Einführung in die Informatik 1

Prof. Dr. Wolf-Dieter Tiedemann
Fakultät Informatik
Technische Hochschule Ingolstadt

19. Januar 2020

Dieses Skriptum ist als Unterstützung für die Hörer meiner Lehrveranstaltung gedacht.

Es wird von mir ausschließlich als `.pdf`-Datei über die eLearning-Plattform moodle der TH Ingolstadt angeboten. Sollte es an anderem Ort erscheinen, so geschieht dies ohne mein Wissen und unter Verletzung der Benutzungsrichtlinien für Informationsverarbeitungssysteme der TH Ingolstadt.

Das Skriptum wird mit jedem Vorlesungszyklus aktualisiert.

Ziel dieser Lehrveranstaltung ist die Entwicklung eines Grundverständnisses davon, wie Algorithmen – Folgen von maschinell ausführbaren Rechenschritten – auf Rechnern – programmgesteuerten Informationsverarbeitungssystemen – ausgeführt werden.

Nach erfolgreicher Teilnahme sind die Studierenden in der Lage

- den Begriff des Algorithmus zu erläutern,
- zu beurteilen, ob ein Problem berechenbar ist, d.h. ein Algorithmus zu seiner Lösung formuliert werden kann,
- die Komplexität eines gegebenen Algorithmus abzuschätzen,
- zu verstehen, wie ein Algorithmus auf einem Rechner bearbeitet wird,
- den Aufbau eines Universalrechners und seine Arbeitsweise zu beschreiben,
- verschiedene fortgeschrittene Konzepte der Rechnerarchitektur einzuordnen.

Der **Inhalt** dieser Lehrveranstaltung umfasst folgende Themenfelder:

- Algorithmen
  - Algorithmenbegriff, Eigenschaften, Darstellungsformen
  - Berechenbarkeit
    - Turing-Berechenbarkeit
    - LOOP-, WHILE-, GOTO-Berechenbarkeit
    - Church-Turing-These
  - Entscheidbarkeit
  - Komplexität
    - $\mathcal{O}$-Notation
    - Komplexitätsklassen P und NP
- Rechnerarchitektur
  - Binäre Informationsdarstellung
    - Natürliche, negative, gebrochene Zahlendarstellungen
    - Maschinenbefehle und -programme
  - Digitale Schaltungen
    - Verknüpfungsglieder, Schaltnetze
    - Speicherglieder, Register, Zähler, Schaltwerke
  - von Neumann-Rechner
  - Fortgeschrittene Konzepte in heutigen Rechnerarchitekturen
    - Caching
    - Mehrkern-Architekturen
    - Befehlspipelining

Einleitung

Am Anfang dieser *Einführung in die Informatik* muss natürlich die Frage stehen: *Was ist Informatik*? Viele Buchautoren, Enzyklopädien, Universitäten und Fachverbände bieten ihre Definitionen für Informatik an. Hier wollen wir die Definition zugrunde legen, die die *Association for Computing Machinery* im Jahre 1989 veröffentlicht hat [1]:

> Computer science is the systematic study of *algorithmic processes* that describe and transform information: their theory, analysis, design, efficiency, implementation, and application.

Die Informatik ist also die Wissenschaft der Informationsverarbeitung. *Informationsverarbeitung* geschieht duch eine schrittweise Umwandlung von Eingabedaten in Ausgabedaten. Die Verarbeitungsvorschrift, die die im Einzelfall durchzuführenden Verarbeitungsschritte vorgibt, heißt **Algorithmus**.

Der Begriff *Algorithmus* leitet sich ab aus dem Namen des persischen Mathematikers und Astronomen Muhammad Ibn Musa Al-Chwarizmi, der um 825 am „Haus der Weisheit" – der Akademie in Bagdad – lehrte und ein Buch verfasste, in dem er Rechenverfahren zur Lösung linearer und quadratischer Gleichungssysteme beschrieb.

Ein Algorithmus löst eine Klasse von Problemen. Unterschiedliche Probleme derselben Klasse sind durch unterschiedliche Eingabedaten gekennzeichnet, die durch den Algorithmus zu korrespondierenden Ausgabedaten umgewandelt werden. Durch einen Algorithmus wird somit eine Relation zwischen Eingabedaten und Ausgabedaten hergestellt. Insofern kann ein Algorithmus mathematisch als Funktion aufgefasst werden, bzw. einer Funktion kann ein Algorithmus zugeordnet werden, der sie berechnet. Intuitiv ist jedoch klar, dass es auch Funktionen gibt, die gar nicht berechenbar sind. Man denke beispielsweise an die Berechnung der Lottozahlen von nächster Woche. Für solche Funktionen existiert kein Algorithmus. Spannend ist also die Frage, ob für eine gegebene Funktion entschieden werden kann, ob sie berechenbar ist oder nicht.

Existiert ein Algorithmus und wird er mit den Sprachmitteln einer konkreten *Programmiersprache* beschrieben, so kann er auf einer *Maschine* ausgeführt werden.

In dieser Aussage liegen die Gegenstände der beiden Kapitel dieser Vorlesung begründet:

1. Welche Probleme können grundsätzlich durch einen Algorithmus gelöst werden und darüber hinaus effizient gelöst werden?
2. Wie muss eine Maschine beschaffen sein, um einen als Programm formulierten Algorithmus auszuführen?

Kapitel 1

Algorithmen

Gemäß obiger Einleitung ist ein Algorithmus eine Handlungsvorschrift, die in einer Folge von Einzelschritten beschreibt, wie aus gegebener Information (Eingabe) gesuchte Information (Ausgabe) ermittelt wird.

Von einem Algorithmus werden bestimmte Eigenschaften erwartet.

1.1 Eigenschaften eines Algorithmus

Algorithmen besitzen die folgenden charakteristischen Eigenschaften:

Allgemeinheit: Ein Algorithmus beschreibt nicht nur die Lösung eines singulären Problems, sondern einer ganzen Klasse gleichartiger Probleme[1]. Die einzelne Problemausprägung ist durch individuelle Eingabedaten gekennzeichnet.

Eindeutigkeit: Ein Algorithmus darf keine widersprüchlichen oder missverständlichen Anweisungen vorgeben.

Finitheit (Endlichkeit): Die Beschreibung eines Algorithmus besitzt eine endliche Länge, d. h. die Anzahl seiner Anweisungen ist endlich (*statische* Finitheit). Zudem darf ein Algorithmus zu jedem Zeitpunkt der Ausführung nur endlich viele Ressourcen belegen (*dynamische* Finitheit).

Ausführbarkeit: Jeder Einzelschritt muss ausführbar sein. Dabei hängt die Ausführbarkeit immer von den Möglichkeiten der ausführenden Maschine ab.

Determinismus: Die Abfolge der einzelnen Schritte eines Algorithmus muss eindeutig festgelegt sein, d. h. zu jedem Zeitpunkt der Ausführung besteht höchstens eine Möglichkeit der Fortsetzung.

Determiniertheit: Wird ein Algorithmus mit den gleichen Eingabewerten und Voraussetzungen wiederholt, so liefert er stets das gleiche Ergebnis.

Determinismus und Determiniertheit sind auseinanderzuhalten: Determinismus kennzeichnet einen Algorithmus, bei dem der gesamte Ablauf eindeutig bestimmt ist. Determiniertheit bezieht sich nur auf die eindeutige Bestimmtheit des Ergebnisses. Deterministische Algorithmen haben durch ihren eindeutigen Ablauf auch ein eindeutiges Ergebnis, sie sind daher stets determiniert. Die Umkehrung gilt jedoch nicht. Es gibt nicht-deterministische Algorithmen, die über verschiedene Wege stets zum gleichen Ziel kommen, also determiniert sind.

[1] Diese Eigenschaft wird gelegentlich auch als *Abstraktion* bezeichnet.

Terminierung: Nach einer endlichen Anzahl von Schritten ist der Algorithmus beendet und liefert ein Ergebnis[2].

Eine in der Praxis besonders wichtige Eigenschaft ist die Korrektheit.

Korrektheit: Der Algorithmus löst die gegebene Aufgabenstellung bzw. „seine" Problemklasse korrekt.

Die Korrektheit eines Algorithmus lässt sich immer nur in Bezug auf eine Spezifikation feststellen. Eine *Spezifikation* ist eine eindeutige Festlegung der berechneten Funktion. Die Festlegung geschieht durch zwei Zusicherungen, die die Wirkung des Algorithmus spezifizieren. Die Zusicherungen werden als Vorbedingung und Nachbedingung bezeichnet. Die *Vorbedingung* legt zulässige Werte für die Eingabedaten des Algorithmus fest. Wenn die Vorbedingung gilt, muss der Algorithmus terminieren. Die *Nachbedingung* formuliert die Eigenschaften, die die Ausgabedaten aufweisen müssen.

Beispiel 1.1 Die Spezifikation einer Funktion $sqrt(x)$, die einen Wert y liefert, dessen Quadrat y^2 dem Eingabewert x entspricht, besteht aus der Vorbedingung

$$x \in \mathbb{R} \wedge x \geq 0$$

und der Nachbedingung

$$y \in \mathbb{R} \wedge y \geq 0 \wedge \ \left|y^2 - x\right| < \epsilon$$

□

Die Überprüfung eines Algorithmus auf Korrektheit kann auf zwei grundsätzlich verschiedene Weisen erfolgen:

1. *Verifikation*: mittels Deduktion wird logisch bewiesen, dass ein Algorithmus seine Spezifikation erfüllt (formaler Beweis der Korrektheit)

   Formale Korrektheitsbeweise sind sehr aufwändig. Eine bekannte Methode, mit der man für Algorithmen, die in einer prozeduralen Programmiersprache beschrieben sind, einen Korrektheitsbeweis durchführen kann, ist der HOARE-Kalkül[3]. Mittels fünf logischer Ableitungsregeln kann die Nachbedingung einer Spezifikation als logische Folgerung aus der Vorbedingung abgeleitet und somit die Korrektheit eines Algorithmus bewiesen werden.

2. *Validation*: der Algorithmus wird durch systematisches Testen mit ausgewählten Eingabedaten erprobt, für die das Ergebnis bekannt ist (empirischer Nachweis[4] der Korrektheit)

Ein Algorithmus, der für jede Eingabe, die die Vorbedingung erfüllt, entweder ein Ergebnis erzeugt, das die Nachbedingung erfüllt, oder nicht terminiert, heißt *partiell korrekt*. Ein Algorithmus heißt *total korrekt*, wenn er partiell korrekt ist und für jede Eingabe, die die Vorbedingung erfüllt, terminiert.

Neben der Korrektheit ist aus praktischer Sicht insbesondere bei komplexen Problemstellungen mit umfangreichen Datenmengen auch die Effizienz wichtig.

[2] Ein Grenzfall sind interaktive bzw. reaktive Algorithmen, die von Eingaben aus ihrer Umgebung abhängen und somit bei Ausbleiben entsprechender Eingaben unendlich lange in einem Wartezustand verweilen.

[3] benannt nach seinem Erfinder C. ANTONY R. HOARE, 1969

[4] Beim Testen kann lediglich das Vorhandensein von Fehlern entdeckt, nicht jedoch die Fehlerfreiheit nachgewiesen werden.
Manche Autoren verstehen unter Validierung zusätzlich auch die Validierung der *Gebrauchstauglichkeit*, d. h. die Prüfung, ob Nutzer im spezifizierten Nutzungskontext auch die Nutzungsziele zufriedenstellend erreichen können.

Effizienz: Ein Algorithmus sollte in so kurzer Zeit und mit so wenig Ressourcen wie möglich zu einem korrekten Ergebnis kommen.

Wichtige Kennzahl ist die *Laufzeitkomplexität* des Algorithmus, d. h. die Wachstumsrate, mit der sich der Bedarf an Speicherplatz und Rechenzeit verändert, wenn die Problemgröße steigt.

Die Effizienz eines Algorithmus ist besonders dann ein wichtiges Merkmal, wenn man für eine gegebene Problemklasse mehrere Algorithmen zur Auswahl hat.

Weitere praxisrelevante Eigenschaften sind *Portabilität* (Algorithmen sollten nicht auf eine bestimmte Maschine zugeschnitten sein, sondern prinzipiell auf beliebigen Maschinen ausführbar sein), *Wiederverwendbarkeit* (Algorithmen sollten so formuliert sein, dass sie auch modular zur Lösung von Teilproblemen in anderem Kontext eingesetzt werden können) und *Erweiterbarkeit* (Algorithmen sollten leicht an geänderte Anforderungen anpassbar sein).

1.2 Darstellung von Algorithmen

Algorithmen können auf verschiedene Weise dargestellt werden.

Naheliegend ist die **natürliche Sprache**. Diese Darstellungsform ist allgemein verständlich, von hoher Mächtigkeit, flexibel und variantenreich, lässt allerdings viel Raum für Mehrdeutigkeiten und Missverständnisse, verhindert keine Widersprüchlichkeiten und erzwingt keine Detaillierung.

Aus diesem Grund wurden bereits in den 1960er Jahren *visuelle* Darstellungsmethoden vorgeschlagen. Zwei bekannte Beispiele sind das *Flussdiagramm* (auch Programmablaufplan oder Programmstrukturplan) und das *Struktogramm* (auch Nassi-Shneiderman-Diagramm).

Das **Flussdiagramm** zur Algorithmendarstellung basiert auf dem von Frank Gilbreth im Jahre 1921 vorgestellten *flow process chart*, einer Methode zur Darstellung und Analyse von Arbeitsabläufen. Die wichtigsten Grafikelemente eines Flussdiagramms sind in Abbildung 1.1 zusammengefasst.

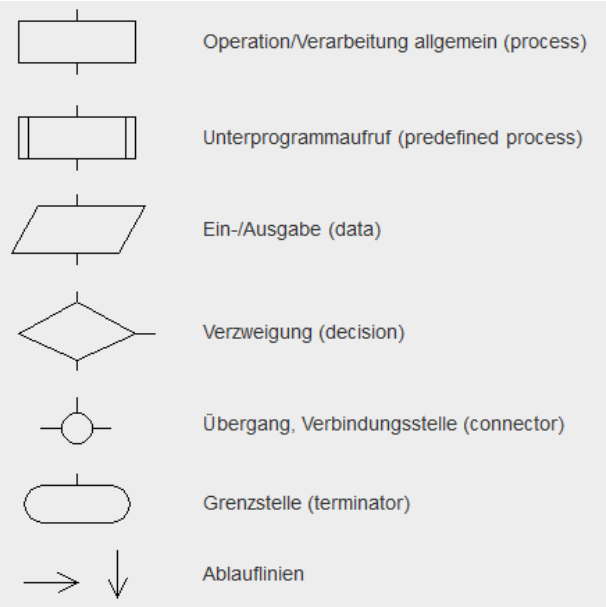


Abbildung 1.1: Wesentliche Grafikelemente eines Flussdiagramms (DIN 66001)

Beispiel 1.2 Ein Flussdiagramm für einen Algorithmus zur Berechnung des arithmetischen Mittels ist in Abbildung 1.2 wiedergegeben. □

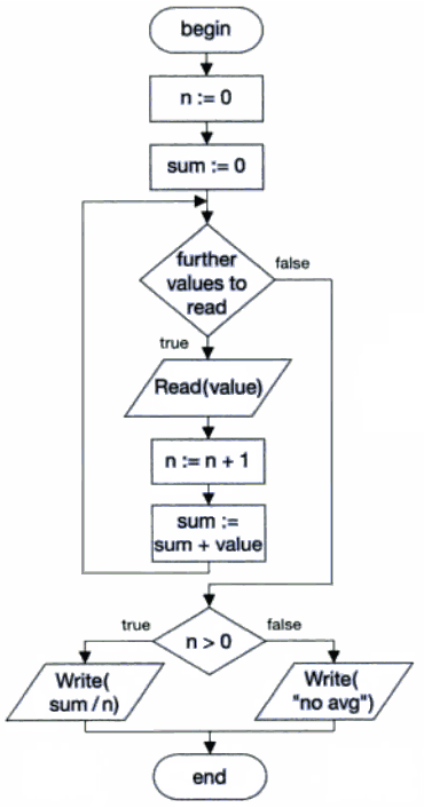


Abbildung 1.2: Flussdiagramm für einen Algorithmus zur Berechnung des arithmetischen Mittels

Die gewichtigste Kritik an Flussdiagrammen entstand in der Softwarekrise Mitte der 1960er Jahre, als man erkannte, dass mit der Leistungssteigerung der Hardware auch die Software immer komplexer wurde und sich kaum noch ausreichend testen (geschweige denn verifizieren) ließ. Durch Softwarefehler entstanden Millionenschäden. Als Konsequenz entstand die Disziplin *Software Engineering*, die sich mit Prinzipien, Methoden und Werkzeugen zur Entwicklung zuverlässiger, umfangreicher Softwaresysteme beschäftigt. Dazu gehört das Prinzip der strukturierten Programmierung, die auf die Verwendung von **goto**-Sprunganweisungen vollständig verzichtet[5]. Leider verleiten Flussdiagramme durch die freie Verkettung von Einzelschritten durch Ablauflinien zu unstrukturiertem, **goto**-durchsetztem „Spaghetti-Code".

Durch eine theoretische Arbeit von Corrado Böhm und Giuseppe Jacopini wurde 1966 gezeigt, dass jeder Algorithmus durch gerade drei Steuerkonstrukte beschrieben werden kann, die rekursiv auf Elementaranweisungen angewendet werden: die *Sequenz*, die *Selektion* und die *Iteration*. Diese Erkenntnis bildet den Hintergrund für die zweite visuelle Darstellungsform, das **Struktogramm**. Abbildung 1.3 fasst die wichtigsten Grafikelemente eines Struktogramms zusammen.

Beispiel 1.3 Ein Struktogramm für einen Algorithmus, der für eine eingegebene Zahl n einen Primzahltest durchführt, ist in Abbildung 1.4 dargestellt. □

[5] Die **goto**-Anweisung wurde in vielen frühen Programmiersprachen aus dem Maschinenbefehlssatz übernommen. Ihre Anwendung brachte jedoch häufig verborgene Seiteneffekte mit sich, die die Software schwer wartbar machten. Edsger W. Dijkstra verfasste dazu 1968 seine berühmte Schrift *"Go To Statement Considered Harmful"*.

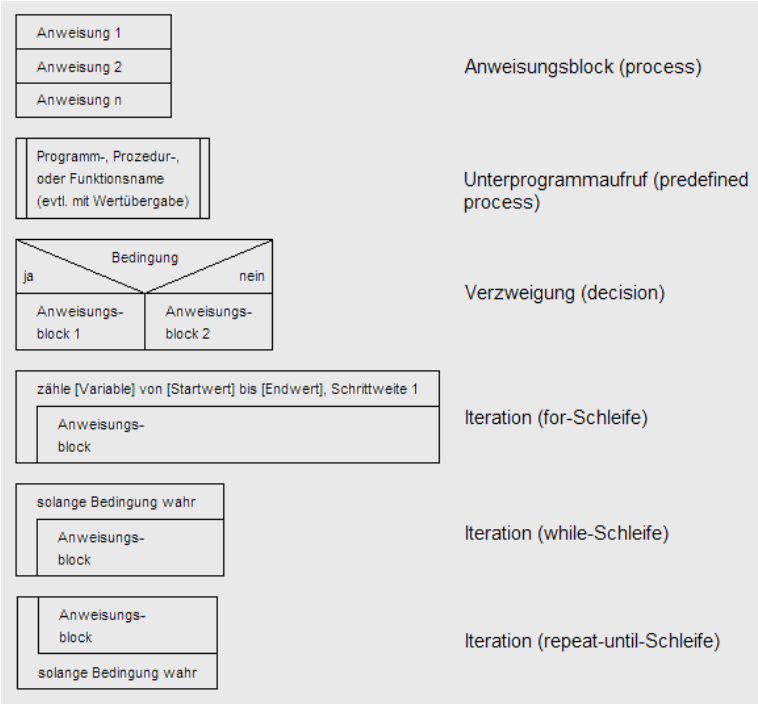


Abbildung 1.3: Wesentliche Grafikelemente eines Struktogramms (DIN 66261)

Die offensichtlichste Darstellungsform für Algorithmen ist natürlich das Formulieren in einer konkreten **Programmiersprache**. Eine Programmiersprache erlaubt es, Algorithmen *präzise* zu beschreiben. Alle elementaren Operationen und alle Steuerkonstrukte sind ausführbar, die zulässigen Datenbereiche sind eindeutig festgelegt. Damit ist das Ziel der Formulierung eines Algorithmus, nämlich die tatsächliche Ausführung auf einer Maschine, unmittelbar erreicht. Allerdings erfordert diese Darstellungsform eine genaue Kenntnis der Syntax und der Semantik der verwendeten Programmiersprache und ist somit nur entsprechenden Fachleuten zugänglich.

Beispiel 1.4 Ein Algorithmus, der das Maximum in einer gegebenen Liste L ganzer Zahlen sucht, ist nachfolgend in der Programmiersprache *Python* sowohl in iterativer Version als auch in rekursiver Version dargestellt.

iterative Version:

```
def find_max(L):
    max = 0
    for i in L:
        if i > max:
            max = i
    return max
```

rekursive Version:

```
def find_max(L):
    if len(L) == 1:
        return L[0]
    i1 = L[0]
    i2 = find_max(L[1:])
    if i1 > i2:
        return i1
    return i2
```

□

Um den Algorithmenentwickler von den syntaktischen Zwängen einer konkreten Programmiersprache zu befreien, ist eine weitere Darstellungsform verbreitet, der **Pseudocode**. Pseudocode ist ein Mittelding zwischen natürlicher Sprache und formaler Programmiersprache. Meist werden

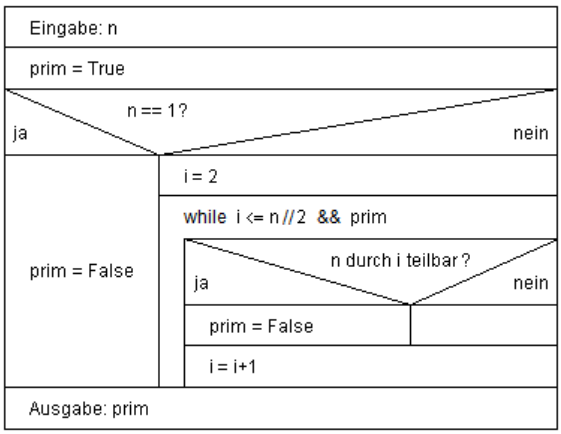


Abbildung 1.4: Struktogramm für einen Primzahltest

Schlüsselwörter verwendet, die an echte Programmiersprachen angelehnt sind, und diese mit natürlichsprachlichen Formulierungen ergänzt. Pseudocode ist nicht standardisiert. Er wird intuitiv verwendet und stellt die Essenz des Algorithmus in den Vordergrund.

