEC (1998). http://www.infotec.org/history.htm. WWW Repository, Association of Information Technology Professionals.

Küpfmüller, K. (1954). Die Entropie der deutschen Sprache. *FTZ*, 7(6):265–272.

Lochmann, D. (1995). *Digitale Nachrichtentechnik.* Verlag Technik, Berlin, 1st edition.

Mendelson, E. (1982). *Boolesche Algebra und Logische Schaltungen.* Schaum's Outline. McGraw–Hill, Hamburg.

Newald and Lindner (1985). Digitale Schaltwerke. Laborunterlagen, Institut für Datenverarbeitung, Technische Universität Wien.

Volkert, J. (1999). Digitale Rechenanlagen. Vorlesungsunterlagen, Institut für Technische Informatik und Telematik, Universität Linz.