Beispiel 1.5 Die Berechnung des größten gemeinsamen Teilers zweier gegebener Zahlen a und b kann durch den *Euklidischen Algorithmus* erfolgen, der durch folgenden Pseudocode beschrieben ist:

IF b größer als a THEN vertausche beide
REPEAT
    teile a ganzzahlig durch b
    IF Rest gleich 0 THEN b ist das Ergebnis und der Algorithmus endet
    ELSE
        verwende b als neuen Dividenden
        verwende den Rest als neuen Divisor

□

Neben natürlich- oder formal-sprachlichen und graphisch unterstützten Algorithmendarstellungen sei als letzte Darstellungsform die Beschreibung als **Hardwareentwurf** genannt. Beschreibungselemente sind hier bspw. Leitungen, logische Schaltglieder, Register, Multiplexer und Codierer. Entsprechende Algorithmen implementieren etwa effiziente Formen der Addition und Multiplikation oder beliebige andere Schalt- und Steuerfunktionen. Wir werden solche Algorithmen in Kapitel 2 genauer kennenlernen. Die Ausführbarkeit ist bei dieser Darstellungsform offensichtlich.

Die genannten Beschreibungselemente sind Elemente realer Digitalrechner. Natürlich kann man sich auch abstrakte Maschinen vorstellen, die eher mathematischen Modellen als realer Hardware ähneln.

Dieser Ansatz wurde (vor der Erfindung des Digitalrechners) in der ersten Hälfte des 20. Jahrhunderts gewählt, um den Begriff des Algorithmus mathematisch präzise zu fassen. Insbesondere konnte durch die mathematische Präzisierung die Frage geklärt werden, ob es für jede denkbare Problemstellung einen Algorithmus zur Lösung gibt oder ob für gewisse Problemstellungen gar keine algorithmische Lösung existiert. Man verwendet in diesem Zusammenhang den Begriff der *Berechenbarkeit*.

1.3 Berechenbarkeit

Ein Problem heißt *berechenbar*, wenn zu seiner Lösung ein Algorithmus formuliert werden kann.

Um den Berechenbarkeitsbegriffs formal zu fassen, wurden u. a. abstrakte Maschinenmodelle entwickelt wie die Registermaschine oder die TURING-Maschine. Letztere bildet das älteste und das bekannteste Modell für die Formalisierung des Berechenbarkeitsbegriffs und soll daher auch hier betrachtet werden.

1.3.1 TURING-Berechenbarkeit

Der englische Mathematiker ALAN M. TURING entwickelte das Modell der TURING-Maschine im Jahre 1936 [2]. Er ging dabei von der Vorstellung aus, dass „Berechnungen normalerweise in der Weise ausgeführt werden, dass bestimmte Symbole auf ein Blatt Papier geschrieben werden. Wir wollen annehmen, dass dieses Papier kariert ist wie das Rechenheft eines Schulkindes“. Ein anschauliches Beispiel ist etwa die Multiplikation nach Schulmethode:

	5	6	7	8	·	4	3	2	1
		2	2	7	1	2			
			1	7	0	3	4		
				1	1	3	5	6	
						5	6	7	8
		2	4	5	3	4	6	3	8

TURING argumentiert weiter, „dass die Zweidimensionalität des Papiers für die Rechnung nicht wesentlich ist“ und geht daher davon aus, „dass die Rechnung auf eindimensionalem Papier ausgeführt wird, d. h. auf einem Band, dass in Zellen unterteilt ist“.

Jede Zelle kann genau ein Symbol aus einer endlichen Menge von Symbolen (dem *Bandalphabet*) aufnehmen oder leer sein. Die Endlichkeit des Alphabets betrachtet TURING nicht als Einschränkung. Er argumentiert mit der Wahrnehmungsfähigkeit des Beobachtenden. Bei einer *sehr* großen Symbolmenge werden sich einzelne Symbole so ähnlich, dass ihr Unterschied nicht mehr wahrgenommen wird. Ist er wichtig, können stattdessen auch zwei gut unterscheidbare Symbole verwendet werden. Damit der Platz auf dem Band nie ausgeht, ist das Band auf beiden Seiten unbegrenzt.

Um nun eine Berechnung auf dem Band auszuführen, müssen Symbole gelesen, gelöscht und geschrieben werden können. Dafür gibt es einen Schreib-/Lesekopf, der genau eine Zelle auf dem Band bearbeiten kann. Zudem kann er sich jeweils eine Zelle nach rechts oder nach links bewegen. Wie welcher Bearbeitungsschritt genau aussieht, hängt von einer *Steuerung* ab. Die Steuerung befindet sich stets in einem bestimmten Zustand aus einer endlichen Menge von Zuständen. Abbildung 1.5 skizziert eine solche TURING-Maschine.

Die Endlichkeit der Zustandsmenge begründet TURING in der gleichen Weise wie die Endlichkeit des Bandalphabets. Bei unendlich vielen Zuständen wären manche Zustände so ähnlich, dass sie kaum mehr unterscheidbar wären und faktisch zusammengeführt werden können. Außerdem kann die Einführung weiterer Zustände, die komplexere Sachverhalte beschreiben sollen, auch dadurch aufgefangen werden, dass zusätzliche Symbole aufs Band geschrieben werden.

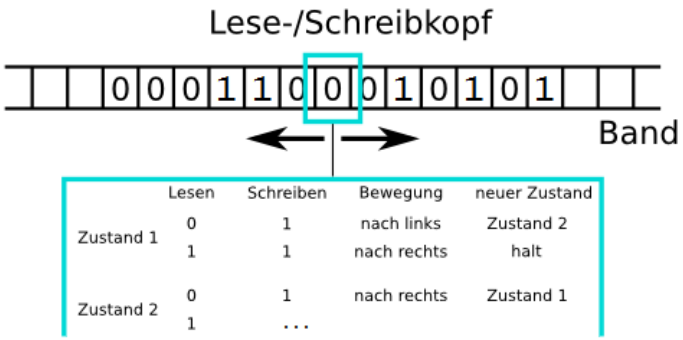


Abbildung 1.5: TURING-Maschine

Für die Arbeitsweise einer TURING-Maschine gelten folgende Konventionen:

- Sei Q die (endliche) Menge aller Zustände. Ein ausgewiesener Zustand $q_0 \in Q$ heißt *Startzustand*. Die TURING-Maschine startet in diesem Zustand q_0.
- Sei Σ die (endliche) Menge möglicher Eingabesymbole. Die Eingabe besteht nur aus solchen Symbolen und befindet sich initial auf dem Band. Der Schreib-/Lesekopf steht initial auf dem ersten (am weitesten links stehenden) Eingabesymbol.
- Zellen des Bands können auch leer sein. Dies wird durch das *Leersymbol* $\square \notin \Sigma$ dargestellt. Damit gilt für das (endliche) *Bandalphabet* Γ, dass $\Sigma \cup \{\square\} \subseteq \Gamma$.
- Ein einzelner Bearbeitungsschritt einer TURING-Maschine wird durch einen *Zustandsübergang* beschrieben. Er wird stets in der gleichen Weise formuliert:

  > Wenn sich die TURING-Maschine im Zustand q befindet und der Schreib-/Lesekopf in seiner augenblicklichen Position das Symbol a liest, dann soll der Schreib-/Lesekopf das Symbol b anstelle von a in die Zelle schreiben, sich in die Richtung R (für rechts), L (für links) oder gar nicht (H für *hold*) bewegen und die TURING-Maschine danach in den Zustand p übergehen.

  Mathematisch wird dies durch eine *Zustandsübergangsfunktion* δ dargestellt. δ ist folgendermaßen definiert:

  $$\delta : (Q \times \Gamma) \longrightarrow (\Gamma \times \{R, L, H\} \times Q)$$

  wobei $\delta(q, a) = (b, M, p)$ die obige Formulierung ausdrückt (M ist hier ein Platzhalter für R, L oder H).
- Die TURING-Maschine stoppt, wenn sie einen Zustand aus der ausgewiesenen Teilmenge $F \subseteq Q$ aller Zustände erreicht. Die Zustände aus F heißen *Endzustände*.

Die formale Definition einer TURING-Maschine ist dementsprechend durch die Auflistung aller dieser maßgeblichen Objekte gegeben [3].

Definition Eine TURING-Maschine TM ist ein 7-Tupel

$$\text{TM} = (Q, \Sigma, \Gamma, \delta, q_0, \square, F).$$

Betrachten wir die Arbeitsweise einer TURING-Maschine an einem einfachen Beispiel.

Beispiel 1.6 Eine *Verdopplungsmaschine* [4] verdoppelt in einer Eingabe Nullen, die zwischen zwei Einsen eingeschlossen sind. Zulässige Eingaben sind 11, 101, 1001, ...
Entsprechende Ergebnis-Bandbelegungen sind 11, 1001 (für 101), 100001 (für 1001), ...

Die Arbeitsweise kann folgendermaßen beschrieben werden:

1. Der Schreib-/Lesekopf steht im Startzustand auf der linken Eins. Er überschreibt diese mit dem Leersymbol und bewegt sich um eine Zelle nach rechts

$$\delta(q_0, 1) = (\square, R, q_1)$$

2. Findet er dort eine Eins (also das Ende der Eingabe), so geht er bis zur ersten leeren Zelle nach rechts, schreibt dort eine Eins, und die Verdopplungsmaschine hält an (d. h. sie geht in ihren einzigen Endzustand f)[6]

$$\begin{aligned}\delta(q_1, 1) &= (1, R, q_2)\\ \delta(q_2, 0) &= (0, R, q_2)\\ \delta(q_2, 1) &= (1, R, q_2)\\ \delta(q_2, \square) &= (1, H, f)\end{aligned}$$

3. Findet der Schreib-/Lesekopf jedoch eine Null, so streicht er diese, geht bis zur ersten leeren Zelle nach rechts und schreibt dort einen Null, ...

$$\begin{aligned}\delta(q_1, 0) &= (\square, R, q_3)\\ \delta(q_3, 0) &= (0, R, q_3)\\ \delta(q_3, 1) &= (1, R, q_3)\\ \delta(q_3, \square) &= (0, R, q_4)\end{aligned}$$

4. ... geht dann eine weitere Zelle nach rechts und schreibt dort eine zweite Null – die verdoppelte Null, um anschließend das Band nach links bis zur ersten nicht-leeren Zelle zurück zu gehen. Dann fährt die Verdopplungsmaschine fort mit Schritt 2.

$$\begin{aligned}\delta(q_4, \square) &= (0, L, q_5)\\ \delta(q_5, 0) &= (0, L, q_5)\\ \delta(q_5, 1) &= (1, L, q_5)\\ \delta(q_5, \square) &= (\square, R, q_1)\end{aligned}$$

Die Verdopplungsmaschine wird also durch folgende TURING-Maschine beschrieben:

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5, f\}$$

$$\Sigma = \{0, 1\}$$

$$\Gamma = \{0, 1, \square\}$$

$$F = \{f\}$$

[6] Um das erste Beispiel so einfach wie möglich zu halten, wird darauf verzichtet, eventuelle Fehler bspw. beim Eingabeformat abzuprüfen. Die Arbeitsvorschrift kann jedoch einfach entsprechend ergänzt werden, indem im erkannten Fehlerfall etwa ein Übergang in einen zweiten (Fehler-) Endzustand e vorgesehen wird.

δ	0	1	$\square$
q_0	–	$\square, R, q_1$	–
q_1	$\square, R, q_3$	$1, R, q_2$	–
q_2	$0, R, q_2$	$1, R, q_2$	$1, H, f$
q_3	$0, R, q_3$	$1, R, q_3$	$0, R, q_4$
q_4	–	–	$0, L, q_5$
q_5	$0, L, q_5$	$1, L, q_5$	$\square, R, q_1$
f	–	–	–

Die Zustandsübergangsfunktion δ wird oft auch durch ein *Zustandsdiagramm* dargestellt. Abbildung 1.6 zeigt ein solches Diagramm für die Verdopplungsmaschine

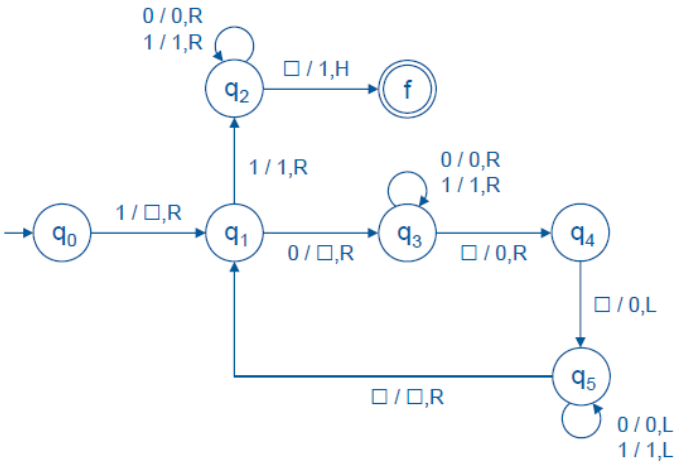


Abbildung 1.6: Zustandsdiagramm für die Verdopplungsmaschine

Die Zustandsübergangsfunktion verkörpert die *Steuerung* der TURING-Maschine. Sie kann auch als ihr *Programm* interpretiert werden. □

Wir wollen noch ein zweites Beispiel betrachten, das etwas eher der Vorstellung einer Rechenoperation entspricht.

Beispiel 1.7 Eine *Inkrementiermaschine* nimmt eine Dezimalzahl als Eingabe und addiert den Wert 1 [5].

Entsprechend unserer Konvention steht der Schreib-/Lesekopf initial auf der höchstwertigen Ziffer der Eingabe. Er muss also zuerst auf die niederwertigste Ziffer bewegt werden. Das geschieht dadurch, dass bei jeder gelesenen Ziffer um einen Schritt nach rechts gegangen wird, bis das Leersymbol erreicht ist. Dann bewegt sich der Schreib-/Lesekopf um eine Position nach links.

$$\begin{aligned}
\delta(q_0, 0) &= (0, R, q_0) \\
\delta(q_0, 1) &= (1, R, q_0) \\
&\vdots \\
\delta(q_0, 9) &= (9, R, q_0) \\
\delta(q_0, \square) &= (\square, L, q_1)
\end{aligned}$$

Nun wird die Ziffer an dieser Position gelesen. Ist es eine Ziffer zwischen 0 und 8, so wird sie durch die nächsthöhere Ziffer ersetzt, in den Endzustand f übergegangen und angehalten. Nur bei der Ziffer 9 wird diese durch die 0 ersetzt, zur nächsthöherwertigen Stelle nach links gegangen und dieselbe Behandlung wiederholt.

$$\begin{aligned}
\delta(q_1, 0) &= (1, H, f) \\
\delta(q_1, 1) &= (2, H, f) \\
&\vdots \\
\delta(q_1, 8) &= (9, H, f) \\
\delta(q_1, 9) &= (0, L, q_1)
\end{aligned}$$

Wird sich auf diese Art nach links über die höchstwertige Stelle hinaus bewegt, so muss eine neue Stelle mit einer führenden 1 geschaffen werden.

$$\delta(q_1, \square) = (1, H, f)$$

Diese TURING-Maschine besitzt nur drei Zustände:

$$Q = \{q_0, q_1, f\}$$

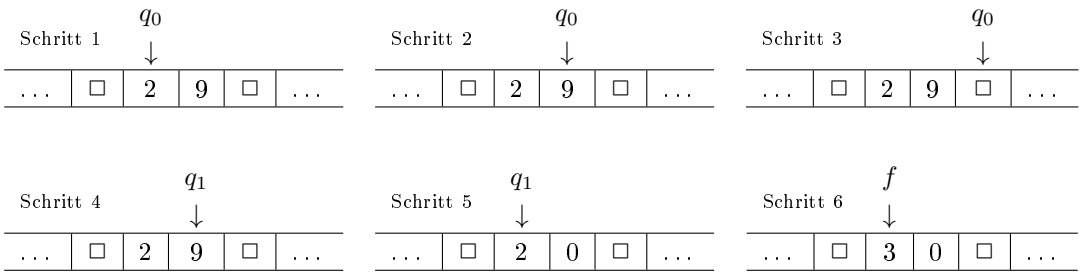
$$\Gamma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \square\}$$

δ	0	1	2	...	8	9	$\square$
q_0	$0, R, q_0$	$1, R, q_0$	$2, R, q_0$	...	$8, R, q_0$	$9, R, q_0$	$\square, L, q_1$
q_1	$1, H, f$	$2, H, f$	$3, H, f$	...	$9, H, f$	$0, L, q_1$	$1, H, f$
f	–	–	–	...	–	–	–

□

Der Ablauf der einzelnen Berechnungschritte, die von einer TURING-Maschine ausgeführt werden, kann durch eine Folge von „Momentaufnahmen“ fortlaufend notiert werden.

Für die Inkrementiermaschine aus Beispiel 1.7 ergeben sich bei einer Eingabe „29“ folgende Momentaufnahmen:



Die einzelne Momentaufnahme heißt *Konfiguration* der TURING-Maschine.

Definition Eine Konfiguration einer TURING-Maschine TM ist ein Tripel

$$k \in \Gamma^* \times Q \times \Gamma^*.$$

Hierbei wird $k = \alpha q \beta$ so interpretiert, dass $\alpha\beta$ den gesamten, nicht-leeren Bandinhalt darstellt, der Schreib-/Lesekopf auf dem ersten Symbol von β steht und q den Zustand angibt, in dem sich TM gerade befindet.

Schritt 2 würde also durch die Konfiguration $k_2 = 2q_09$ beschrieben, Schritt 3 durch $k_3 = 29q_0$. Die Startkonfiguration ist $k_1 = q_029$.

Bezeichne $KONF_{TM}$ die Menge aller Konfigurationen einer TURING-Maschine TM.

Definition Die *Schrittrelation* $\vdash \subseteq KONF_{TM} \times KONF_{TM}$ stellt den Übergang von einer Konfiguration zu einer Nachfolgekonfiguration dar, d. h. $k \vdash k'$ gilt genau dann, wenn für

$$k = \alpha aqb\beta \quad \text{mit } \alpha, \beta \in \Gamma^*,\ a, b \in \Gamma,\ q \in Q$$

und $c \in \Gamma$, $q' \in Q$ folgendes gilt:

$$k' = \begin{cases} \alpha acq'\beta & \text{falls } \delta(q,b) = (c, R, q') \\ \alpha q'ac\beta & \text{falls } \delta(q,b) = (c, L, q') \\ \alpha aq'c\beta & \text{falls } \delta(q,b) = (c, H, q') \end{cases}$$

Beispiel 1.8 Erhält die Inkrementiermaschine aus Beispiel 1.7 die Eingabe „99“, so durchläuft sie folgende Konfigurationen:

$$\begin{aligned} & q_099 \\ \vdash\ & 9q_09 && \text{da} && \delta(q_0, 9) = (9, R, q_0) \\ \vdash\ & 99q_0 && \text{da} && \delta(q_0, 9) = (9, R, q_0) \\ \vdash\ & 9q_19 && \text{da} && \delta(q_0, \square) = (\square, L, q_1) \\ \vdash\ & q_190 && \text{da} && \delta(q_1, 9) = (0, L, q_1) \\ \vdash\ & q_1\square 00 && \text{da} && \delta(q_1, 9) = (0, L, q_1) \\ \vdash\ & f100 && \text{da} && \delta(q_1, \square) = (1, H, f) \end{aligned}$$

□

Folgen von Konfigurationen $k_1, k_2, k_3, \ldots$, die in der Relation $k_1 \vdash k_2 \vdash k_3 \vdash \ldots$ stehen, heißen *Berechnungen*.

Endliche Berechnungen $k_1, k_2, \ldots, k_t$ heißen *erfolgreiche* Berechnungen, wenn $k_1 \in \Gamma^* \times \{q_0\} \times \Gamma^*$ und $k_t \in \Gamma^* \times F \times \Gamma^*$ ist.

Auf dieser Grundlage wird der Begriff der TURING-Berechenbarkeit definiert.

Wenn ein Algorithmus Eingaben aus einer Menge X akzeptiert und Ausgaben aus einer Menge Y erzeugt, dann berechnet er eine (evtl. partielle[7]) Funktion $f : X \to Y$.

Definition Eine Funktion $f : \Sigma^* \to \Gamma^*$ heißt **TURING-berechenbar**, falls es eine TURING-Maschine TM gibt, so dass für alle $x \in \Sigma^*$ und $y \in \Gamma^*$ gilt:

$$f(x) = y \quad \text{genau dann, wenn} \quad q_0x \vdash^* q_ty$$

wobei $q_t \in F$.

[7] Eine **partielle** Funktion bildet jedes Element der Definitionsmenge X auf *höchstens* ein Element der Zielmenge Y ab. Eine *totale* Funktion hingegen bildet jedes Element von X auf *genau* ein Element von Y ab. Somit ist jede totale Funktion auch eine partielle Funktion, aber nicht umgekehrt. Um eine Funktion als partielle Funktion zu kennzeichnen, schreibt man oft auch: $f : X \rightharpoonup Y$.

Mit anderen Worten: f ist TURING-berechenbar, wenn es eine TM gibt, die f realisiert, d. h. bei Eingabe von $x \in \Sigma^*$ eine erfolgreiche Berechnung des Funktionswerts $f(x) \in \Gamma^*$ durchführt und stoppt, oder (das ist implizit ausgesagt), falls $f(x)$ *undefiniert* ist (was bei einer partiellen Funktion möglich ist), auch in eine unendliche Schleife gehen kann.

Es stellt sich nun die Frage, ob es auch *nicht-berechenbare* Funktionen gibt. Es ist nämlich gar nicht so leicht, Funktionen zu finden, die nicht-berechenbar sind.

Beispiel 1.9 Wir betrachten zwei Funktionen, bei denen man vermuten könnte, dass sie unberechenbar wären, sie es aber nicht sind.

i. Die überall undefinierte Funktion Ω ist berechenbar. Eine entsprechende TURING-Maschine ist eine Maschine die nie anhält, etwa:

$$\delta(q_0, a) = (q_0, a, R) \quad \text{für alle } a \in \Gamma$$

ii. Die Kreiszahl π ist bekanntlich eine irrationale Zahl, deren Darstellung in jedem Zahlensystem unendlich lang und nicht periodisch ist. Sei h eine Funktion, die prüft, ob die Ziffer 5 in der Dezimaldarstellung von π n-mal hintereinander vorkommt, wobei n eine beliebige (natürliche) Eingabe darstellt:

$$h(n) = \begin{cases} 1 & \text{falls es eine 5er-Kette der Länge } n \text{ in } \pi \text{ gibt} \\ 0 & \text{sonst} \end{cases}$$

Auch h ist berechenbar. Falls es beliebig lange 5er-Ketten in π gibt, ist $h(n) = 1$ für alle n. Diese Funktion ist berechenbar. Falls nicht, gibt es eine längste 5er-Kette und diese hat die Länge k. Dann ist

$$h(n) = \begin{cases} 1 & \text{falls } n \leq k \\ 0 & \text{sonst} \end{cases}$$

Auch diese Funktion ist für alle k berechenbar. Niemand weiß zwar, welche dieser berechenbaren Funktionen die Richtige ist, aber für die Aussage der Berechenbarkeit ist nur wichtig, dass eine TURING-Maschine existiert, die die Funktion realisiert. □

Kann es also überhaupt nicht-berechenbare Funktionen geben?

Hierzu stellen wir folgende Überlegung an: Jede TURING-Maschine ist durch einen endlichen Text beschreibbar (bspw. die Funktionstabelle der Übergangsfunktion). Nehmen wir an, dass der ASCII-Code mit 128 Zeichen als Zeichensatz für Textbeschreibungen genügt. Es gibt 128 verschiedene Texte der Länge 1, 128^2 verschiedene Texte der Länge 2, 128^3 verschiedene Texte der Länge 3, usw. Schreiben wir alle verschiedenen Texte bis zu einer Länge n nacheinander auf eine Liste, so wird diese Liste die Länge $128 + 128^2 + \ldots + 128^n$ haben. Natürlich wird nicht jeder Text eine sinnvolle TURING-Maschine beschreiben und n darf beliebig groß werden, aber wir können erkennen, dass wir jeden Text (und damit auch jede TURING-Maschine) einer natürlichen Zahl – nämlich der entsprechenden laufenden Nummer auf der Liste – zuordnen können.

Die Menge aller TURING*-Maschinen ist abzählbar unendlich.*

Dagegen ist die Menge aller Funktionen überabzählbar. Man kann mit einem ähnlichen Ansatz wie oben zeigen, dass für ein Alphabet Σ gilt: Ist Σ endlich, so ist Σ^* abzählbar unendlich. Somit genügt es, die Funktionen $f : \mathbb{N} \to \mathbb{N}$ zu betrachten.

Nehmen wir an, die Menge aller dieser Funktionen wäre abzählbar. Dann könnten wir all diese Funktionen in einer Liste aufzählen: $f_1, f_2, f_3, f_4, \ldots$ Jede dieser Funktionen nimmt eine natürliche Zahl als Argument und ordnet ihr einen Funktionswert zu[8]. Folgende Zuordnungstabelle stellt dies dar:

[8] Wir betrachten hier nur totale Funktionen. Die Mächtigkeit der Menge der partiellen Funktionen $\mathbb{N} \to \mathbb{N}$ ist noch größer.

	1	2	3	4	...
f_1	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	
f_2	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	
f_3	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$	
f_4	$f_4(1)$	$f_4(2)$	$f_4(3)$	$f_4(4)$	
$\vdots$					

Nun definieren wir eine Funktion $g : \mathbb{N} \to \mathbb{N}$ wie folgt:

$$g(n) = f_n(n) + 1.$$

Offenbar stimmt diese Funktion g mit keiner der aufgelisteten Funktionen überein, da für alle $n \in \mathbb{N}$ mindestens der Funktionswert $f_n(n) \neq g(n)$ ist. Damit führt die Annahme, dass wir in unserer Liste alle Funktionen $\mathbb{N} \to \mathbb{N}$ aufgezählt haben, zu einem Widerspruch. Dieser Widerspruch kann nur dadurch aufgelöst werden, dass die Annahme falsch war. Die Menge der Funktionen $\mathbb{N} \to \mathbb{N}$ ist also *überabzählbar unendlich*.

Somit muss es nicht-berechenbare Funktionen geben!

Beispiel 1.10 Eine nicht-berechenbare Funktion **bb** wurde 1962 vom ungarischen Mathematiker Tibor Radó entdeckt. Radó betrachtete spezielle Turing-Maschinen, die aus genau einem Endzustand und n weiteren Zuständen bestehen. Das Bandalphabet besteht (neben dem Leersymbol) aus genau einem Symbol „|“ und die H-Bewegung ist nicht erlaubt.

Eine solche Maschine mit n Zuständen (ohne den Endzustand), die mit dem leeren Band beginnt und nach einer endlichen Anzahl von Schritten hält, heißt *Biber mit n Zuständen*.

Ein Biber mit n Zuständen, der die *maximale* Anzahl von Strichsymbolen auf das leere Band schreibt, heißt *fleißiger Biber mit n Zuständen*. Die Zahl seiner erzeugten Striche wird mit $\mathbf{bb}(n)$ (engl. *busy beaver*) bezeichnet. Die Funktion $\mathbf{bb} : \mathbb{N} \to \mathbb{N}$ heißt Radó-Funktion.

Für $n = 1$ könnten wir nun spontan einen Entwurf für eine Turing-Maschine vorschlagen, die sehr viele Striche erzeugt, beispielsweise:

Leider kommt diese Maschine nie zum Halt und ist deswegen keine gültige Lösung. Ein zweiter Versuch erzeugt nur einen einzigen Strich, ist dafür aber ein gültiger Biber:

Die spannende Frage ist, wieviele Striche ein *fleißiger* Biber mit $n = 1$ erzeugen würde.

Man kann dies in folgender Weise herausbekommen [6]: Die Überführungsfunktion δ einer solchen Turing-Maschine ist wie folgt definiert:

$$\delta : (Q \backslash F \times \Gamma) \longrightarrow (\Gamma \times \{R, L\} \times Q).$$

Da die Mächtigkeit der Menge $Q \backslash F$ gleich 1 ist und die Mächtigkeit der Menge $\Gamma = \{|, \square\}$ gleich 2, hat die Quellmenge von δ $1 \cdot 2 = 2$ Elemente.

Entsprechend überlegt man sich, dass die Zielmenge $2 \cdot 2 \cdot 2 = 8$ Elemente hat.

Nun weiß man, dass die Anzahl der Abbildungen aus einer Quellmenge mit m Elementen in eine Zielmenge mit n Elementen gleich n^m ist[9]. Die Anzahl der möglichen TURING-Maschinen ist dann gegeben durch die Anzahl der Abbildungen, also $8^2 = 64$.

Erzeugt man systematisch alle diese 64 TURING-Maschinen, so erkennt man, dass viele Maschinen gar nicht halten (wie beispielsweise unser erster Entwurf oben). Sortiert man diese Maschinen aus, bleiben nur Biber übrig. Diese simuliert man und notiert jeweils die erzeugte Strichzahl. Die maximale Strichzahl bestimmt den Wert der Funktion **bb**.

Es zeigt sich, dass unser zweiter Entwurf bereits ein fleißiger Biber für $n = 1$ ist: mehr als einen Strich schafft kein Biber, d. h. $\mathbf{bb}(1) = 1$.

In gleicher Weise behandelt man den allgemeinen Fall mit $n > 1$. Die Quellemenge hat die Mächtigkeit $2n$ und die Zielmenge die Mächtigkeit $2{\cdot}2{\cdot}(n{+}1)$. Die Anzahl der möglichen TURING-Maschinen ist dann $(4 \cdot (n+1))^{2n}$.

Tabelle 1.1 zeigt in Abhängigkeit von n die Anzahl der möglichen TURING-Maschinen sowie die bisher ermittelten Funktionswerte der Funktion **bb**.

n	Anzahl der TM	$\mathbf{bb}(n)$
1	64	1
2	20.736	4
3	16.777.216	6
4	$2,56 \cdot 10^{10}$	13
5	$\approx 6,34 \cdot 10^{13}$	≥ 4098
6	$\approx 2,32 \cdot 10^{17}$	$\geq 1,29 \cdot 10^{865}$
7	$\approx 1,18 \cdot 10^{21}$	?

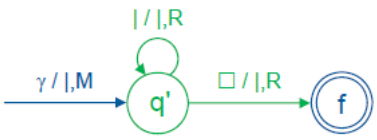
Tabelle 1.1: RADÓ-Funktion

Man erkennt, dass die Anzahl der TURING-Maschinen, die man untersuchen muss, extrem schnell ansteigt. Aus diesem Grund kennt man auch erst die ersten 4 Werte der RADÓ-Funktion. Für alle größeren n gibt es nur Schätzungen.

Was man jedoch zeigen kann, ist, dass die RADÓ-Funktion streng monoton wächst.

Nehmen wir dazu ein beliebiges $n \in \mathbb{N}$ an. Der zugehörige fleißige Biber besitzt einen Zustandsgraph, der folgendermaßen endet (in der Kantenbeschriftung steht γ für einen Strich oder ein Leersymbol und M für eine Kopfbewegung):

Man konstruiert daraus einen Biber mit $n + 1$ Zuständen, der $\mathbf{bb}(n) + 1$ Striche schreibt, indem man einen Zustand q' wie folgt hinzufügt:



[9] Man veranschaulicht sich diesen Zusammenhang am einfachen Beispiel einer Funktion $f : \{1, 2\} \to \{a, b, c\}$. Für $(f(1), f(2))$ gibt es die $3^2 = 9$ Möglichkeiten (a, a), (a, b), (a, c), (b, a), (b, b), (b, c), (c, a), (c, b), (c, c).

Ein fleißiger Biber mit $n+1$ Zuständen schreibt also mindestens $\mathbf{bb}(n)+1$ Striche und damit ist gezeigt, dass $\mathbf{bb}$ streng monoton wächst.

Über dieses streng monotone Wachstum wird schließlich der Beweis geführt, dass die RADÓ-Funktion nicht berechenbar ist.

Angenommen, es gibt einen Algorithmus, der für jede natürliche Zahl n den Wert $\mathbf{bb}(n)$ berechnet. Das heißt, es gibt eine TURING-Maschine B mit endlich vielen Zuständen, die auf einem Band, auf dem anfangs die (unäre) Zahl n steht, nach endlich vielen Schritten die (unäre) Zahl $\mathbf{bb}(n)$ hinterläßt. Die Anzahl der Zustände dieser Maschine B sei gleich m.

Eine TURING-Maschine A, die genau n Striche auf einem ursprünglich leeren Band erzeugt, kann mit $\lfloor \frac{n}{2} \rfloor + 10$ Zuständen konstruiert werden (siehe Übung 2).

Schaltet man die beiden Maschinen A und B hintereinander und setzt sie auf das leere Band an, so schreibt die Maschine AB genau $\mathbf{bb}(n)$ Striche auf das Band und hält dann an. AB ist also ein Biber mit $\lfloor \frac{n}{2} \rfloor + 10 + m$ Zuständen. Weiter leistet AB dasselbe wie ein fleißiger Biber mit n Zuständen.

Ist nun aber n so groß, dass

$$n > \lfloor \frac{n}{2} \rfloor + 10 + m$$

so schreibt AB mit weniger Zuständen dieselbe Anzahl von Strichen aufs Band wie der fleißige Biber mit n Zuständen.

Dies steht im Widerspruch zur strengen Monotonie der RADÓ-Funktion. □

Die TURING-Maschine ist ein abstraktes Modell eines einfachen Rechners. Es könnte sich die Frage stellen, ob die Mächtigkeit eines solchen Modells überhaupt mit der Mächtigkeit einer Programmiersprache vergleichbar ist.

Die Antwort liegt auf der Hand: Da Programmiersprachen in Maschinensprache übersetzt werden, können Programmiersprachen nicht mehr leisten als Maschinen. Die Frage ist also eher, ob und in wie weit Programmiersprachen genauso mächtig sind wie bspw. TURING-Maschinen.

Im folgenden Abschnitt wollen wir daher formale Berechnungsmodelle kennenlernen, die sich eher an Programmiersprachen anlehnen.

1.3.2 LOOP-, WHILE-, GOTO-Berechenbarkeit

Bei der Einführung der Struktogramme wurde bereits auf das BÖHM–JACOPINI Theorem hingewiesen, das aussagt, dass Algorithmen durch nur drei Kontrollstrukturen formuliert werden können: Sequenz, Selektion und Iteration.

Iterationen können auf verschiedene Arten bewerkstelligt werden. In Programmiersprachen sind die **for**-Schleife, die **while**-Schleife und der **goto**-Sprung übliche Konstrukte.

Betrachten wir nun Programmiersprachen mit einer sehr einfachen, auf das Wesentliche reduzierten Syntax, die jeweils eine der drei Iterationsformen abbilden.

Die erste dieser Programmiersprachen heiße **LOOP**. Sie berechnet Funktionen über $\mathbb{N}$ und verfügt über eine unbeschränkte Menge von Variablen.

LOOP-Programme sind aus folgenden syntaktischen Elementen aufgebaut:

Wertzuweisungen:

$x_i := x_j + c$
$x_i := x_j \dot{-} c$

wobei $x_i, x_j \in \mathbb{N}$ Variablen und $c \in \mathbb{N}$ Konstante sind und „$\dot{-}$“ die sogenannte modifizierte Subtraktion beschreibt. Die modifizierte Differenz $a \dot{-} b$ ist definiert als $\max\{a - b,\ 0\}$.
Jede Wertzuweisung ist ein LOOP-Programm.

Hintereinanderausführung:

$P_1; P_2$

wobei P_1, P_2 LOOP-Programme sind, die hintereinander ausgeführt werden. Auch eine Hintereinanderausführung ist ein LOOP-Programm.

LOOP-Iterationen:

LOOP x DO P END

wobei x eine Variable und P ein LOOP-Programm sind. P wird x-mal ausgeführt. Auch eine Iteration ist ein LOOP-Programm.
Der Wert von x wird zu Beginn der Iteration festgestellt und bestimmt die Anzahl der Iterationen, unabhängig davon, ob x in P verändert wird.

Obgleich die LOOP-Programmiersprache nur sehr wenige, allgemeine Sprachkonstrukte definiert, ist es möglich, sich daraus weitere Konstrukte abzuleiten, die im Weiteren als Abkürzungen verwendet werden dürfen.

Solche Abkürzungen sind beispielsweise:

Abkürzung	LOOP-Programm
$x_i := x_j$	$x_i := x_j + 0$
$x_i := c$	$x_i := x_j + c$, wobei x_j eine Variable mit Wert 0 ist
IF $x = 0$ THEN P END	$y := 1$; LOOP x DO $y := 0$ END; LOOP y DO P END
IF $x = 0$ THEN P_1 ELSE P_2 END	$y := 1; z := 0$; LOOP x DO $y := 0; z := 1$ END; LOOP y DO P_1 END; LOOP z DO P_2 END

Natürlich lassen sich Funktionen als LOOP-Programm formulieren.

Definition Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt *LOOP-berechenbar*, falls es ein LOOP-Programm P gibt, das f berechnet, d. h. das mit den Eingabewerten $n_1, n_2, \ldots, n_k$ in den Variablen $x_1, x_2, \ldots, x_k$ (und dem Wert 0 in allen anderen Variablen) gestartet wird und mit dem Ergebnis $f(n_1, n_2, \ldots, n_k)$ in der Variablen x_0 anhält.

LOOP-Programme halten offenbar immer an, da jede LOOP-Schleife nur endlich oft durchlaufen wird und es in einem LOOP-Programm auch nur endlich viele LOOP-Schleifen geben kann.

Somit können mit einem LOOP-Programm nur totale Funktionen berechnet werden.

Durch eine Erweiterung um ein einziges zusätzliches Konstrukt entsteht die **WHILE**-Programmiersprache. Dieses Konstrukt ist:

$$\text{WHILE } x_i \neq 0 \text{ DO } P \text{ END}$$

wobei P ein LOOP- bzw. WHILE-Programm darstellt und x_i eine Variable. Die Semantik dieses neuen Konstrukts ist so definiert, dass das Programm P solange wiederholt ausgeführt wird, wie der Wert von x_i ungleich Null ist.

Streng genommen könnte man in WHILE-Programmen das LOOP-Konstrukt auch weglassen, denn LOOP x DO P END kann simuliert werden durch:

$$y := x;$$
$$\text{WHILE } y \neq 0 \text{ DO } y := y \dot{-} 1; P \text{ END}$$

Definition Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt *WHILE-berechenbar*, falls es ein WHILE-Programm P gibt, das f berechnet, d. h. das mit den Eingabewerten $n_1, n_2, \ldots, n_k$ in den Variablen $x_1, x_2, \ldots, x_k$ (und dem Wert 0 in allen anderen Variablen) gestartet wird und, falls $f(n_1, n_2, \ldots, n_k)$ definiert ist, mit diesem Ergebnis in der Variablen x_0 stoppt oder, falls $f(n_1, n_2, \ldots, n_k)$ *undefiniert* ist, niemals anhält.

Wir können damit zwei Eigenschaften feststellen:

Ein WHILE-Programm hat eine höhere Berechnungsmächtigkeit als ein LOOP-Programm: es kann auch partielle Funktionen berechnen.

Jede LOOP-berechenbare Funktion ist auch WHILE-berechenbar.

Das dritte Modell, die **GOTO**-Programmiersprache orientiert sich eher an der Maschinenprogrammierung tatsächlicher Rechnerarchitekturen. GOTO-Programme bestehen aus Sequenzen von Anweisungen A_i, die jeweils durch eine Marke M_i adressiert werden:

$$M_1 : A_1;\ M_2 : A_2;\ \ldots\ M_k : A_k$$

Mögliche Anweisungen sind:

Wertzuweisungen:

(genauso definiert, wie bei LOOP- und WHILE-Programmen)

unbedingter Sprung:

GOTO M_i

legt fest, dass die nächste Anweisung diejenige ist, die durch die Marke M_i adressiert ist.

bedingter Sprung:

IF $x_i = c$ THEN GOTO M_i

wie der unbedingte Sprung, falls der Wert der Variablen x_i gleich c ist, sonst wird die jeweils folgende Anweisung abgearbeitet.

Stopanweisung:

HALT

beendet ein GOTO-Programm.

Marken vor Anweisungen dürfen bei der Formulierung von GOTO-Programmen weggelassen werden, wenn sie von keinem GOTO verwendet werden.

Es ist leicht erkennbar, dass GOTO-Programme auch in unendliche Schleifen geraten können (M_i : GOTO M_i).

Die Definition der GOTO-Berechenbarkeit erfolgt analog zur WHILE-Berechenbarkeit.

Folgende Vermutung ist naheliegend:
Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden.

Hierzu genügt es, das WHILE-Konstrukt zu betrachten (Wertzuweisungen und Hintereinanderausführungen sind in beiden Ansätzen gleich definiert). Für WHILE $x \neq 0$ DO P END gibt es folgende GOTO-Entsprechung:

M_1: IF $x = 0$ GOTO M_2;
P;
GOTO M_1;
M_2: ...

Etwas weniger offensichtlich kann auch die umgekehrte Richtung gezeigt werden.

Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden.

Betrachten wir das GOTO-Programm $M_1 : A_1;\ M_2 : A_2;\ \ldots\ M_k : A_k$. Das Programm wird simuliert durch folgendes WHILE-Programm:

$cnt := 1$;
WHILE $cnt \neq 0$ DO
  IF $cnt = 1$ THEN A'_1 END;
  IF $cnt = 2$ THEN A'_2 END;
  ...
  IF $cnt = k$ THEN A'_k END;
END

Hierbei gilt für die modifizierten Anweisungen:

$$A'_i = \begin{cases} x_i := x_j \pm c; cnt := cnt + 1 & \text{falls } A_i = x_i := x_j \pm c \\ cnt := j & \text{falls } A_i = \text{GOTO } M_j \\ \text{IF } x = c \text{ THEN } cnt := j & \\ \quad \text{ELSE } cnt := cnt + 1 \text{ END} & \text{falls } A_i = \text{IF } x = c \text{ THEN GOTO } M_j \\ cnt := 0 & \text{falls } A_i = \text{HALT} \end{cases}$$

Dieses Simulationsprogramm hat nur eine einzige WHILE-Schleife.

Würde man ein beliebiges WHILE-Programm (mit mehreren, ggf. sogar ineinander verschachtelten) WHILE-Schleifen zunächst in ein GOTO-Programm transformieren und dieses wieder zurückwandeln, so wird eine Erkenntnis deutlich, die als Kleenesche[10] Normalform für WHILE-Programme bekannt ist:

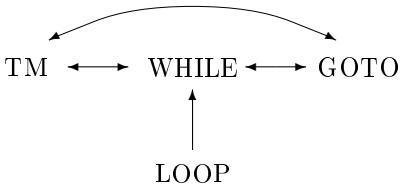
> Jede WHILE-berechenbare Funktion kann durch ein WHILE-Programm mit nur einer einzigen WHILE-Schleife berechnet werden.

Darüber hinaus haben wir nunmehr gezeigt, dass *WHILE- und GOTO-Programme die gleiche Berechnungsmächtigkeit besitzen.*

[10] nach Stephen C. Kleene, US-amerikanischer Mathematiker und Logiker

Interessanterweise lässt sich auch beweisen, dass dies dieselbe Berechnungsmächtigkeit ist wie von TURING-Maschinen.

Der Beweis soll an dieser Stelle nicht im Detail aufgerollt werden, da er vergleichsweise aufwändig ist. Er umfasst mehrere Schritte. Zuerst wird gezeigt, dass sogenannte Mehrband-TURING-Maschinen[11] WHILE-Programme simulieren können. Da Mehrbandmaschinen aber auf Einbandmaschinen abgebildet werden können, heißt das, dass jede WHILE-berechenbare Funktion auch TM-berechenbar ist. In einem zweiten Schritt wird gezeigt, dass GOTO-Programme TURING-Maschinen simulieren können. Damit ist jede TM-berechenbare Funktion auch GOTO-berechenbar. Aufgrund der Gleichmächtigkeit von WHILE- und GOTO-Sprachen ist damit bewiesen, dass beide auch dieselbe Mächtigkeit wie TURING-Maschinen besitzen, d. h. jede WHILE-berechenbare Funktion und jede GOTO-berechenbare Funktion ist auch TM-berechenbar und umgekehrt:



Zur LOOP-Berechenbarkeit bleibt abschließend festzuhalten, dass sie nur totale Funktionen berechnen kann (das wurde oben bereits festgehalten), es aber totale Funktionen gibt, die *nicht* LOOP-berechenbar sind.

Das klassische Beispiel hierfür ist die ACKERMANN-Funktion[12], eine Funktion, die ähnlich wie die RADÓ-Funktion sehr schnell wächst.

Definition Die ACKERMANN-Funktion $A : \mathbb{N}^2 \to \mathbb{N}$ ist durch die Folge:

$$\begin{array}{lllll} (1) & A(0,\, n) & = n+1 & & \text{für } n \geq 0 \\ (2) & A(m+1,\, 0) & = A(m,\, 1) & & \text{für } m \geq 0 \\ (3) & A(m+1,\, n+1) & = A(m,\, A(m+1,\, n)) & & \text{für } m,\, n \geq 0 \end{array}$$

definiert.

Beispiel 1.11 Der Wert $A(1,\, 3)$ berechnet sich folgendermaßen:

$$\begin{aligned} A(1,3) &= A(0, A(1,2)) \\ &= A(0, A(0, A(1,1))) \\ &= A(0, A(0, A(0, A(1,0)))) \\ &= A(0, A(0, A(0, A(0,1)))) \\ &= A(0, A(0, A(0,2))) \\ &= A(0, A(0,3)) \\ &= A(0,4) \\ &= 5 \end{aligned}$$

□

Durch entsprechende weitere Berechnungen lässt sich eine Wertetabelle aufstellen (siehe Tabelle 1.2).

[11] Eine Mehrband-TURING-Maschine verfügt über mehrere Speicherbänder, die jeweils einen eigenen Schreib-/Lesekopf besitzen, wobei diese Schreib-/Leseköpfe unabhängig voneinander bewegt werden können. Ansonsten verhält sich eine Mehrband-TURING-Maschine genauso wie eine Einband-TURING-Maschine.

[12] benannt nach WILHELM ACKERMANN, der sie 1928 gefunden hat. Die ursprüngliche Funktionsdarstellung wird heute allerdings kaum mehr verwendet, sondern stattdessen eine vereinfachte Darstellung, die 1935 von RÓZSA PÉTER vorgeschlagen wurde.

	$n=0$	$n=1$	$n=2$	$n=3$	$n=k$
$A(0,n)$	1	2	3	4	$k+1$
$A(1,n)$	2	3	4	5	$k+2$
$A(2,n)$	3	5	7	9	$2k+3$
$A(3,n)$	5	13	29	61	$2^{k+3}-3$
$A(4,n)$	13	$2^{16}-3$	$2^{65536}-3$	$2^{2^{65536}}-3$	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{k+2 \text{ viele Potenzen}}-3$
$A(5,n)$	$2^{16}-3$	$2^{65536}-3$	$\ldots$		

Tabelle 1.2: Wertetabelle der ACKERMANN-Funktion

Die ACKERMANN-Funktion besitzt folgende Eigenschaften:

Die Berechnung von $A(m,n)$ terminiert für alle $m,n \in \mathbb{N}$, d. h. die Funktion A liefert stets eine natürliche Zahl als Ergebnis. Mit anderen Worten: A ist eine *totale* Funktion.

Man beweist diese Aussage mittels doppelter vollständiger Induktion.

- (Induktionsanfang$_m$) Für $m=0$ und alle n ist nach Definition (1) $A(0,n)=n+1$ immer definiert.
- (Induktionsvoraussetzung$_m$) $A(m,n)$ sei definiert für alle n.
- (Induktionsbehauptung$_m$) $A(m+1,n)$ ist definiert für alle n.
- (Induktionsschluss$_m$) Beweis der Induktionsbehauptung durch vollständige Induktion über n:
  - (Induktionsanfang$_n$) Der Fall $n=0$ gilt nach Definition (2) $A(m+1,0)=A(m,1)$ und $A(m,*)$ ist nach Induktionsvoraussetzung$_m$ immer definiert.
  - (Induktionsvoraussetzung$_n$) $A(m+1,n)$ sei definiert.
  - (Induktionsbehauptung$_n$) $A(m+1,n+1)$ ist definiert.
  - (Induktionsschluss$_n$) nach Definition (3) ist $A(m+1,n+1)=A(m,A(m+1,n))$. Der innere Ausdruck $A(m+1,n)$ ist nach Induktionsvoraussetzung$_n$ immer definiert. Damit ist es dann nach Induktionsvoraussetzung$_m$ auch der äußere Ausdruck $A(m,*)$.

Weiterhin besitzt A einige Monotonie-Eigenschaften, die man auch in Tabelle 1.2 gut erkennen kann:

A. $n < A(m,n)$

B. $A(m,n) < A(m,n+1)$

C. $A(m,n+1) \leq A(m+1,n)$

D. $A(m,n) < A(m+1,n)$

für alle $m,n \geq 0$.

Aus B und D folgt die allgemeine Monotonie-Eigenschaft: für alle $m \leq m'$ und alle $n \leq n'$ gilt $A(m,n) \leq A(m',n')$.

Die Beweise dieser Eigenschaften basieren wieder auf vollständiger Induktion und werden der Übung überlassen.

A ist nicht LOOP-berechenbar.

Man beweist dies, indem man zeigt, dass die ACKERMANN-Funktion schneller wächst als jede LOOP-berechenbare Funktion:

Sei P ein LOOP-Programm und seien $x_0, x_1, \ldots, x_k$ die in P vorkommenden Variablen. Den Variablen $x_1, x_2, \ldots, x_k$ werden die Anfangswerte $n_1, n_2, \ldots, n_k$ zugewiesen (alle restlichen Variablen haben den Wert 0). Die Endwerte, die nach Ablauf des Programms P in den Variablen stehen, bezeichnen wir als $n'_0, n'_1, \ldots, n'_k$.

Nun definieren wir eine Funktion $f_P : \mathbb{N} \to \mathbb{N}$, die die größtmögliche Summe aller Variablenendwerte nennt, wenn das Programm P mit Anfangswerten gestartet wird, die in ihrer Summe den Wert n nicht überschreiten:

$$f_P(n) = \max \left\{ \sum_{i=0}^{k} n'_i \;\middle|\; \sum_{i=0}^{k} n_i \leq n \right\}.$$

Diese Funktion beschreibt gewissermaßen die maximalen, durch ein LOOP-Programm P berechenbaren Zahlenwerte.

Im nächsten Schritt wollen wir zeigen, dass es für jedes LOOP-Programm P eine Zeile k in Tabelle 1.2 gibt, in der die Werte $A(k,n)$ mit steigendem n stärker wachsen als in der Funktion $f_P(n)$, d. h.

$$\text{es gibt ein } k, \text{ sodass} \quad f_P(n) < A(k,n) \quad \text{für alle } n.$$

Dies geschieht durch Induktion über den Aufbau des LOOP-Programms P:

- Falls P die Form einer Wertzuweisung $x_i := x_j \pm c$ hat, so dürfen wir ohne Beschränkung der Allgemeinheit annehmen, dass $c \in \{0,1\}$.[13]

  Da der Anfangswert von x_j maximal n sein kann (dann müssen alle anderen Variablen den Anfangswert 0 haben) und c maximal den Wert 1 annehmen kann, hat x_i nach Ausführung von P maximal den Wert $n+1$. Somit ist die Summe der Variablenwerte $x_i + x_j = 2n+1$. Eine größere Summe der Variablenwerte ist nicht möglich. Es gilt $f_P(n) \leq 2n+1$.

  In Tabelle 1.2 sieht man, dass bei $k=2$ gilt: $f_P(n) < A(2,n)$. Damit ist gezeigt, dass es für Wertzuweisungen ein k gibt, für das $f_P(n) < A(k,n)$ gilt. Dies ist die Induktionsvoraussetzung.

- Falls P die Form $P_1; P_2$ hat, so gibt es nach Induktionsvoraussetzung Konstanten k_1, k_2, so dass $f_{P_1}(n) < A(k_1,n)$ und $f_{P_2}(n) < A(k_2,n)$.

  Sei $k_3 = \max\{k_1, k_2\}$. Nun gilt:

$$\begin{array}{lll} f_P(n) & = & f_{P_2}(f_{P_1}(n)) \\ & < & A(k_2, A(k_1,n)) \\ & \leq & A(k_3, A(k_3,n)) \qquad \text{wegen allgemeiner Monotonie} \\ & < & A(k_3, A(k_3+1,n)) \qquad \text{wegen Monotonie-Eigenschaften B, D} \\ & = & A(k_3+1, n+1) \qquad \text{wegen Definition (3) von } A \\ & \leq & A(k_3+2, n) \qquad \text{wegen Monotonie-Eigenschaft C} \end{array}$$

  D. h., $f_P(n) < A(k,n)$ gilt, wenn man $k = k_3 + 2$ wählt.

[13] Durch eine Hintereinanderschaltung von Wertzuweisungen $x_i := x_j + 1; x_i := x_i + 1; \ldots; x_i := x_i + 1$ kann bspw. eine Wertzuweisung $x_i := x_j + c$ mit $c > 1$ simuliert werden.

- Habe nun P die Form LOOP x_i DO Q END. Wir können ohne Beschränkung der Allgemeinheit annehmen, dass x_i in Q nicht vorkommt[14]. Die LOOP-Schleife wird x_i-mal durchlaufen. Sei $m \leq n$ derjenige Variablenwert für x_i für den $f_P(n)$ maximal wird.
  - Falls $m = 0$, dann wird die LOOP-Schleife gar nicht durchlaufen und alle Variablen behalten ihren ursprünglichen Wert, d. h. die Summe über alle Variablen bleibt unverändert und es gilt $f_P(n) \leq n$. Nach Monotonie-Eigenschaft A gilt $n < A(m,n)$ für alle $m \geq 0$. Somit gilt $f_P(n) < A(k,n)$ bereits, wenn man $k = 0$ wählt.
  - Falls $m = 1$, dann wird die LOOP-Schleife einmal durchlaufen und es gilt, da x_i in Q nicht vorkommt, $f_P(n) = f_Q(n-1) + 1$. Da es nach Induktionsvoraussetzung eine Konstante k_1 gibt, für die $f_Q(n-1) < A(k_1, n-1)$ ist, gilt

    $$\begin{aligned} f_P(n) &< A(k_1, n-1) + 1 && \text{bzw.} \\ f_P(n) &\leq A(k_1, n-1) && \text{wegen der Ganzzahligkeit der Werte} \\ &< A(k_1, n). && \text{wegen Monotonie-Eigenschaft B} \end{aligned}$$

    Wählen wir $k = k_1$, so ist $f_P(n) < A(k,n)$ für alle n erfüllt.
  - Falls $m \geq 2$, dann gibt es mehrere Schleifendurchläufe und es gilt:

    $$\begin{aligned} f_P(n) &= \underbrace{f_Q(f_Q(\ \dots\ f_Q}_{m \text{ mal}}(n-m)\dots) + m \\ &< A(k_1, \underbrace{f_Q(f_Q(\ \dots\ f_Q}_{m-1 \text{ mal}}(n-m)\dots) + m && \text{nach Ind.vorauss. wie oben} \\ f_P(n) &\leq A(k_1, \underbrace{f_Q(f_Q(\ \dots\ f_Q}_{m-1 \text{ mal}}(n-m)\dots) + m - 1 && \text{wg. Ganzzahligkeit wie oben} \\ &\ \ \vdots \\ f_P(n) &\leq \underbrace{A(k_1, (A(k_1, (\ \dots\ A(k_1}_{m \text{ mal}}, n-m)\dots) \\ &< \underbrace{A(k_1, (A(k_1, (\ \dots\ A(k_1}_{m \text{ mal}} + 1, n-m)\dots) && \text{wegen Monotonie-Eig. D} \\ &= \underbrace{A(k_1, (A(k_1, (\ \dots}_{m-2 \text{ mal}}, A(k_1+1, n-m+1)\dots) && \text{wegen Def. (3) von } A \\ &\ \ \vdots \\ &= A(k_1+1, n-1) \\ &< A(k_1+1, n) && \text{wegen Monotonie-Eig. B} \end{aligned}$$

    Also genügt es in diesem Fall $k = k_1 + 1$ zu wählen.

Damit haben wir nachgewiesen, dass es für jedes LOOP-Programm P eine Konstante k gibt, für die $A(k,n)$ stärker wächst als $f_P(n)$. Schließlich kommen wir zum finalen Schritt, nämlich dem Nachweis, dass die ACKERMANN-Funktion nicht LOOP-berechenbar ist.

Angenommen, die ACKERMANN-Funktion wäre LOOP-berechenbar. Dann ist auch die Funktion $B(n) = A(n,n)$ LOOP-berechenbar. Sei P ein LOOP-Programm das B berechnet. P wird am Ende den Ausgabewert $B(n)$ in der Variablen x_0 liefern und weitere Variablen können Werte größer 0 enthalten, sodass gilt $B(n) \leq f_P(n)$.

Zu P wählen wir nun gemäß der vorangegangenen Überlegung eine Konstante k, so dass für alle n gilt $f_P(n) < A(k,n)$. Für den speziellen Fall $n = k$ ergibt sich nun

$$B(k) \leq f_P(k) < A(k,k) = B(k),$$

[14] Per Definition hätte eine Veränderung von x_i in Q keine Auswirkungen auf die Anzahl der Iterationen. Somit könnte man ggf. in Q statt x_i auch eine „frische“ Variable verwenden, der initial der Wert von x_i zugewiesen wird.

was offensichtlich einen Widerspruch darstellt. Daraus folgt, dass die ACKERMANN-Funktion nicht LOOP-berechenbar ist.

Dass die ACKERMANN-Funktion als totale Funktion im intuitiven Sinne berechenbar ist, d. h. dass man einen Algorithmus für ihre Berechnung angeben kann, erscheint plausibel. Das folgende Beispiel zeigt, dass dies auch im formalen Sinne durch Angabe eines WHILE-Programms möglich ist.

Beispiel 1.12 Die „Arbeitsweise“ der ACKERMANN-Funktion wurde in Beispiel 1.11 ausführlich nachgespielt. Man überlegt sich, dass diese „Arbeitsweise“ auch durch folgendes Stapelverfahren abgebildet werden kann:

			0	1				
		1	1	0	2			
	2	1	0	0	0	3		
3	1	0	0	0	0	0	4	
1	0	0	0	0	0	0	0	5

Das Stapelverfahren interpretiert die zwei obersten Operanden des Stapels, m und darüber n, als $A(m, n)$ und manipuliert den Stapel wie folgt:

- Ersetze $0, n$ durch $n + 1$ (Stapel wird niedriger),
- Für $m > 0$ ersetze $m, 0$ durch $m - 1, 1$,
- Für $m, n > 0$ ersetze m, n durch $m - 1, m, n - 1$ (Stapel wird höher).

Dadurch wird $A(m, n)$ stets gemäß der Definitionsgleichungen berechnet.

Ein Stapel (engl. *stack*) ist eine in der Informatik häufig verwendete Datenstruktur, auf der zwei wesentliche Operationen definiert sind: PUSH und POP. PUSH(x) legt einen Wert x oben auf dem Stapel ab, POP(x) entfernt einen Wert vom Stapelspeicher und weist ihn der Variablen x zu.

Mit diesen Operationen kann man ein entsprechendes WHILE-Programm angeben:
(nach Konvention befinden sich m in x_1, n in x_2 und das Ergebnis $A(m, n)$ wird in x_0 abgelegt)

```
INIT(stack);
PUSH(x_1, stack);
PUSH(x_2, stack);
WHILE SIZE(stack) ≠ 1 DO
  POP(x_2, stack);
  POP(x_1, stack);
  IF x_1 = 0 THEN PUSH(x_2 + 1, stack);
    ELSIF x_2 = 0 THEN PUSH(x_1 − 1, stack); PUSH(1, stack)
    ELSE PUSH(x_1 − 1, stack); PUSH(x_1, stack); PUSH(x_2 − 1, stack)
  END;
END;
POP(x_0, stack)
```

Offen ist an dieser Stelle noch, wie die (blau gefärbten) Stack-Operationen durch WHILE-Programme implementiert werden können. Hierzu verwenden wir eine Codierungsfunktion π, die den Stackinhalt auf eine natürliche Zahl *stack* abbildet. *stack* wird dann bspw. in Variable x_3 abgelegt.

INIT(*stack*) wird dann durch $stack := 0$ simuliert und PUSH(x, *stack*) durch $stack := \pi(x, stack)$.

POP und SIZE benötigen die entsprechende **De**codierungsfunktion $\pi^{-1}(y)$, die ein Paar $(\pi_1^{-1}(y), \pi_2^{-1}(y))$ zurückliefert, so dass $\pi(\pi_1^{-1}(y), \pi_2^{-1}(y)) = y$ ist. Damit kann POP(x, *stack*) durch $x := \pi_1^{-1}(stack); stack := \pi_2^{-1}(stack)$ simuliert werden und die Bedingung SIZE(*stack*) $\neq 1$ durch $\pi_2^{-1}(stack) \neq 0$.

Als Codierungsfunktion π kann die CANTOR*sche Paarungsfunktion* verwendet werden. Es ist bekannt, dass diese Funktion und ihre Umkehrfunktion LOOP-berechenbar sind. □

1.3.3 CHURCH-TURING-These

Neben der TURING-Maschine und der WHILE- bzw. GOTO-Programmiersprache wurden noch verschiedene andere Berechenbarkeitsmodelle entwickelt, die hier im einzelnen nicht mehr behandelt werden sollen. Dazu gehören u. a.:

- Registermaschinen (auch *Random Access Machines*) nach J. SHEPHERDSON und H. STURGIS, 1963,
- der λ-Kalkül von A. CHURCH, 1936,
- μ-rekursive Funktionen (deren Unterklasse der *primitiv*-rekursiven Funktionen identisch ist mit den LOOP-berechenbaren Funktionen),
- MARKOV-Algorithmen, 1960.

Für alle diese Modelle wurde inzwischen bewiesen, dass sie dieselbe Berechnungsmächtigkeit wie die TURING-Maschine besitzen.

Aufgrund dieser Äquivalenzen und der Tatsache, dass es bis heute niemandem gelungen ist, einen noch umfassenderen Berechenbarkeitsbegriff zu finden, ist man davon überzeugt, mit den angegebenen Modellen genau *den* Berechenbarkeitsbegriff erfasst zu haben [3]. Diese Überzeugung formuliert man als sog. CHURCH-TURING-These:

> Die Klasse der TURING-berechenbaren Funktionen stimmt mit der Klasse der im intuitiven Sinne berechenbaren Funktionen überein.

Diese These ist nicht formal beweisbar, da der Begriff der „intuitiv“ berechenbaren Funktionen nicht exakt ist. Man versteht darunter alle Funktionen, von denen man sich irgendwie vorstellen kann, dass sie sich algorithmisch lösen lassen.

1.4 Entscheidbarkeit

Eng verknüpft mit der Frage der Berechenbarkeit einer Funktion ist der Begriff der Entscheidbarkeit eines Problems. Ein *Entscheidungsproblem* ist die Frage, ob ein beliebiges Element x aus einer Grundmenge M eine bestimmte Eigenschaft P hat. Die Lösung des Entscheidungsproblems ist eine Antwort „Ja“ oder „Nein“.

Die Menge aller x, die die Eigenschaft P erfüllen, d. h. die die Antwort „Ja“ erzeugen, heißt die *Sprache* des Entscheidungsproblems:

$$L_P = \{x \mid x \in M \text{ und } x \text{ hat die Eigenschaft } P\}.$$

Ein konkretes Entscheidungsproblem könnte die Frage sein, ob die Zahl $n \in \mathbb{N}$ eine Primzahl ist. L_{Primzahl} ist dann die Sprache des Primzahl-Problems bzw. die Menge aller Primzahlen.

Definition Eine Sprache (bzw. Menge) $L_P \subseteq M$ heißt **entscheidbar**, wenn es einen Algorithmus gibt, der zu jedem $x \in M$ nach endlich vielen Schritten die Antwort „Ja“ oder „Nein“ auf die Frage liefert, ob $x \in L_P$ ist.

Eine zweite Version dieser Definition nutzt den Begriff der *charaktistischen Funktion* $\chi : M \to \{0,1\}$ für Teilmengen $T \subseteq M$:

$$\chi_T(x) := \begin{cases} 1 & \text{falls } x \in T \\ 0 & \text{falls } x \notin T \end{cases}$$

L_P heißt entscheidbar genau dann, wenn die charakteristische Funktion $\chi_{L_P} : M \to \{0,1\}$ (TURING-) berechenbar ist.

Auch hier stellt sich natürlich sofort die Frage, ob es unentscheidbare Probleme gibt. Das klassische unentscheidbare Problem ist das **Halteproblem**:

> Es ist nicht entscheidbar, ob eine gegebene TURING-Maschine für eine gegebene Eingabe anhält.

Die Beweistechnik, die hier angewendet wird, beginnt mit der Einführung einer Codierung von TURING-Maschinen und ihren Eingaben in ein einheitliches Alphabet, der Einfachkeit halber $\{0,1\}$.

Das kann geschehen, indem man alle Zustände in Q, alle Bandsymbole in Γ und die Kopfbewegungen in $\{R, L, H\}$ durchnummeriert, d. h. mit einer natürlichen Zahl indiziert. Nun übersetzt man jede Transition $\delta(q_i, a_j) = (a_{j'}, M_k, q_{i'})$ in ein Wort

$$w_{i,j,j',k,i'} = \#\#\text{bin}(i)\#\text{bin}(j)\#\text{bin}(j')\#\text{bin}(k)\#\text{bin}(i')$$

wobei „bin“ die Binärcodierung einer natürlichen Zahl erzeugt. Alle diese Wörter schreibt man in beliebiger Reihenfolge hintereinander und erhält so eine Codierung der zugrundeliegenden TURING-Maschine über dem Alphabet $\{0, 1, \#\}$.

Diese Codierung transformiert man schließlich in ein Wort über $\{0,1\}$, indem man abbildet:

$$\begin{aligned} 0 &\to 00 \\ 1 &\to 01 \\ \# &\to 11 \end{aligned}$$

Wir notieren die entsprechende Codierung einer TM M als $\langle M \rangle$.

Nun stellen wir uns eine unendliche binäre Matrix vor, deren Zeilen und Spalten jeweils über alle Binärzahlen laufen und assoziieren mit den Zeilen mögliche Codes für TURING-Maschinen (nicht alle bilden sinnvolle TURING-Maschinen ab!) und mit den Spalten möglichen Codes für Eingaben.

Angenommen, das Halteproblem wäre entscheidbar, dann könnte in allen Kreuzungspunkten eine 1 stehen, falls die jeweilige TURING-Maschine für die jeweilige Eingabe anhält, sonst eine 0.

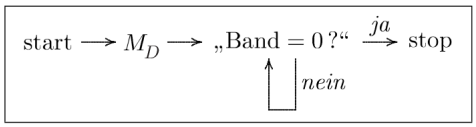
An der Diagonalen dieser Matrix ließe sich ablesen, ob eine TURING-Maschine anhält, wenn sie ihren eigenen Code als Eingabe erhält, oder nicht.

Damit können wir eine Sprache definieren, die genau die 1-Fälle beinhaltet:

$$D = \{\langle M \rangle \mid M \text{ ist eine TM, die bei Eingabe von } \langle M \rangle \text{ anhält}\}$$

Diese Sprache wäre infolge unserer Annahme entscheidbar.

Da also D entscheidbar ist, ist χ_D durch eine TURING-Maschine M_D berechenbar. Diese fiktive Maschine M_D modifizieren wir zu einer Maschine M'_D, die durch folgende Abbildung definiert ist:



Das heißt, M'_D stoppt genau dann, wenn M_D den Wert 0 ausgeben würde. Falls M_D den Wert 1 ausgibt, geht M'_D in eine Endlosschleife.

M'_D starten wir nun mit einer Eingabe $\langle M'_D \rangle$. Zwei Fälle sind zu unterscheiden:

Falls $\langle M'_D \rangle$ mit dieser Eingabe anhält, also $\langle M'_D \rangle \in D$, so muss nach Konstruktion von $\langle M'_D \rangle$ gelten, dass M_D ein Ergebnis 0 auf das Band geschrieben hat. Das wiederum bedeutet, dass $\chi_D(\langle M'_D \rangle) = 0$. Also muss $\langle M'_D \rangle \notin D$ sein, was jedoch einen Widerspruch darstellt. Falls $\langle M'_D \rangle$ mit dieser Eingabe *nicht* anhält, also $\langle M'_D \rangle \notin D$, so muss entsprechend gelten, dass M_D ein Ergebnis 1 auf das Band geschrieben hat. Das wiederum bedeutet, dass $\chi_D(\langle M'_D \rangle) = 1$ und $\langle M'_D \rangle \in D$, was ebenfalls einen Widerspruch darstellt.

Diese Widersprüche zeigen, dass die Annahme falsch war. *Also ist das Halteproblem nicht entscheidbar.*

Bisher haben wir uns vorrangig damit beschäftigt herauszufinden, ob es unlösbare Probleme gibt. Wenn wir uns nun auf die lösbaren Probleme konzentrieren, stellen wir fest, dass die *Komplexität* von Problemen durchaus unterschiedlich sein kann und dass man darüber hinaus dasselbe Problem *effizient* und weniger effizient lösen kann. Damit beschäftigt sich der folgende Abschnitt.

1.5 Komplexität

Die *Komplexität* eines Problems ist ein Maß dafür, wie schwierig es ist, das Problem zu lösen. Sie wird beschrieben durch den Bedarf an Ressourcen, die man zur Lösung des Problems auf einem Rechner benötigt. Bei diesen Ressourcen handelt es sich typischerweise um *Rechenzeit* (gemessen in Berechnungsschritten) und *Speicherplatz* (gemessen in Speicherzellen). Dabei kann die *Zeitkomplexität* niemals kleiner sein als die *Raumkomplexität*, da für das Schreiben einer Speicherzelle jeweils ein Rechenschritt benötigt wird.

Im Allgemeinen wächst der Ressourcenbedarf mit der Größe des Problems: die Sortierung von 1000 Werten ist aufwändiger, als die Sortierung von 10 Werten. Der Primzahltest einer großen Zahl dauert länger als der Primzahltest einer kleinen Zahl.

Um für einen gegebenen Algorithmus ein Maß angeben zu können, das erlaubt, den Ressourcenbedarf abhängig von der Problemgröße abzuschätzen, wird in der Informatik die $\mathcal{O}$-Notation verwendet.

1.5.1 $\mathcal{O}$-Notation

Die $\mathcal{O}$-Notation[15] beschreibt den Wachstumsgrad des Ressourcenbedarfs von Algorithmen (Funktionen) in Abhängigkeit von ihrer Eingabegröße.

Definition Eine Funktion $f(n)$ wächst *mit der Ordnung* $\mathcal{O}(g(n))$, wenn eine (positive) Konstante c existiert, so dass $|f(n)| \leq c \cdot |g(n)|$ für alle $n > n_0$.

[15] Diese Notation wurde erstmals im Jahr 1892 von PAUL BACHMANN benutzt und später von EDMUND LANDAU übernommen und bekannt gemacht. $\mathcal{O}$ wird daher auch als LANDAU-Symbol bezeichnet.

Dabei sind f und g Funktionen $\mathbb{N} \to \mathbb{R}$, $c \in \mathbb{R}^+$ und $n, n_0 \in \mathbb{N}$. Man schreibt $f(n) \in \mathcal{O}(g(n))$.

In der Regel betrachtet man nur monoton wachsende Funktionen: Es soll ja der Ressourcenverbrauch von Algorithmen beschrieben werden. Die Funktion g ist eine *asymptotische Schranke* für f, d. h. für hinreichend große $n > n_0$ ist $f(n)$ bis auf einen konstanten Faktor gleich $g(n)$.

Für $\mathcal{O}$-Terme gelten einige Rechenregeln, die es erlauben, die asymptotische Funktion g möglichst einfach zu halten:

i. $\mathcal{O}(c \cdot g(n)) = \mathcal{O}(g(n))$

ii. $\mathcal{O}(g_1(n) + g_2(n)) = \mathcal{O}(\max\{|g_1(n)|, |g_2(n)|\})$

iii. $\mathcal{O}(g_1(n) \cdot g_2(n)) = \mathcal{O}(g_1(n)) \cdot \mathcal{O}(g_2(n))$

iv. $\mathcal{O}(a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0) = \mathcal{O}(n^k)$, falls $a_k \neq 0$

Exemplarisch sei hier Regel i bewiesen: Eine Funktion $f(n) \in \mathcal{O}(c \cdot g(n))$ muss für große n per Definition die Bedingung $|f(n)| \leq c' \cdot |c \cdot g(n)|$ für ein $c' \in \mathbb{R}^+$ erfüllen. Setzt man $c'' = c' \cdot c$, so gilt $|f(n)| \leq c'' \cdot |g(n)|$ und damit auch $f(n) \in \mathcal{O}(g(n))$.

Die weiteren Rechenregeln werden in ähnlicher Weise bewiesen.

Damit können $\mathcal{O}$-Terme sehr einfach notiert werden. So ist bspw. $4n^3 + 6n + 12 \in \mathcal{O}(n^3)$.

Um zu sehen, wie die $\mathcal{O}$-Notation bei der Beurteilung von Algorithmen eingesetzt werden kann, betrachten wir das Sortierproblem.

Beispiel 1.13 Ein Sortierproblem liegt vor, wenn Elemente einer Menge M, auf der eine (totale[16]) Ordnungsrelation $\leq$ definiert ist, in einer bestimmten Reihenfolge angeordnet werden sollen, bspw. „aufsteigend der Größe nach“.

Für zwei Sortieralgorithmen, *SelectionSort* und *TreeSort*, wird auf den Vorlesungsfolien eine Komplexitätsanalyse gezeigt. □

Die Rechenzeiten für Probleme unterschiedlicher Komplexität werden in Tabelle 1.3 gegenübergestellt. Sie wurden gemessen, indem ein Standardtestprogramm in Iterationen ausgeführt wurde, die dem jeweils angegebenen $\mathcal{O}$-Term entsprechen.

Komplexität	$n = 10$	$n = 20$	$n = 50$	$n = 100$	$n = 10^3$	$n = 10^4$	$n = 10^5$
$\mathcal{O}(\log(n))$							1 ns
$\mathcal{O}(n)$							6 µs
$\mathcal{O}(n \log(n))$						8 µs	0.1 ms
$\mathcal{O}(n^2)$					60 µs	6 ms	0.6 s
$\mathcal{O}(2^n)$	600 µs	0.6 s	18.7 h	hängt	hängt	hängt	hängt
$\mathcal{O}(n!)$	22 ms	111 y	hängt	hängt	hängt	hängt	hängt

ermittelt auf einem 16 700 Dhrystone-MIPS-Rechner (i5-4690 Quadcore, 3.8 GHz)

Tabelle 1.3: Beispiele für Laufzeiten von Algorithmen unterschiedlicher Komplexität

An diesen Laufzeiten erkennt man gut, dass nur Algorithmen mit *polynomialer Komplexität* praktisch berechenbar sind. Es hat sich etabliert, solche Algorithmen als **effiziente Algorithmen** zu

[16] Eine Relation $\leq$ auf einer Menge M heißt *totale Ordnung*, wenn sie reflexiv ($x \leq x$), antisymmetrisch ($x \leq y \wedge x \leq y \Rightarrow x = y$), transitiv ($x \leq y \wedge y \leq z \Rightarrow x \leq z$) und total ($x \leq y \vee y \leq x$) ist.

bezeichnen (obgleich schon die Komplexität n^3 häufig enorme Laufzeiten verursacht). Algorithmen mit (mindestens[17]) *exponentieller* Komplexität heißen nicht-effizient. Exponentielle Algorithmen sind *nicht praktisch* berechenbar.

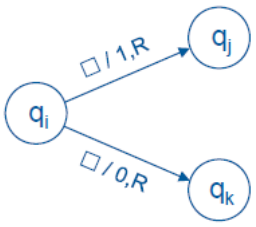
1.5.2 Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$

Die Gleichsetzung von Effizienz mit polynomialer Rechenzeit legt nahe, alle Probleme, die sich in diesem Sinne effizient lösen lassen, in einer Komplexitätsklasse zusammenzufassen.

Definition $\mathcal{P}$ ist die Klasse der Probleme, für die es einen Polynomialzeitalgorithmus gibt.

Um nachzuweisen, dass ein Problem in $\mathcal{P}$ enthalten ist, genügt es, einen Polynomialzeitalgorithmus für das Problem anzugeben. Ein Problem, das mindestens exponentielle Komplexität hat, liegt nicht in $\mathcal{P}$.

Dazwischen gibt es aber noch eine interessante Klasse von Problemen, nämlich solche, die durch einen **nichtdeterministischen** Algorithmus in Polynomialzeit lösbar wären. Was bedeutet „nichtdeterministisch“? Eine nichtdeterministische TURING-Maschine würde statt einer Übergangs*funktion* eine Übergangs*relation* verwenden, d. h. sie würde ermöglichen, dass es für einen Zustandsübergang mehrere Möglichkeiten gibt, z. B.:



Ein nichtdeterministisches LOOP-Programm würde bspw. erlauben:

$$x_i := x_j \mid x_k$$

Wenn es bei einer nichtdeterministischen Berechnung einen Ablauf gibt, der eine Lösung der zugrundeliegenden Funktion liefert (d. h. bei dem bei allen Auswahlmöglichkeiten immer zielsicher die zur Lösung führende Möglichkeit erraten wird) und der eine polynomiale Laufzeit hat, dann spricht man von *nichtdeterministisch polynomialer* Komplexität[18].

Definition $\mathcal{NP}$ ist die Klasse der Probleme, die sich mit *nichtdeterministisch polynomialem* Aufwand lösen lassen.

Offenbar ist $\mathcal{P} \subseteq \mathcal{NP}$, denn deterministische Algorithmen sind ja nur Spezialfälle nichtdeterministischer Algorithmen: die Anzahl der Auswahlmöglichkeiten für den jeweils nächsten Verarbeitungsschritt ist genau 1.

Bislang unbekannt ist, ob $\mathcal{P} \subset \mathcal{NP}$ gilt. Dies ist eine der bekanntesten ungelösten Fragen der Informatik. Es konnte immerhin gezeigt werden, dass es bestimmte Probleme in $\mathcal{NP}$ gibt (also Probleme, für die kein polynomialer Algorithmus bekannt ist), auf die alle anderen Probleme in $\mathcal{NP}$ in Polynomialzeit reduziert werden können. Solche Probleme heißen **$\mathcal{NP}$-vollständig**.

Seit vielen Jahren wird erfolglos versucht, einen polynomialen Algorithmus für ein $\mathcal{NP}$-vollständiges Problem zu finden. Wenn dies gelänge hieße das, dass alle Probleme in $\mathcal{NP}$ in Polynomialzeit lösbar wären. Dass es nicht gelingt wird als ein starkes Indiz (aber natürlich nicht als Beweis) für $\mathcal{P} \neq \mathcal{NP}$ angesehen.

[17] Komplexitäten wie $\mathcal{O}(n!)$ oder $\mathcal{O}(n^n)$ heißen *überexponentiell*.

[18] Eine gleichwertige Betrachtungsweise sagt aus, dass zur Berechnung der Lösung zwar keine Algorithmen mit polynomialer Laufzeit zur Verfügung stehen, Lösungsvorschläge aber in polynomialer Zeit geprüft werden können.

Beispiel 1.14 Sehr viele praktisch relevante Probleme liegen in der Klasse $\mathcal{NP}$. Eine Zusammenstellung bekannter Probleme findet man bspw. in [7].

Eines der bekanntesten $\mathcal{NP}$-vollständigen Probleme ist das Problem des Handlungsreisenden (*Travelling Salesman Problem*): Ein Handlungsreisender soll auf einer Rundreise n vorgegebene Stationen besuchen und schließlich zu seinem Ausgangspunkt zurückkehren. Die Entfernungen (Kosten) zwischen allen Paaren von Stationen sind gegeben. Die Gesamtlänge der Rundreise soll minimal sein.

Alle bisher vorgeschlagenen Algorithmen laufen im Prinzip auf dasselbe Schema hinaus:

- Bilde alle Permutationen der n Stationen.
- Ignoriere diejenigen, die keine Rundreise darstellen.
- Von den verbliebenen Permutationen wähle diejenige mit minimalen Kosten.

Der erste Schritt, das Bilden der Permutationen, liegt in $\mathcal{O}(n!)$. □

Kapitel 2

Rechnerarchitektur

Die Ausführung von Algorithmen erfolgt in der Praxis natürlich nicht auf abstrakten Maschinen wie der TURING-Maschine, sondern auf *realen Rechnern*. Solche Rechner sind Gegenstand des Fachgebiets Rechnerarchitektur.

Ein **Rechner** ist ein *programmgesteuertes Informationsverarbeitungssystem*. Seine Aufgabe ist die Informations*verarbeitung* – das Erfassen, Eingeben, Sortieren, Filtern, Strukturieren, Konvertieren, Manipulieren, Verknüpfen, Speichern, Archivieren, Übertragen, Ausgeben und Löschen von *Information*.

Information ist ein Begriff, der in unterschiedlicher Bedeutung verwendet wird. Umgangssprachlich wird er oft mit *Daten* gleichgesetzt. In der Informatik – der Wissenschaft der Informationsverarbeitung – unterscheidet man beide Begriffe jedoch. Daten sind maschinell verarbeitbare Zeichen (Symbole oder Signale). Erst ihre Interpretation durch den Menschen macht sie zu Information. Oft unterteilt man Information in verschiedene Informationsarten, bspw. Zahlen-, Text-, Bild-, Audio- und Videoinformation.

Neben diesen auf die Interaktion mit dem menschlichen Nutzer ausgerichteten Informationsarten gibt es noch zwei spezielle Informationsarten, die für die Formulierung und Ausführung von Programmen wichtig sind, nämlich Befehle und Adressen. Ein *Befehl* beschreibt eine ausführbare Operation eines (bestimmten) Rechners und eine *Adresse* eine Stelle innerhalb dieses Rechners, an der eine bestimmte Information abgelegt ist. Adressen werden in Befehlen genutzt. Eine Abfolge von Befehlen, die geeignet ist, eine bestimmte Aufgaben- oder Problemstellung (einen Algorithmus) zu bearbeiten, bildet ein **Programm**.

Wir werden im nächsten Abschnitt feststellen, dass all diese unterschiedlichen Informationsarten in Digitalrechnern[1] stets durch *Ziffernfolgen* dargestellt werden.

2.1 Binäre Informationsdarstellung

Ziffernfolgen sind (natürliche) Zahlen. Bereits im letzten Kapitel haben wir bei verschiedenen Betrachtungen beliebige, manchmal sogar strukturierte Symbolfolgen zur Vereinfachung immer in die natürlichen Zahlen abgebildet.

Betrachten wir kurz, wie die medialen Informationsarten – Text, Bild (Grafik, Foto), Video (Animation, Film) und Audio (Sprache, Musik) – datencodiert werden.

[1] Das Adjektiv *digital* bedeutet „in Ziffern dargestellt“ (engl. *digit*: Ziffer, lat. *digitus*: Finger, die Ziffern des Dezimalsystems sind an den Fingern abzählbar).

Text ist eine sequenzielle Folge von Zeichen eines bestimmten Zeichensatzes. Während die Darstellung der Zeichen auf einem Monitor mittels vorgefertigter Muster für die Glyphen erfolgt, werden zur rechnerinternen Darstellung von Buchstaben, Ziffern, Sonder- und Steuerzeichen standardisierte Codes verwendet, die jedes Zeichen auf eine eindeutige *Zahl* abbilden. Da verschiedene Standardcodes existieren, ist es für eine korrekte Interpretation wichtig, den zugrundeliegenden Code zu kennen. Die gebräuchlichsten Codes sind der ASCII (*American Standard Code for Information Interchange*) und der Unicode – ein sehr komplexer Code, der *jedes* jemals verwendete Symbol codiert. Der Unicode ist in mehrere Ebenen untergliedert. Die bei uns gebräuchlichen Schriftzeichen sind in der *Basic Multilingual Plane* eingeordnet. Für diese Ebene wurden verschiedene Zeichencodierungen vereinbart. Eine Zeichencodierung wird auch als *Transformationsformat* bezeichnet (*Unicode Transformation Format*, UTF). Am weitesten verbreitet sind UTF-16 und UTF-8. UTF-16 ist die interne Zeichendarstellung der Betriebssysteme Windows und macOS. UTF-8 wird bei Unix/Linux sowie im Internet verwendet. Beide entsprechen in ihren ersten 128 Zeichen dem ASCII.

Bildinformation wie ein Foto oder ein Piktogramm (engl. *icon*) wird rechnerintern als *Rastergrafik* gespeichert. Eine Rastergrafik ist eine Matrix aus Bildpunkten (engl. *picture elements*, kurz *pixel*), wobei jeder Bildpunkt durch eine Datenstruktur dargestellt wird, die seine Farbe beschreibt. Zur Farbbeschreibung verwendet man typischerweise den RGB-Farbraum, d. h. jede beliebige Farbnuance wird durch Mischung der drei Primärfarben Rot, Grün und Blau in jeweils unterschiedlicher Helligkeit dargestellt. Ein Bildpunkt ist somit ein Tripel $(\mathrm{R}, \mathrm{G}, \mathrm{B})$, das mit drei *Zahlen* den Rotanteil R, den Grünanteil G und den Blauanteil B beschreibt. Sehr häufig wird für die höchste Intensitätsstufe jeweils die Zahl 255 gewählt. Die Matrix aus Zahlentripeln wird oftmals noch in unterschiedlicher Weise komprimiert. GIF, JPEG, PNG sind übliche Datenkompressionsverfahren.

Grafikanwendungen, die selbst Bilder generieren, verwenden zur internen Darstellung *Vektorgrafiken*. So sind geografische Karten, CAD-Zeichnungen oder virtuelle 3 D-Bilder üblicherweise Vektorgrafiken. Eine Vektorgrafik besteht aus einer Liste von Einzelobjekten, aus denen das Bild aufgebaut ist. Jedes Einzelobjekt wird durch eine Vektorfolge und Farbattribute repräsentiert. Die gesamte Information wird häufig als Text in einer Markup-Sprache (XML-Derivat) gehalten. Text wiederum entspricht, wie oben geschildert, interpretierten *Zahlen*.

Bewegtbild- bzw. **Videoinformation** wird durch eine Folge von Bildern dargestellt. Wenn diese Bilder in kurzen Zeitabständen aufeinanderfolgen, entsteht beim menschlichen Betrachter die Illusion einer Bewegung. Hierfür genügen bereits etwa 16 bis 18 Bilder pro Sekunde. Kompressionsverfahren nutzen aus, dass sich direkt aufeinanderfolgende Bilder oft nur wenig voneinander unterscheiden und es somit genügt, pro Bildpunkt nur den Differenzwert zum Vorgängerbild abzuspeichern, der i. allg. gering oder gar Null ist.

Akustische Information wie Sprache oder Musik wird durch eine Schallwelle repräsentiert. Eine solche Welle kann durch Analog/Digital-Wandlung in eine Folge von Zahlen konvertiert werden. Hierbei wird mit einer festgelegten Abtastrate für jeden Abtastzeitpunkt die Amplitude des Signals bestimmt und einer endlichen Anzahl von Quantisierungsstufen zugeordnet. Die identifizierenden Nummern der Quantisierungsstufen bilden in ihrer zeitlichen Folge die digitalen Audiodaten. Die ISDN-Telefonie verwendet bspw. eine Abtastrate von 8000 Hz und eine Auflösung in 256 $(= 2^8)$ Quantisierungsstufen, eine CD eine Abtastrate von 44 100 Hz und eine Auflösung in 65 536 $(= 2^{16})$ Stufen (jeweils vor Datenkompression).

Auf Zahlen-, Befehls- und Adressformate werden wir im weiteren Verlauf dieses Abschnitts noch genauer eingehen.

Bleiben wir zunächst bei Ziffernfolgen (also natürlichen Zahlen). Der Mensch ist gewohnt, mit zehn Ziffern umzugehen, also das *dezimale* Zahlensystem zu verwenden. Früheste Rechner wie bspw. der ENIAC, der UNIVAC oder die ersten IBM-Rechner taten dies ebenso. Bald entschied man sich jedoch für das *duale* Zahlensystem mit nur zwei unterschiedlichen Ziffern. Dafür gab es im wesentlichen drei Gründe:

1. Es ist technisch wesentlich einfacher und störsicherer, Elemente mit nur zwei statt zehn verschiedenen Zuständen zu bauen (man betrachte etwa einen Schalter, der entweder geöffnet

oder geschlossen ist, eine Strom- oder Spannungsstärke, die einen Schwellwert übersteigt oder nicht, eine Magnetisierung, die in einem Ferromagnetikum magnetische Domänen in oder gegen die Bewegungsrichtung eines Schreib-/Lesekopfs ausrichtet).

2. Für die Verarbeitung zweiwertig (*binär*) dargestellter Information sind mathematische Theorien bekannt, die die Entwicklung effizienter Methoden und optimierter Komponenten und Systeme ermöglichen, etwa die BOOLEsche Algebra oder die formale Logik.

3. Rechner verarbeiten in großem Umfang auch nicht-numerische Informationen (Text, Bild, Audio, Video), die in jedem Fall codiert werden müssen. Da spielt es keine Rolle, ob im Dezimal- oder im Dualsystem codiert wird.

Interessanterweise sind aber sowohl das dezimale als auch das duale Zahlensystem Ausprägungen desselben Zahlensystemtyps – des *polyadischen* Zahlensystems[2].

2.1.1 Polyadische Zahlensysteme

Ein polyadisches Zahlensystem mit der Basis B, auch B-adisches Zahlensystem genannt, ist ein Zahlensystem, in dem eine **natürliche** Zahl n in Potenzen von B zerlegt und durch folgende Summe dargestellt wird:

$$n = \sum_{i=0}^{N-1} b_i B^i \tag{2.1}$$

wobei $B \in \mathbb{N},\ B \geq 2$ die *Basis* des Zahlensystems und b_i *Ziffern* genannt werden. N ist die Anzahl Stellen der Zahlendarstellung. In einem Rechner ist N endlich.

$B = 10$ definiert das Dezimalsystem, $B = 2$ das Dualsystem. Andere gebräuchliche Zahlensysteme sind das *Oktalsystem* ($B = 8$) und das *Hexadezimalsystem* ($B = 16$).

Es ist folgende Schreibweise üblich (Ziffernschreibweise):

$$n = (b_{N-1}\, b_{N-2}\, \ldots\, b_1\, b_0)_B \tag{2.2}$$

wobei

- die führenden Ziffern weggelassen werden, wenn sie gleich 0 sind und
- die Kennzeichnung der Basis entfällt, wenn über die Basis kein Zweifel besteht.

Beispiel 2.1 Die Zahl $n = zweihunderteins$ besitzt folgende Darstellungen

- im Dezimalsystem

  $n = 2 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0$
  bzw. $n = (201)_{10}$

- im Dualsystem

  $n = 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
  bzw. $n = (11001001)_2$

- im Oktalsystem

  $n = 3 \cdot 8^2 + 1 \cdot 8^1 + 1 \cdot 8^0$
  bzw. $n = (311)_8$

[2] Polyadische Zahlensysteme werden auch als *Stellenwertsysteme* bezeichnet. Andere (nicht-polyadische) Zahlensysteme sind bspw. das Strichlistensystem oder das römische Zahlensystem.

- im Hexadezimalsystem

  $n = 12 \cdot 16^1 + 9 \cdot 16^0$
  bzw. $n = (C9)_{16}$
  Im Hexadezimalsystem benutzt man für die Zahlen *zehn, elf, ..., fünfzehn* die Symbole (Ziffern) $A, B, \ldots, F$.

□

Rationale Zahlen (Bruchzahlen) werden in einem polyadischen Zahlensystem entsprechend ausgedrückt durch eine Summe:

$$r = \sum_{i=-M}^{N-1} b_i B^i \tag{2.3}$$

bzw. in Ziffernschreibweise:

$$r = (b_{N-1}\, b_{N-2}\, \ldots\, b_1\, b_0\, ,b_{-1}\, \ldots\, b_{-M+1}\, b_{-M})_B. \tag{2.4}$$

2.1.2 Konvertierungsverfahren

Obgleich Rechner intern Dualzahlen verwenden, erfolgt die Ein- und Ausgabe (an der Benutzerschnittstelle) durch besser lesbare Zahlen, also vorrangig Dezimalzahlen, aber auch Hexadezimal- oder gelegentlich Oktalzahlen. Es ist also erforderlich, Zahlen zu einer Basis B in Zahlen zu einer anderen Basis B' umzuwandeln (zu *konvertieren*).

Konvertierung von Dezimal- in Dualzahlen

Natürliche Dezimalzahlen konvertiert man (wie in Beispiel 2.1 bereits angedeutet), indem man zuerst die höchste Zweierpotenz sucht, die kleiner oder gleich dieser Zahl ist. Diese höchste Zweierpotenz wird von der Zahl subtrahiert. Anschließend wird mit dem Rest solange in gleicher Weise weiterverfahren, bis ein Rest 0 erreicht ist. Auf diese Weise ermittelt man alle Zweierpotenzen, die in ihrer Summe die ursprüngliche Zahl ergeben. Transformiert man nun diese Summe in *Polynomschreibweise*, wobei ausschließlich die Koeffizienten 1 und 0 benötigt werden, so ergibt die Aneinanderreihung der Koeffizienten die duale Ziffernfolge.

Beispiel 2.2 Man möchte $(43)_{10}$ in eine Dualzahl konvertieren:

$$\begin{aligned} 43 - 32 &= 11 \\ 11 - 8 &= 3 \\ 3 - 2 &= 1 \\ 1 - 1 &= 0 \end{aligned}$$

d. h.

$$\begin{aligned} (43)_{10} &= 32 + 8 + 2 + 1 \\ &= 2^5 + 2^3 + 2^1 + 2^0 \\ &= 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= (101011)_2. \end{aligned}$$

□

Das gleiche Ergebnis kann auch erzielt werden, indem man die gegebene Zahl durch die Basis 2 dividiert, dadurch einen Quotienten und einen Rest ermittelt, und nun die gleiche Berechnung für den eben ermittelten Quotienten in gleicher Weise solange wiederholt, bis man den Wert 0 erhält[3]. Die gesuchte Dualdarstellung ergibt sich dann durch eine Aneinanderreihung der Reste in *umgekehrter* Ermittlungsreihenfolge.

Beispiel 2.3

$$\begin{aligned} 43:2 &= 21 \text{ Rest } 1 \\ 21:2 &= 10 \text{ Rest } 1 \\ 10:2 &= 5 \text{ Rest } 0 \\ 5:2 &= 2 \text{ Rest } 1 \\ 2:2 &= 1 \text{ Rest } 0 \\ 1:2 &= 0 \text{ Rest } 1 \end{aligned}$$

d. h. $(43)_{10} = (101011)_2$. □

Rationale Zahlen konvertiert man am besten gemäß einer Vorgehensweise, die sich an Beispiel 2.2 anlehnt.

Beispiel 2.4

$$\begin{aligned} 11,375 - 8 &= 3,375 \\ 3,375 - 2 &= 1,375 \\ 1,375 - 1 &= 0,375 \\ 0,375 - 0,25 &= 0,125 \\ 0,125 - 0,125 &= 0 \end{aligned}$$

d. h.

$$\begin{aligned} (11,375)_{10} &= 8 + 2 + 1 + 0,25 + 0,125 \\ &= 2^3 + 2^1 + 2^0 + 2^{-2} + 2^{-3} \\ &= 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &= (1011,011)_2. \end{aligned}$$

□

Alternativ gibt es auch hier eine zweite Vorgehensweise, die allerdings **nur** auf den *gebrochenen Anteil* anwendbar ist.

Beispiel 2.5

$$\begin{aligned} ,375 \cdot 2 &= 0,75 \\ ,75 \cdot 2 &= 1,5 \\ ,5 \cdot 2 &= 1,0 \end{aligned}$$

d. h.

$$(,375)_{10} = (,011)_2.$$

Achtung: beim Ablesen der Ziffernfolge der Nachkommastellen gilt hier die Leserichtung von oben nach unten. □

[3] Diese Vorgehensweise leitet sich ab aus dem HORNER-Schema – einem Umformungsverfahren für Polynome, das die Potenzschreibweise auf die Multiplikation zurückführt. So gilt bspw. $ax^3 + bx^2 + cx + d = ((ax + b)\,x + c)\,x + d$.

Die Konvertierung einer gebrochenen Dezimalzahl mit einer endlichen Anzahl von Nachkommastellen muss nicht notwendigerweise auch in eine Dualzahl mit einer endlichen Anzahl von Nachkommastellen münden. Außerdem ist wegen der i. allg. vorgegebenen, begrenzten Nachkommastellenzahl oft nur eine näherungsweise Darstellung der konvertierten Zahl möglich. Die dadurch entstandene Ungenauigkeit bezeichnet man als *Konvertierungsfehler*.

Konvertierung von Dual- in Dezimalzahlen

Die Methoden für diese Konvertierungsrichtung stellen gewissermaßen die Umkehrung der Methoden aus Abschnitt 2.1.2 dar.

Der erste Ansatz leitet sich wieder unmittelbar aus der Polynomschreibweise einer polyadischen Zahlendarstellung ab: man addiert alle Zweierpotenzen auf, deren Exponenten einem Stellenindex in der Dualzahl entsprechen, an dem eine 1 steht.

Beispiel 2.6 Man möchte $(1001101,01)_2$ in eine Dezimalzahl konvertieren.

Hilfreich kann dabei eine Tabelle sein, die jeder Stelle der Dualzahl ihre Zweierpotenz zuordnet. Man summiert dann alle Zweierpotenzen auf, in deren Spalte eine 1 steht.

1	0	0	1	1	0	1	0	1
2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}
64	32	16	8	4	2	1	0,5	0,25

d. h.

$$\begin{aligned}(1001101,01)_2 &= 64+8+4+1+0,25\\ &= (77,25)_{10}.\end{aligned}$$

□

Alternativ gibt es auch hier eine zweite Methode, die sich wieder aus dem HORNER-Schema ableitet und die den Vorteil hat, dass man keine vorberechneten Zweierpotenzen kennen muss. Allerdings müssen der ganzzahlige Anteil und der gebrochene Anteil einer Zahl in unterschiedlicher Weise behandelt werden.

Bei der Konvertierung des *ganzzahligen* Anteils beginnt man damit, die höchstwertige Dualziffer als vorläufige (dezimale) Gesamtsumme aufzufassen. In jedem weiteren Schritt wird dann die bisherige Gesamtsumme verdoppelt und die nächste Dualziffer addiert.

Beispiel 2.7

$$\begin{aligned}1 &= 1\\ 1\cdot 2+0 &= 2\\ 2\cdot 2+0 &= 4\\ 4\cdot 2+1 &= 9\\ 9\cdot 2+1 &= 19\\ 19\cdot 2+0 &= 38\\ 38\cdot 2+1 &= 77\end{aligned}$$

d. h.

$$(1001101)_2 = (77)_{10}.$$

□

Die Vorgehensweise bei der Konvertierung des *gebrochenen* Anteils verbleibt nach ähnlichem Schema als Übungsaufgabe zu entwickeln.

Weitere Konvertierungen

In speziellen Fällen ist eine besonders einfache Konvertierung möglich, nämlich immer dann, wenn die Basis B des einen Zahlensystems eine Potenz der Basis B' des anderen Zahlensystems ist. Dies ist bspw. gegeben, wenn von Dualzahlen in Oktal- oder Hexadezimalzahlen konvertiert werden soll ($B = 2$ und $B' = 2^3$ bzw. 2^4). Hier kann man direkt Gruppen von 3 bzw. 4 Dualziffern in jeweils eine Oktal- bzw. Hexadezimalziffer umwandeln.

Beispiel 2.8 Die Dualzahl 1010011100 entspricht umgewandelt

$$\begin{aligned}(001\ 010\ 011\ 100)_2 &= (1234)_8\\ (0010\ 1001\ 1100)_2 &= (29\mathrm{C})_{16}\end{aligned}$$

□

2.1.3 Rechenregeln im Dualsystem

Die Grundrechenarten lassen sich mit den aus dem Dezimalsystem bekannten Verfahren (stellenweises Rechnen, Weitergabe von Überträgen usw.) ausführen.

Folgende Tabelle stellt in der ersten Zeile die Rechenregeln zusammen, die in der i-ten Stelle anzuwenden sind, wobei c_{i+1} den Übertrag (*carry*) in die nächsthöhere Stelle beschreibt. In der zweiten Zeile werden diese Regeln auf ein einfaches Beispiel angewendet.

Addition	Subtraktion	Multiplikation	Division
$0+0=0$ $0+1=1$ $1+0=1$ $1+1=0$ und $c_{i+1}=1$	$0-0=0$ $0-1=1$ und $c_{i+1}=-1$ $1-0=1$ $1-1=0$	auf Addition zurückgeführt	auf Subtraktion zurückgeführt
$\begin{array}{rr} & 101\\ + & 011\\ c: & 111\\ \hline & 1000\end{array}$	$\begin{array}{rr} & 101\\ - & 011\\ c: & -1\\ \hline & 010\end{array}$	$\begin{array}{r} 11\cdot 101\\ \hline 11\\ 00\\ 11\\ \hline 1111\end{array}$	$\begin{array}{r\|l} 10001 & :11=101\\ \cline{2-2} -11 & \\ \cline{1-1} 001 & \\ 101 & \\ -11 & \\ \cline{1-1} 010 & \text{(Rest)}\end{array}$

2.1.4 Darstellung negativer Zahlen

Die Darstellung eines negativen Vorzeichens ist in Systemen, die mit nur zwei Symbolen arbeiten, nur über Umwege möglich. So könnte eine bestimmte, reservierte Stelle einer Ziffernfolge als Vorzeichenstelle „missbraucht“ werden.

Es macht Sinn, hierfür die vorderste (höchstwertige) Ziffer zu wählen. So könnte die `0` für ein positives und die `1` für ein negatives Vorzeichen stehen. Um auch für unterschiedlich lange Ziffernfolgen die Position der Vorzeichenstelle eindeutig identifizieren zu können, ist es notwendig, eine feste Länge für alle Ziffernfolgen vorauszusetzen.

Würde nun der Rest der Ziffern einfach als Dualzahl für den Betrag der Zahl interpretiert werden, würde man zwei Nachteile beobachten: Erstens gibt es für die Null nun zwei Darstellungen – eine positive und eine negative Null. Zweitens müsste man bei Rechenoperation die Vorzeichenstelle immer gesondert behandeln, ansonsten würde man bspw. rechnen

$$\begin{array}{rr} 5 & 00101\\ +(\text{-}2) & +10010\\ \hline \neq \text{-}7 & 10111\end{array}$$

Negative Zahlen sollten idealerweise so codiert werden, dass die Anwendung aller Rechenregeln auch ungeachtet des Vorzeichens immer sofort zum richtigen Ergebnis führt.

Beispiel 2.9 Was wäre eine geeignete Binärdarstellung für -10_{10}?

$$\begin{array}{rr} & 0\,0\,0\,0\,1\,0\,1\,0 \\ + & ?\,?\,?\,?\,?\,?\,?\,? \\ \hline = & 0\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

Auf den ersten Blick fällt die Antwort nicht ganz leicht. Leicht wäre es, ein Ergebnis zu erzeugen, in dem jede Stelle 1 ist: hierzu müsste man nur jede Stelle des ersten Operanden invertieren, um den zweiten Operanden zu erzeugen. Wenn man nun noch 1 addiert, entsteht ein Ergebnis, das tatsächlich in allen Stellen bis auf Position N eine 0 aufweist. Diese Position N existiert jedoch gar nicht: Wir haben uns auf Zahlendarstellungen mit N Stellen (Positionen $N-1$ bis 0) festgelegt. Der Übertrag in Position N kann ignoriert werden.

$$\begin{array}{rr} & 0\,0\,0\,0\,1\,0\,1\,0 \\ + & 1\,1\,1\,1\,0\,1\,0\,1 \\ + & \qquad\qquad\qquad 1 \\ \hline = & 1\,0\,0\,0\,0\,0\,0\,0\,0 \end{array}$$

□

Die Darstellung einer negativen Zahl kann also erfolgen, indem man ihren Betrag in eine Dualzahl mit N Stellen codiert, ihn stellenweise invertiert und dann 1 addiert. Diese Darstellungsform negativer ganzer Zahlen heißt **Zweierkomplement**-Darstellung. Die höchstwertige Stelle zeigt an, ob die Zahl positiv (0) oder negativ (1) ist.

Bei einer Zweierkomplement-Darstellung kann man sich die Zahlen auf einem Zahlenring modulo 2^N angeordnet vorstellen. Dabei wird die Zahl Null als positive Zahl aufgefasst, wodurch die Darstellung unsymmetrisch wird (siehe Abbildung 2.1). Der Betrag der kleinsten darstellbaren negativen Zahl beträgt 2^{N-1}, der der größten positiven Zahl $2^{N-1}-1$,

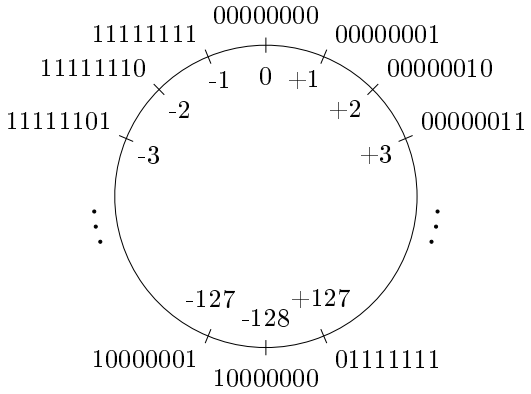


Abbildung 2.1: Zahlenring modulo 2^8 im Zweierkomplement

In der Zahlenringdarstellung ist zu erkennen, dass man eine Summe $a+b$ dadurch erhält, dass man von a ausgehend b Schritte *im Uhrzeigersinn* weiter geht. Eine Differenz $a-b$ erhält man dadurch, dass man von a ausgehend b Schritte *gegen* den Uhrzeigersinn geht.

Das gleiche Subtraktionsergebnis erzielt man jedoch auch, wenn man eine ausreichende Anzahl von Schritten *im* Uhrzeigersinn weiter geht (also die Subtraktion auf eine Addition zurückführt).

Beispiel 2.10 Das Ergebnis der Subtraktion $3 - 2$ ist natürlich 1. Wieviele Schritte müssen wir im Zahlenring in Abb. 2.1 von 3 ausgehend im Uhrzeigersinn gehen, um bei 1 anzukommen? Es sind offenbar 124 (von 3 bis +127) plus 128 (negative Zahlen) plus 2, also insgesamt 254 Schritte. Genau diese (vorzeichenlos interpretierte) Dualzahl bildet den Code für -2.

Tatsächlich scheint es also egal zu sein, ob wir 2 subtrahieren oder das Komplement von 2 addieren. Überzeugen Sie sich selbst, dass dies auch für beliebige andere Subtraktionen gilt! □

Daraus leitet sich das Subtraktionsverfahren im Zweierkomplement ab, das die Subtraktion auf die Addition des Zweierkomplements zurückführt (im Bsp. für $N = 5$):

	$14_{10} - 7_{10}$	$9_{10} - 13_{10}$
Komplementbildung	$7:$ 0 0 1 1 1 $\neg 7:$ 1 1 0 0 0 $+$ 1 $-7_{2C}:$ 1 1 0 0 1	$13:$ 0 1 1 0 1 $\neg 13:$ 1 0 0 1 0 $+$ 1 $-13_{2C}:$ 1 0 0 1 1
Addition des Komplements	$14:$ 0 1 1 1 0 $-7_{2C}:$ $+$ 1 1 0 0 1 1 0 0 1 1 1	$9:$ 0 1 0 0 1 $-13_{2C}:$ $+$ 1 0 0 1 1 1 1 1 0 0

Der durch einen roten Rahmen gekennzeichnete Übertrag in die $N + 1$-te Stelle darf ignoriert werden.

Die **Konvertierung** einer im Zweierkomplement dargestellten negativen Dualzahl in eine Dezimalzahl erfolgt naheliegenderweise durch:

1. die Komplementierung der Zahl (wodurch ihre positive Gegenzahl entsteht),
2. die Anwendung eines Verfahrens aus Abschnitt 2.1.2 zur Konvertierung in eine positive Dezimalzahl,
3. Voranstellen eines Minuszeichens vor die positive Dezimalzahl.

Beispiel 2.11 Sei $N = 8$ und $z = 1110\,0111_2$. Offenbar ist z negativ, da die höchstwertige Stelle 1 ist, d. h. $z = -n_{2C}$ wobei n die positive Gegenzahl zu z ist.

Somit ist $n = 0001\,1000 + 1 = 0001\,1001_2 = 25_{10}$.

Also ist $z = -25_{10}$. □

Eine alternative Methode leitet sich aus der Betrachtung des Zahlenrings ab, der in Abb. 2.1 exemplarisch für $N = 8$ gezeigt wurde.

Dort erkennt man:

- die kleinste darstellbare negative Zahl ist $-2^{N-1} = 1\ 00\ ..\ 00$,
- die nächsthöhere Zahl ist $-2^{N-1} + 1 = 1\ 00\ ..\ 01$.

So geht es weiter

- bis zur größten negativen Zahl $-1 = -2^{N-1} + (2^{N-1} - 1) = 1\ 11\ ..\ 11$.

Offenbar kann man den Dezimalwert einer negativen Binärzahl in Zweierkomplementdarstellung auch erhalten, indem man auf den Dezimalwert der unteren Wertebereichsgrenze denjenigen Dezimalwert addiert, der sich aus der Konvertierung der Binärzahl **ohne** die führende 1 ergibt.

Beispiel 2.12 Die Zahl $z = 1110\,0111_2$ aus Beispiel 2.11 ergibt sich zu:

$$-128 + 64 + 32 + 4 + 2 + 1 = -25.$$

□

2.1.5 Darstellung gebrochener Zahlen

Die Konvertierung rationaler Zahlen haben wir bereits kennengelernt (siehe Gleichungen 2.3 und 2.4). Die Darstellung in Form von Kommazahlen ist allerdings nachteilig, wenn Zahlen in extremen Größenordnungen liegen. In diesen Fällen ist es zweckmäßig, die signifikanten Ziffern der Zahlen getrennt von ihrer Größenordnung zu notieren, d. h. in der sog. („wissenschaftlichen") Exponentialschreibweise oder **Gleitkommazahlendarstellung** (*floating point number representation*). Eine Zahl r wird hier geschrieben als

$$r = (-1)^V \cdot m \cdot B^e \tag{2.5}$$

wobei m die *Mantisse* und e der *Exponent* genannt wird. B ist die Basis des Zahlensystems und V die Vorzeichenziffer ($V = 0$ beschreibt ein positives Vorzeichen, $V = 1$ ein negatives Vorzeichen).

Um eine größtmögliche Genauigkeit darstellen zu können, wird die Mantisse *normiert*, d. h. das Komma wird (bei gleichzeitiger Anpassung des Exponenten) so verschoben, dass genau eine von 0 verschiedene Ziffer vor dem Komma steht. Es gilt somit

$$1 \leq m < B. \tag{2.6}$$

Beispiel 2.13 Im Dezimalsystem wird normiert

$$\begin{aligned} 1234,56 &\rightarrow 1,23456 \cdot 10^3 \\ 0,0815 &\rightarrow 8,15 \cdot 10^{-2} \end{aligned}$$

und im Dualsystem

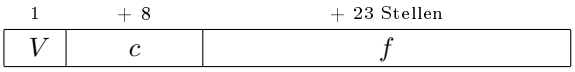
$$\begin{aligned} 1010,101 &\rightarrow 1,010101 \cdot 2^3 \\ 0,01101 &\rightarrow 1,101 \cdot 2^{-2} \end{aligned}$$

□

Im Dualsystem muss die Stelle links vor dem Komma nach Gleichung 2.6 stets eine 1 sein.

Der technische Standard **IEEE 754** für Gleitkommazahlendarstellungen (der heute praktisch von allen Hard- und Softwareherstellern verwendet wird), definiert einheitliche Formate zur Darstellung einer Zahl in Exponentialschreibweise.

Das 32-stellige Grundformat (*single precision*) besitzt folgenden Aufbau:



Das Format verwendet die Betragsdarstellung mit der Vorzeichenstelle V. Die Mantisse m wird im IEEE-754-Format nicht vollständig, sondern nur durch ihren echt gebrochenen Teil f (engl. *fraction*) dargestellt, also nur durch die Stellen rechts vom Komma. Es gilt

$$1, f = m \tag{2.7}$$

Die *redundante* 1 vor dem Komma wird einfach weglassen und sich lediglich gemerkt, dass sie dort eigentlich hingehören würde.

Um beim Exponenten $\pm e$ das Vorzeichen zu vermeiden, wird eine Exzess-$(2^{N-1} - 1)$-Darstellung verwendet. Stehen N Stellen für die Darstellung des Exponenten zur Verfügung, so wählt man den Versatz (engl. *bias*)

$$b = 2^{N-1} - 1 \tag{2.8}$$

und bildet daraus die *Charakteristik* (engl. *biased exponent*)

$$c = e + b \tag{2.9}$$

Typ	Gesamtanzahl Stellen	Stellen c	Stellen f
half precision	16	5	10
single precision	32	8	23
double precision	64	11	52
quadruple precision	128	15	112

Tabelle 2.1: Formate nach Standard IEEE 754-2008

die dann im Format anstelle des Exponenten eingesetzt wird.

Im *single precision*-Format ist $N = 8$ und somit $b = 127$. Es würde also bspw. statt eines Exponenten $e = -19$ die Charakteristik $c = 108$ erscheinen.

Der Standard IEEE 754 kennt noch weitere Formate, die alle gleich strukturiert sind, aber unterschiedliche Feldlängen besitzen. Tabelle 2.1 gibt einen Überblick.

Die Null lässt sich in diesen Formaten eigentlich nicht darstellen, da sie sich nicht normieren lässt (vgl. Forderung 2.6). Per Definition verwendet man daher die Darstellung mit

$$V = 0, \quad f = 0 \quad \text{und} \quad c = 0. \tag{2.10}$$

Der jeweils höchste Zahlenwert der Charakteristik ($c = 11\ldots1_2$) ist dafür reserviert, besondere Werte darzustellen:

- falls $f = 0$ beschreibt dies den Wert $\pm\infty$,
- andernfalls wird der Wert **NaN** (*not a number*) beschrieben, der keine Zahleninterpretation besitzt und z. B. verwendet werden kann, um Variablen als nicht initialisiert zu kennzeichnen oder das Ergebnis nicht zulässiger Ausdrücke (z. B. $x/0$, $\sqrt{-1}$) darzustellen.

Beispiel 2.14 Die reelle Zahl $-11,625 = -1011,101_2 = -1,011101_2 \cdot 2^3$ wird im IEEE 754 *single precision*-Format dargestellt als

1 10000010 01110100000000000000000

Dabei bestimmt sich die Charakteristik zu $c = 3 + 127 = 130_{10} = 10000010_2$. □

Gleitkommaoperationen sind wesentlich aufwändiger als Ganzzahloperationen: Mantisse und Exponent müssen getrennt voneinander behandelt werden. Bei einer Addition/Subtraktion müssen die Operanden so ausgerichtet werden, dass sie den gleichen Exponenten aufweisen. Das Rechenergebnis muss in aller Regel neu normiert werden. Aus diesen Gründen werden Gleitkomma-Rechenoperationen entweder durch Software-Bibliotheksfunktionen realisiert oder (getrennt von der Ganzzahlverarbeitung) durch eine spezielle Gleitkomma-Recheneinheit (*Floating Point Unit*, FPU).

2.1.6 Datentypen

Ein *Datentyp* beschreibt eine Menge von Datenobjekten, die alle die gleiche Struktur haben und mit denen die gleichen Operationen ausgeführt werden können. In höheren Programmiersprachen unterscheidet man bspw. bool, char, int, float, string, array, list, enum, set, struct u. a. In der Rechnerarchitektur beschränkt man sich darauf, dass ein Datentyp lediglich nennt, wieviele Binärstellen zur Speicherung eines Datenobjekts dieses Typs zur Verfügung stehen.

Eine einzelne Stelle kann eine einzige Binärziffer aufnehmen. Sie wird als **Bit** (von engl. *binary digit*) bezeichnet.

Die kleinste *adressierbare* Datenmenge in einem Rechner ist das **Byte**[4]. Ein Byte ist eine Folge von 8 bit. Daher wird manchmal auch synomym der Begriff *Oktett* verwendet.

Die nächstgrößere Datenmenge ist das **Wort**. Das Wort ist eine sehr charakteristische Dateneinheit für einen Rechner. Die meisten internen *Register* – Hardwareelemente, die die unmittelbaren Operanden und Ergebnisse jeder Berechnung zwischenspeichern – sind so dimensioniert, dass sie genau ein Wort aufnehmen können. Auch das *Bussystem* – die Sammelschiene, die Daten zwischen den verschiedenen Rechnereinheiten hin und her transportiert – verwendet als Transporteinheit ein Wort.

Die *Wortbreite* ist je nach Rechner bzw. Prozessor unterschiedlich[5], wobei sie i. d. R. immer ein 2^n-Vielfaches eines Bytes darstellt, also 8, 16, 32 oder 64 bit.

Zur Darstellung etwa von Gleitkommazahlen in den verschiedenen Formaten oder Zahlen beschränkter Größe gibt es abhängige Formate wie das **Halbwort**, das **Doppelwort** und das **Quadwort**. Seltener verwendet wird das **Halbbyte** (auch *Nibble* oder *Tetrade*), das als Folge von 4 bit genau eine Hexadezimalzahl abbildet.

Die Bits einer Folge werden üblicherweise von rechts nach links mit 0 beginnend durchnummeriert. Somit entspricht ihre Positionsnummer ihrer Wertigkeit im dualen Zahlensystem. Bit 0 wird dementsprechend als niederwertigstes Bit (*least significant bit*, LSB), das Bit mit dem jeweils höchsten Index (beim Byte also 7) als höchstwertiges Bit (*most significant bit*, MSB) bezeichnet.

Somit ist es naheliegend, auch die Bytes innerhalb eines Worts wie in Abbildung 2.2 von rechts nach links aufsteigend durchzunummerieren. Man bezeichnet diese Vorgehensweise als *Little Endian*-Reihenfolge. Bytes mit einer niedrigeren Nummer werden auch in einer niedrigeren Adresse gespeichert. Intel-Architekturen nutzen dieses Format. Umgekehrt kann man auch das höchstwertige Byte an die niederwertigste Adresse schreiben. Dies bezeichnet man dann als *Big Endian*-Reihenfolge. SP ARC und IBM z Systems adressieren in dieser Weise. ARM unterstützt beide Systematiken.

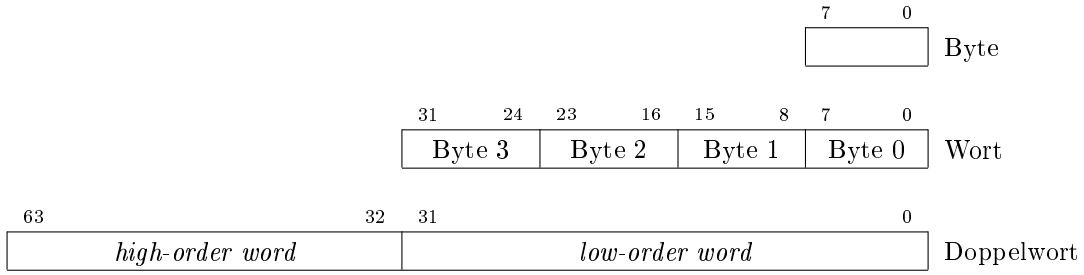


Abbildung 2.2: Grundlegende Datentypen einer 32-bit-Architektur

Um große Mengen von Bits und Bytes zu beschreiben, verwendet man die **SI-Präfixe** aus dem Internationalen Einheitensystem[6], also Kilo (k), Mega (M), Giga (G), Tera (T), Peta (P), Exa (E), usw. Dies sind *Dezimalpräfixe*.

[4] Das Wort Byte wurde 1956 durch den IBM-Rechnerarchitekten Werner Buchholz geprägt, der die englische Bedeutung *bit* = "bisschen" für das Wortspiel missbrauchte, die nächstgrößere Einheit als "Biss" = *bite* zu bezeichnen. Die verballhornte Schreibweise *byte* wurde eingeführt, um versehentliche Verwechslungen mit *bit* zu vermeiden.

[5] In manchen Architekturen, die im Laufe ihrer Evolution die Wortbreite vergrößert haben, wird der Begriff **word** aus Kompatibilitätsgründen mit fixer Bedeutung verwendet. So begann Intels x86-Architektur mit einer Wortbreite von 16 bit und verwendet seitdem den Datentyp `WORD` für Bitfolgen von 16 bit. Für die Wortbreite von 32 bit wurde der Typ `DWORD` eingeführt, für 64 bit `QWORD`. Aus den gleichen Gründen bezeichnet der Datentyp `word` in den heutigen 64 bit-Architekturen PowerPC und UltraSPARC immernoch eine Bitfolge von 32 bit. Eine Folge von 64 bit heißt `double word`.

[6] frz. *Système International d'unités*, abgekürzt SI

Datenmengen treten typischerweise in Vielfachen von Zweierpotenzen auf. Lange wurden die SI-Präfixe deswegen näherungsweise auch in der Bedeutung von *Binärpräfixen* benutzt. Um diese Mehrdeutigkeit zu vermeiden, die sich zudem bei steigenden Potenzen in wachsenden prozentualen Ungenauigkeiten auswirkt, hat die IEC[7] im Jahre 1998 offizielle Binärpräfixe eingeführt (siehe Tabelle 2.2). Die SI-Präfixe wurden um den Zusatz „bi“ für *binary* bzw. ein „i“ im Symbol ergänzt. Die Akzeptanz dieser IEC-Präfixe ist bis heute jedoch gering.

SI-Präfix	Zehnerpotenz	„nahe“ Zweierpotenz		**IEC-Präfix**
k (Kilo)	10^3	1 024	2^{10}	Ki (Kibi)
M (Mega)	10^6	1 048 576	2^{20}	Mi (Mebi)
G (Giga)	10^9	1 073 741 824	2^{30}	Gi (Gibi)
T (Tera)	10^{12}	1 099 511 627 776	2^{40}	Ti (Tebi)
P (Peta)	10^{15}	1 125 899 906 842 624	2^{50}	Pi (Pebi)
E (Exa)	10^{18}	1 152 921 504 606 846 976	2^{60}	Ei (Exbi)

Tabelle 2.2: SI- und IEC-Präfixe

2.1.7 Befehlssatz und Maschinensprache

Die Befehle, die ein Rechner ausführen kann, sogenannte *Maschinenbefehle*, sind viel elementarer und weniger komfortabel als die Befehle und Kontrollstrukturen, die höhere Programmiersprachen zur Verfügung stellen. Programme, die in einer höheren Programmiersprache verfasst werden, sind deshalb auch nicht unmittelbar ausführbar, sondern müssen zunächst übersetzt (kompiliert oder interpretiert) werden. Das Ergebnis der Übersetzung ist ein *Maschinenprogramm*.

Natürlich muss auch ein Maschinenbefehl einer vorgegebenen Syntax genügen, die als *Maschinensprache* bezeichnet wird. Die Menge aller Maschinenbefehle eines Rechners bezeichnet man als den *Befehlssatz* (engl. *instruction set*) dieses Rechners.

Der Umfang des Befehlssatzes variiert je nach betrachtetem Rechner tlw. erheblich. Unterschieden werden die beiden Gegensätze

- CISC (*Complex Instruction Set Computer*) und
- RISC (*Reduced Instruction Set Computer*).

CISC-Rechner stellen dem Maschinenprogrammierer relativ große Befehlssätze zur Verfügung, die zum Teil recht komplexe Rechenoperationen ausführen können. Dies ermöglicht i. allg. kompaktere Maschinenprogramme, hat für die rechnerinterne Organisation aber den Nachteil, dass die Ausführungszeiten verschiedener Maschinenbefehle sehr unterschiedlich ausfallen können. RISC-Rechner streben möglichst kleine Befehlssätze mit sehr einfachen Befehlen an. Der Programmierkomfort muss durch Programmbibliotheken und den Compiler bzw. Interpreter geschaffen werden.

Unter den gängigen Rechnerarchitekturen gelten die Intel-Architektur (x86, IA-32, Intel-64) und IBM z Systems als CISC-Architekturen, während Architekturen wie ARM, PowerPC, UltraSPARC und MIPS allesamt RISC-Architekturen verkörpern.

Allgemein lassen sich die Befehle eines Befehlssatzes in verschiedene *Befehlsgruppen* untergliedern. Man unterscheidet typischerweise:

- arithmetische Befehle (Addition, Subtraktion, Multiplikation, Division, ...),

[7] *International Electrotechnical Commission*

- logische Befehle (Und-, Oder-Verknüpfung, Negation, ...),
- Transportbefehle (Laden bzw. Verschieben von Werten in Register oder in den Speicher),
- Schiebe- und Rotationsbefehle (bezogen auf Registerinhalte),
- Befehle zur Programmablaufsteuerung (Test- und Vergleichsbefehle, Sprungbefehle, Unterprogrammaufruf und -rücksprung, ...),
- Systembefehle (Ein-/Ausgabebefehle für Peripheriegeräte, Befehle die den Zustand des Rechners in besonderer Weise verändern wie HLT und SYSCALL)[8].

Befehle in Maschinensprache bestehen aus Binärzahlen, die als Bytes im Speicher abgelegt werden. Jeder Befehl wird durch einen individuellen numerischen Code (seinen Operations-Code oder kurz **Opcode**) identifiziert. Aus dem Opcode, der immer am Anfang des Befehls steht, lässt sich die Funktion des Befehls (arithmetische oder logische Verknüpfung, Transportfunktion, ...), die Art, in der die ggf. benötigten Operanden angegeben werden, und implizit auch die Länge des Befehls ablesen.

Wir wollen an dieser Stelle als Beispiel für eine sehr einfache Maschinensprache die Sprache verwenden, die Glenn Brookshear in seinem Lehrbuch [8] einführt. Das Format der Befehle orientiert sich an typischen RISC-Architekturen.

Diese Sprache basiert auf einem Rechnermodell, das in Abbildung 2.3 wiedergegeben ist. Der Registersatz umfasst 16 Rechenregister, die mit 0 beginnend durchnummeriert sind. Alle Rechenregister haben eine Breite von 1 Byte. Es gibt zwei Spezialregister: Der *Befehlszähler* PC enthält jeweils die Speicheradresse desjenigen Befehls, der als nächstes aus dem Speicher geholt werden soll. Auch dieses Register hat eine Breite von 1 Byte. Das *Befehlsregister* IR nimmt den jeweils nächsten auszuführenden Befehl auf. Alle Befehle haben eine einheitliche Länge[9] von jeweils 2 Byte. Aus diesem Grund ist auch das Befehlsregister 2 Byte breit[10]. Der Speicher bietet Raum für $2^8 = 256$ Bytes. Jedes Byte ist einzeln adressierbar.

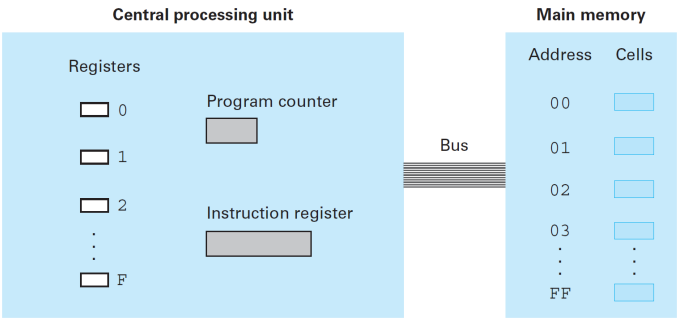


Abbildung 2.3: Brookshears Rechnermodell (aus [8])

Der Befehlssatz besteht aus den zwölf in Tabelle 2.3 angegebenen Maschinenbefehlen. In jedem Befehl wird durch die ersten 4 bit der Opcode dargestellt, die folgenden 12 bit beschreiben die Operanden. Abkürzend werden Opcode und Operanden durch Hexziffern notiert.

[8] Systembefehle sind *privilegierte* Befehle, d. h. ihre Ausführung ist nur dem Betriebssystem und nicht einem Anwendungsprogramm gestattet. Hintergrund ist, dass ein konfliktfreier Zugriff auf exklusiv nutzbare Ressourcen am einfachsten gewährleistet werden kann, wenn die Verwaltung durch eine zentrale Instanz – das Betriebssystem – erfolgt. Anwendungsprogramme können Betriebssystemfunktionen über einen SYSCALL aufrufen.

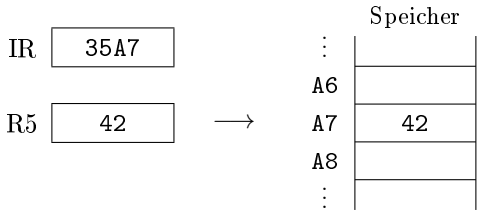
[9] Die einheitliche Befehlslänge ist typisch für RISC-Rechner. In CISC-Rechnern haben Befehle durchaus unterschiedliche Längen.

[10] Im Allgemeinen hat ein Befehlsregister dieselbe Breite wie alle anderen Register auch. Brookshears Rechnermodell ist somit in dieser Hinsicht untypisch, schafft aber ein einfacheres Verständnis der Abläufe. Zum Holen eines Befehls aus dem Speicher müssen uniform stets zwei Speicherzugriffe und Bustransfers hintereinander durchgeführt werden.

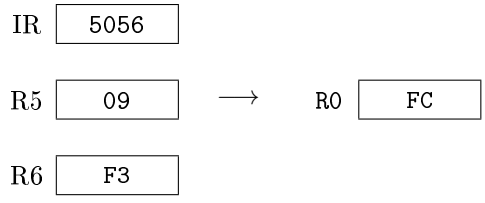
Opcode	Operanden	Beschreibung
1	RXY	Lade Speicherwort aus Adresse XY in Register R
2	RXY	Lade Wert XY in Register R
3	RXY	Speichere Inhalt von Register R in Speicheradresse XY
4	0RS	Verschiebe Inhalt von Register R in Register S
5	RST	Addiere Inhalte der Register S und T und lege Ergebnis in R ab (alle Werte im Zweierkomplement)
6	RST	Addiere Inhalte der Register S und T und lege Ergebnis in R ab (alle Werte im an IEEE 754 angelehnten Gleitkommaformat 1+3+4 bit)
7	RST	OR-verknüpfe Inhalte der Register S und T und lege Ergebnis in R ab
8	RST	AND-verknüpfe Inhalte der Register S und T und lege Ergebnis in R ab
9	RST	XOR-verknüpfe Inhalte der Register S und T und lege Ergebnis in R ab
A	R0X	Rotiere den Inhalt von Register R um X Stellen nach rechts
B	RXY	Springe zum Befehl in Speicheradresse XY wenn der Inhalt von Register R gleich dem Inhalt von Register R0 ist
C	000	Halte die weitere Befehlsausführung an

Tabelle 2.3: Befehlssatz der Maschinensprache nach [8]

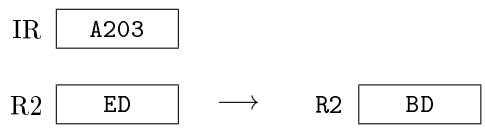
Beispiel 2.15 Der Befehl `35A7` bewirkt, dass der Rechner den Inhalt von Register 5 (bspw. den Wert `42`) in Speicheradresse `A7` speichert.



Mit Ausführung des Befehls `5056` werden die als Zweierkomplementdarstellungen interpretierten Inhalte der Register 5 und 6 addiert und das Ergebnis in Register 0 abgelegt. Unter der Annahme, dass R5 die Dezimalzahl 9 und R6 die Dezimalzahl -13 (= $\texttt{F3}_{16}$) enthält, wird somit in R0 das Ergebnis $-4_{10} = \texttt{FC}_{16}$ geschrieben.



Durch den Befehl `A203` wird eine Rotation des Binärmusters in Register 2 um 3 Stellen nach rechts ausgelöst. Wurde R2 bspw. auf den Wert $\texttt{1110 1101}_2 = \texttt{ED}_{16}$ gesetzt, so ändert sich der Wert zu $\texttt{1011 1101}_2 = \texttt{BD}_{16}$.



□

Es ist sehr mühsam, direkt mit hexadezimal codierten Maschinenbefehlen zu programmieren. Üblicherweise benutzen Maschinenprogrammierer daher eine direkt auf die Maschinensprache abbildbare symbolische Sprache, die **Assemblersprache**. Ein einfaches Übersetzungsprogramm, der *Assembler*, leistet die Abbildung. Für jeden Maschinenbefehl gibt es einen entsprechenden Assemblerbefehl, der gewissermaßen die *mnemonische* Version[11] des Maschinenbefehls darstellt.

Für die Maschinenbefehle aus Tabelle 2.3 bietet es sich bspw. an, folgende Assemblerbefehle zu verwenden:

Opcode	Operanden	Assemblernotation
1	RXY	`LOAD` R, XY
2	RXY	`LOADI` R, XY
3	RXY	`STORE` XY, R
4	0RS	`MOVE` S, R
5	RST	`ADD` R, S, T
6	RST	`ADD-FLOAT` R, S, T
7	RST	`OR` R, S ,T
8	RST	`AND` R, S, T
9	RST	`XOR` R, S, T
A	R0X	`ROTATE-RIGHT` R, X
B	RXY	`JUMP` XY, R
C	000	`HALT`

Tabelle 2.4: Assemblerbefehle

Dabei folgt die Syntax der Assemblerbefehle der üblichen Konvention, dass der jeweils erste Operand der sog. *Zieloperand* (engl. *destination*) ist, d. h. derjenige Operand, der angibt, wo das Ergebnis der Operation abzulegen ist.

Durch eine Folge von Assemblerbefehlen entsteht ein „lesbares Maschinenprogramm“ bzw. ein **Assemblerprogramm**. Bei der Interpretation eines Assemblerprogramms ist zu beachten, dass jede Zeile genau einem Maschinenbefehl im Speicher entspricht und somit jede Zeile mit einer Speicheradresse assoziiert werden kann, die den Ort des Befehls im Speicher beschreibt. Zur Verdeutlichung wird diese Speicheradresse vor jeden Assemblerbefehl notiert, gefolgt von einem Doppelpunkt als Trennzeichen. Die meisten Assemblersprachen erlauben hier auch *symbolische* Speicheradressen zu verwenden. Die Befehle werden in genau der Reihenfolge ausgeführt, in der sie im Programmtext erscheinen. Einzige Ausnahme ist der *Sprungbefehl*, der den sequentiellen Kontrollfluss auf einen anderen als den nächsten Befehl umlenkt. *Kommentare* werden üblicherweise mit einem Semikolon eingeleitet und enden mit dem Zeilenende. Sie werden von einem Assembler nicht in Maschinencode übersetzt.

Beispiel 2.16 Als Beispiel für ein einfaches Assemblerprogramm soll der Wert der arithmetischen Reihe berechnet werden, die alle natürlichen Zahlen von 1 bis n aufsummiert[12], also

$$\sum_{i=1}^{n} i.$$

Aus Sicht eines Programms wird der Speicher in zwei[13] Segmente aufgeteilt: das Code- und das Datensegment. Im *Codesegment* liegt nach dem Laden des Programms der gesamte auszuführende Programmcode in Maschinensprache. Das *Datensegment* enthält alle Daten, die während des Programmlaufs benötigt werden.

[11] Merkhilfe, von griechisch *mnéme* für Gedächtnis, Erinnerung

[12] Carl Friedrich Gauss entdeckte als neunjähriger Schüler, dass dieser Wert auch durch die Formel $\frac{n\cdot(n+1)}{2}$ berechnet werden kann.

[13] Oft existiert noch ein drittes Segment, das *Stacksegment*, das den Stack unterbringt. Der *Stack* ist ein LIFO-Speicher (Last-In-First-Out), der Daten aufnimmt, die dynamisch zur Laufzeit eines Programms entstehen. In der einfachen Architektur, die in dieser Vorlesung behandelt wird, wollen wir auf einen Stack verzichten.

Es sei angenommen, dass das Codesegment bei Adresse 0 und das Datensegment bei Adresse 14 beginnt. Der Wert n wurde in einem vorhergehenden Arbeitsschritt bereits in das erste Byte des Datensegments geladen. Das Ergebnis soll am Ende ins zweite Byte des Datensegments gespeichert werden.

Das Programm könnte wie folgt aussehen (dabei sei angenommen, dass die Entwurfsentscheidung getroffen wurde, die Summe herabzählend und mit n beginnend zu berechnen):

```
     ; Initialisierung
00:  LOADI 0,00     ; Konstantwert 0 für Laufschleifenende
02:  LOAD  1,14     ; Register 1 = Laufvariable i, mit n initialisiert
04:  LOADI 2,00     ; Register 2 = Zwischensumme, mit 0 initialisiert
06:  LOADI 3,FF     ; Konstantwert -1 zum Herabzählen der Laufvariablen

     ; Laufschleife
08:  JUMP  10,1     ; wenn Register 1 null ist: Laufschleife verlassen
0A:  ADD   2,2,1    ; i zur Zwischensumme hinzuaddieren
0C:  ADD   1,1,3    ; i um 1 herunterzählen (-1 addieren)
0E:  JUMP  08,0     ; unbedingt zum Anfang der Laufschleife springen

     ; Abschluss
10:  STORE 15,2     ; Ergebnis speichern
12:  HALT           ; Ende
```

Das entsprechende, bspw. von einem Assembler erzeugte Maschinenprogramm wird wie folgt im Speicher erscheinen:

0	20
1	00
2	11
3	14
4	22
5	00
6	23
7	FF
8	B1
9	10
A	52
B	21
C	51
D	13
E	B0
F	08
10	32
11	15
12	C0
13	00
14	n
15	Ergebnis

□

In den nächsten Abschnitten wollen wir betrachten, wie Rechner aufgebaut sein müssen, damit sie solche Maschinenprogramme abarbeiten und dabei binäre Daten verarbeiten können. Beginnen wollen wir damit, einige Grundschaltungen kennenzulernen, aus denen letztlich alle Teilsysteme eines Rechners bestehen.

2.2 Digitale Schaltungen

Die digitale Schaltungstechnik beruht auf Methoden, die sich aus den Rechenregeln der *Schaltalgebra*[14] ableiten. Die Schaltalgebra ist über zwei Werten, 0 und 1, definiert. Sei

$$\mathbb{B} = \{0, 1\} \tag{2.11}$$

die Menge, die genau diese beiden Konstanten (*Schaltkonstanten*) enthält. Es existieren die Operatoren AND ($\wedge$, $\cdot$), OR ($\vee$, $+$) und NOT ($\neg$, $\overline{\ }$). Für die *Schaltvariablen* $a, b, c \in \mathbb{B}$ gelten folgende Axiome:

Kommutativgesetze	$a \cdot b = b \cdot a$	$a + b = b + a$
Distributivgesetze	$a \cdot (b + c) = a \cdot b + a \cdot c$	$a + b \cdot c = (a + b) \cdot (a + c)$
Identitätselemente	$a \cdot 1 = a$	$a + 0 = a$
Komplementierung	$a \cdot \overline{a} = 0$	$a + \overline{a} = 1$

und die daraus abgeleiteten Theoreme:

Assoziativgesetze	$(a \cdot b) \cdot c = a \cdot (b \cdot c)$	$(a + b) + c = a + (b + c)$
Idempotenzgesetze	$a \cdot a = a$	$a + a = a$
Dominanzgesetze	$a \cdot 0 = 0$	$a + 1 = 1$
Absorptionsgesetze	$a \cdot (a + b) = a$	$a + a \cdot b = a$
DE MORGAN-*Gesetze*	$\overline{a \cdot b} = \overline{a} + \overline{b}$	$\overline{a + b} = \overline{a} \cdot \overline{b}$
Involutionsgesetz	$\overline{\overline{a}} = a$	

Eine *Schaltfunktion* $f(x_1, x_2, \ldots x_n)$ ist eine Zuordnungsvorschrift, die jeder der 2^n möglichen Wertekombinationen der Schaltvariablen $x_1, x_2, \ldots x_n \in \mathbb{B}$ einen eindeutigen Wert $f(x_1, x_2, \ldots x_n) \in \mathbb{B}$ zuordnet.

Mit diesen Begriffen lassen sich folgende Grundelemente digitaler Systeme identifizieren:

- *Verknüpfungsglieder* realisieren logische Operatoren,
- *Schaltnetze* realisieren Schaltfunktionen,
- *Speicherglieder* dienen der Speicherung von Schaltvariablen,
- *Schaltwerke* entsprechen Schaltnetzen mit Speichergliedern.

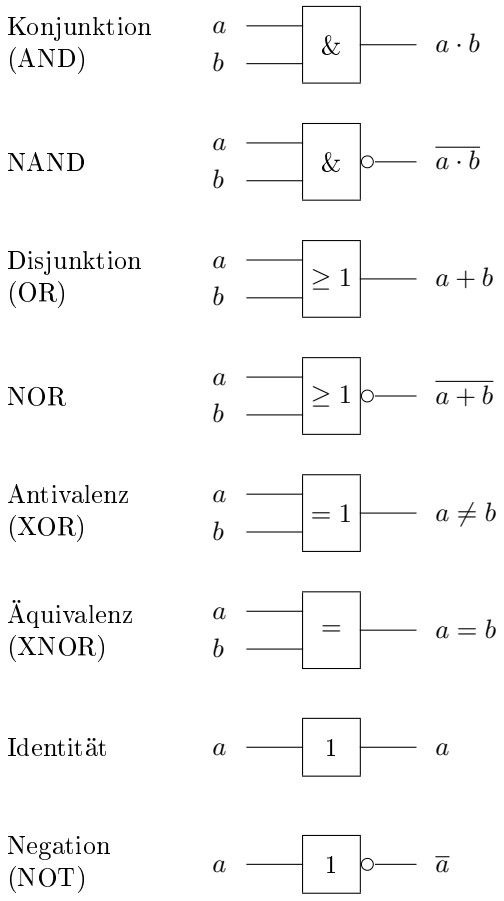
2.2.1 Verknüpfungsglieder

Ein Verknüpfungsglied (engl. *gate*) ist eine elektronische Schaltung, die eine Verknüpfung mehrerer Eingangsvariablen zu genau einer Ausgangsvariablen bewirkt.

Verknüpfungsglieder realisieren üblicherweise nur einfache Verknüpfungen einer oder zweier Schaltvariablen. Die Grundverknüpfungen sind AND, OR und NOT. Durch geeignete Zusammenschaltung dieser drei Verknüpfungsglieder kann jede beliebige andere Verknüpfung realisiert werden.

[14] Die *Schaltalgebra* geht auf CLAUDE E. SHANNON (1916-2001) zurück, der sich wiederum auf eine algebraische Struktur stützte, die GEORGE BOOLE (1815-1864) als Modell zur mathematischen Behandlung der Logik entwickelte und die heute als *Boolesche Algebra* bezeichnet wird. SHANNON benutzte die Boolesche Algebra 1938 erstmals, um prinzipielle Eigenschaften von elektrischen Serien- und Parallelschaltungen zu untersuchen. Heute wird zwischen Schaltalgebra und Boolescher Algebra nur noch selten begrifflich unterschieden, da beide mathematisch gleichwertig sind und sich die Unterschiede lediglich in der Terminologie widerspiegeln.

In Schaltungen werden für die wichtigsten Verknüpfungsglieder folgende Schaltsymbole (nach Norm IEC 60617) verwendet:



Ebenfalls üblich sind Verknüpfungsglieder mit mehr als zwei Eingängen, die entweder alle mit dem gleichen Operator verknüpft werden (siehe bspw. Abbildung 2.4a), oder sog. *Komplexgatter*, die komplexere logische Funktionen realisieren (siehe bspw. Abbildung 2.4b).

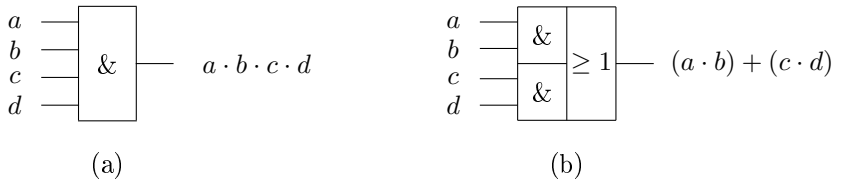


Abbildung 2.4: Verknüpfungsglieder mit mehr als zwei Eingängen

Die technische Realisierung von Verknüpfungsgliedern erfolgt durch Transistorschaltungen.

Ein **Transistor** (abgeleitet aus dem engl. *Transfer Resistor*[15]) ist ein elektrisch steuerbarer Widerstand, den es in unterschiedlichen Bauformen gibt. Die in heutigen hochintegrierten Schaltungen meistgenutzte Bauform ist der **Feldeffekttransistor** (FET). Seine Funktionsweise sei mit Hilfe von Abbildung 2.5 kurz erläutert.

[15] siehe `http://www.nobelprize.org/educational/physics/transistor/function/firsttransistor.html`

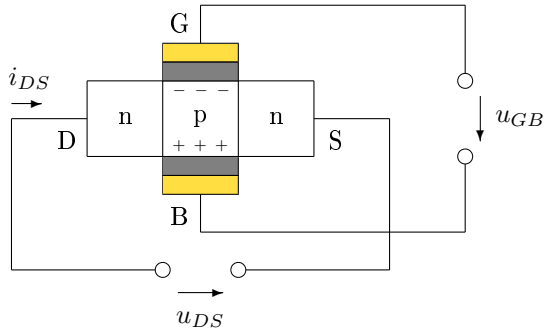


Abbildung 2.5: Feldeffekttransistorschaltung

Zentral für die Funktionsweise ist die Zonenfolge unterschiedlich dotierten Halbleitermaterials. Im Bild wechseln sich eine n-Zone, eine p-Zone und wieder ein n-Zone ab. Eine *n-Zone* besteht aus n-dotiertem Halbleitermaterial, d. h. einem Material, in dessen Kristallstruktur durch gezielten Einbau von Fremdatomen ein Elektronenüberschuss erzeugt wurde. Umgekehrt besteht eine *p-Zone* aus p-dotiertem Halbleitermaterial, in dem in entsprechender Weise ein Überschuss von „Defektelektronen“ („Löchern“), also ein Elektronenmangel erzeugt wurde.

Sowohl Elektronen als auch Löcher bewegen sich aufgrund thermischer Energiezufuhr. Somit dringen sie über die Grenzfläche hinweg auch eine kurze Strecke in das jeweils umgekehrt dotierte Gebiet ein. Dabei „rekombinieren“ Elektronen und Löcher und neutralisieren sich gegenseitig. So entsteht in einem schmalen Bereich links und rechts der Grenzflächen eine ladungsträgerfreie Zone, die sich wie ein Isolator verhält, die *Sperrzone*. Ein Stromfluss ($i_{DS} > 0$) längs der npn-Strecke ist nicht möglich.

Durch ein genügend großes elektrisches Feld, das auf die mittlere p-Zone wirkt, kann allerdings ein interessanter Effekt erzeugt werden: es bildet sich an der zum Pluspol gerichteten Seite ein (schmaler) Inversionskanal (Elektronenüberschuss), der die Sperrzonen überwindet und die npn-Strecke leitend werden lässt.

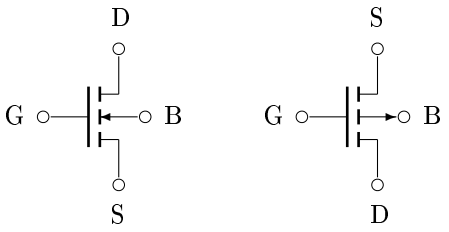
Bezeichnet man die vier Anschlüsse als *Gate* (G), *Bulk* (B), *Drain* (D) und *Source* (S)[16], so wird die Strecke DS leitend, sobald $u_{GB} \approx 1$ V gilt.

Die hier dargestellte Bauform eines FET wird auch als **nMOS**-FET bezeichnet. Diese Abkürzung beschreibt die Zonenfolge *npn* und die charakteristische Materialschichtung, durch die der Feldeffekt erzeugt wird: die Gate-Elektrode wird aus **M**etall (i. d. R. Aluminium) erzeugt und durch eine Isolationsschicht aus **O**xid (SiO_2) vom (p-dotierten) Halbleitermaterial (engl. **S**emiconductor) getrennt.

Das Schaltsymbol eines (selbstsperrenden) nMOS-Transistors ist in Abbildung 2.6 (a) gezeigt. Seine Funktion entspricht einem Schalter, der schließt, sobald $u_{GB} \geq 1$ V wird.

Die komplementäre Bauform heißt **pMOS**-FET. Ein solcher Transistor entsteht durch die Zonenfolge *pnp* und bildet einen p-Inversionskanal aus, sobald $u_{GB} \approx -2$ V gilt. Längs der Schaltstrecke wird der Anschluss mit dem höheren Potential nun als *Source* bezeichnet. Das entsprechende Schaltsymbol zeigt Abbildung 2.6 (b). Die Funktion dieses Transistors entspricht einem Schalter, der schließt, sobald $u_{GB} \leq -2$ V wird.

[16] Von den beiden an und für sich symmetrischen Anschlüssen Drain und Source wird im Kontext einer Schaltung derjenige als Drain bezeichnet, der am *höheren* Potential liegt.



(a) nMOS (b) pMOS

Abbildung 2.6: Schaltsymbole für Feldeffekttransistoren

Transistoren erlauben, Verknüpfungsglieder physikalisch zu realisieren.

Das Schaltbild eines **Inverters** in CMOS-Technik[17] ist in Abbildung 2.7 gezeigt.

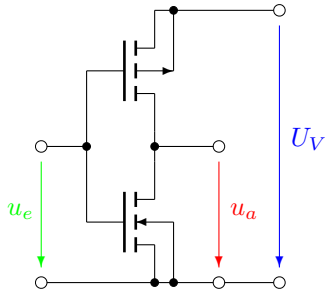


Abbildung 2.7: Inverterschaltung

Für eine Betriebsspannung $U_V = 5$ V und eine Eingangsspannung $u_e = 0$ V sperrt der nMOS-Transistor. Der pMOS-Transistor hingegegen leitet, da sein $u_G - u_B = u_e - U_V = -5$ V genügend negativ ist. Somit wird die Versorgungsspannung zur Ausgangsspannung durchgeschaltet, d. h. $u_a = U_V = 5$ V.

Falls die Eingangsspannung jedoch einen Wert $u_e = 5$ V annimmt, leitet nun der nMOS-Transistor und der pMOS-Transistor sperrt. Dies zieht den Wert der Ausgangsspannung auf das Bezugspotential (Masse, GND), also $u_a = 0$ V.

Die Verknüpfungsglieder **NAND** und **NOR** sind Verallgemeinerungen dieser Inverterschaltung und in den Abbildungen 2.8 und 2.9 gezeigt (in beiden Abbildungen verstehen sich alle Spannungsangaben jeweils relativ zum Bezugspotential $0\,V$).

Durch Hintereinanderschalten von einem NAND und einem Inverter bzw. einem NOR und einem Inverter bildet man AND- und OR-Verknüpfungsglieder. Für XOR- und XNOR-Verknüpfungsglieder gibt es CMOS-Implementierungen, die mit nur 4 Transistoren auskommen [9].

Neben der logischen Verknüpfung von Signalzuständen gibt es noch eine weitere Art der Verknüpfung: den Anschluss von Signalleitungen an Sammelleitungen zur Datenübertragung (*Bus*leitungen). Verknüpfungsglieder, die dies leisten, heißen *Bustreiber*.

[17] *Complementary* MOS ist eine Technik, die sowohl nMOS- als auch pMOS-Transistoren auf einem gemeinsamen Substrat verwendet. CMOS-Schaltkreise realisieren ihre Logikfunktion durch die Verschaltung von nMOS/pMOS-Transistorpaaren, bei denen in jedem Schaltzustand genau einer der beiden Transistoren sperrt und der andere leitet. Dadurch zeichnet sich die CMOS-Technik durch eine besonders geringe Leistungsaufnahme aus und stellt deswegen auch die meistgenutzte Technik beim Entwurf integrierter Schaltungen dar.

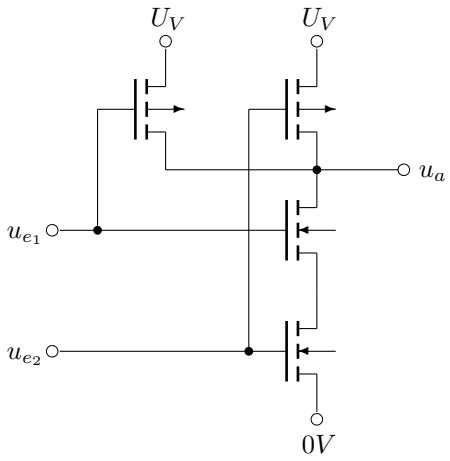


Abbildung 2.8: NAND-Schaltung

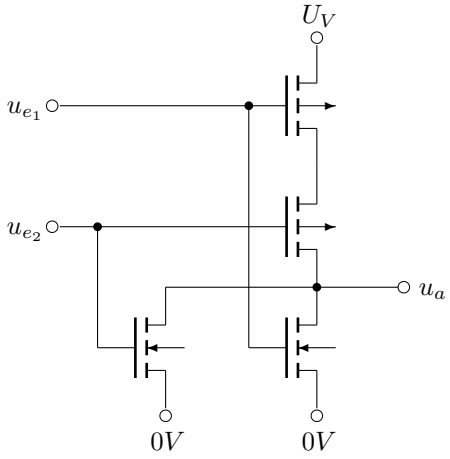


Abbildung 2.9: NOR-Schaltung

Man unterscheidet zwei verschiedene Bauformen für Treiberschaltungen:

- *Wired-or*- bzw. *Open-Drain*-Treiber und
- *Tristate*-Treiber.

Beim **Wired-Or**- bzw. **Open-Drain**-Prinzip führt die Busleitung im „Ruhezustand“ den hohen Pegel. Ein Teilnehmer, der einen bestimmten Umstand anzeigen möchte, versetzt den Transistor in der Treiberstufe in den leitenden Zustand und sorgt so dafür, dass die Speisespannung am Arbeitswiderstand abfällt und der niedrige Pegel sich auf der Busleitung durchsetzt (siehe Abbildung 2.10). Man spricht hier von einer *low-aktiven* Signalisierung.

Wired-or-Treiber werden bevorzugt für Steuerleitungen verwendet, die von mehreren Teilnehmern aktiviert werden können.

Beim **Tristate**-Prinzip liegt die Busleitung, solange kein Teilnehmer aktiv ist, weder auf `0` noch auf `1`, sondern auf einem „dritten“ Wert Z, der als *hochohmig* bezeichnet wird. Z beschreibt, dass zu keinem der beiden Versorgungspotentiale `1` (hoher Pegel) oder `0` (niedriger Pegel) eine elektrische Verbindung besteht.

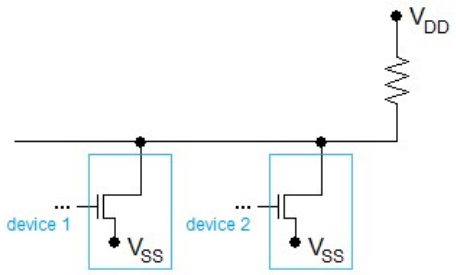


Abbildung 2.10: *Wired-or*-Busankopplung

Erst dadurch, dass der Transistor in der Treiberstufe in den leitenden Zustand versetzt wird, kann ein elektrischer Pegel `0` oder `1` auf die Busleitung durchgeschaltet werden. Es muss gewährleistet sein, dass höchstens ein einziger Busteilnehmer auf den Bus schreibt. Abbildung 2.11 zeigt die Busankopplung.

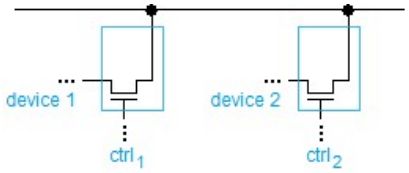


Abbildung 2.11: *Tristate*-Busankopplung

Ein Tristate-Treiberbaustein ist im Wesentlichen ein Signalverstärker, besitzt jedoch einen zusätzlichen Steuereingang C (*control*). Solange an diesem C-Eingang der Wert 0 anliegt, liegt der Ausgang auf Z.

2.2.2 Schaltnetze

Werden (logische) Verknüpfungsglieder zusammengeschaltet, so spricht man von *Schaltnetzen* oder *kombinatorischen Schaltungen* (engl. *combinational circuits*). Ziel ist die Realisierung von Schaltfunktionen.

Bei der Zusammenschaltung sind folgende (topologischen) Richtlinien zu beachten:

- Die Eingänge der Verknüpfungsglieder dürfen nur mit Ausgängen voranliegender Verknüpfungsglieder oder mit Schaltnetzeingängen verbunden sein oder müssen mit Schaltkonstanten beschaltet werden (Signalrückführungen sind nicht zulässig).
- Die Ausgänge von Verknüpfungsgliedern dürfen nicht miteinander verbunden werden.

Um für eine gegebene, beliebige Schaltfunktion **systematisch** zu einer Schaltnetzrealisierung zu kommen, bedient man sich sog. *Normalformen*, d. h. man stellt die Schaltfunktion durch normierte Ausdrücke dar.

Normalformen

Sei $n \geq 1$ und $f : \mathbb{B}^n \to \mathbb{B}$ eine beliebige n-stellige Schaltfunktion. Dann kann f dargestellt werden durch eine Funktionstafel mit 2^n Zeilen, wobei die Argumente so angeordnet stehen, dass in der i-ten Zeile ($0 \leq i < 2^n$) gerade die Dualzahldarstellung von i steht. i heißt ein *Index* zu f.

Beispiel 2.17 Eine dreistellige Schaltfunktion $f : \mathbb{B}^3 \to \mathbb{B}$ sei durch folgende Funktionstafel oder *Wertetabelle* beschrieben:

i	x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

□

Sei $i_1 \dots i_n$ die Ziffernfolge der n-stelligen Dualdarstellung von Index i.

i heißt *einschlägiger* Index zu f, falls $f(i_1, \dots, i_n) = 1$ ist.

In Beispiel 2.17 sind 3, 5 und 7 die einschlägigen Indizes zu f.

Die Funktion $m_i : \mathcal{B}^n \to \mathcal{B}$, definiert durch

$$m_i(x_1, \dots, x_n) = x_1^{i_1} \cdot x_2^{i_2} \cdot \dots \cdot x_n^{i_n}$$

heißt *i-ter Minterm* von f. Dabei ist

$$x_j^{i_j} = \begin{cases} x_j & \text{falls } i_j = 1 \\ \overline{x}_j & \text{falls } i_j = 0 \end{cases}$$

Der Einfachkeit halber wird die Argumentliste eines Minterms oft weggelassen.

In Beispiel 2.17 ist also z. B. $m_3 = \overline{x}_1 \cdot x_2 \cdot x_3$ und $m_4 = x_1 \cdot \overline{x}_2 \cdot \overline{x}_3$.

Wesentliche Eigenschaft eines Minterms m_i ist, dass er genau dann zu 1 evaluiert, wenn seine Argumentliste den Wert der Dualdarstellung von i annimmt, also $x_1 = i_1, \dots, x_n = i_n$. Hieraus folgt ein Satz, der auch als *Hauptsatz der Schaltalgebra*[18] bezeichnet wird:

> Jede Schaltfunktion $f : \mathbb{B}^n \to \mathbb{B}$ ist eindeutig darstellbar als Disjunktion der Minterme ihrer einschlägigen Indizes, d. h. ist $I \subseteq \{0, \dots, 2^n - 1\}$ die Menge der einschlägigen Indizes von f, so gilt
>
> $$f = \sum_{i \in I} m_i$$

Dieser Ausdruck heißt auch *kanonische disjunktive Normalform* KDNF.[19]

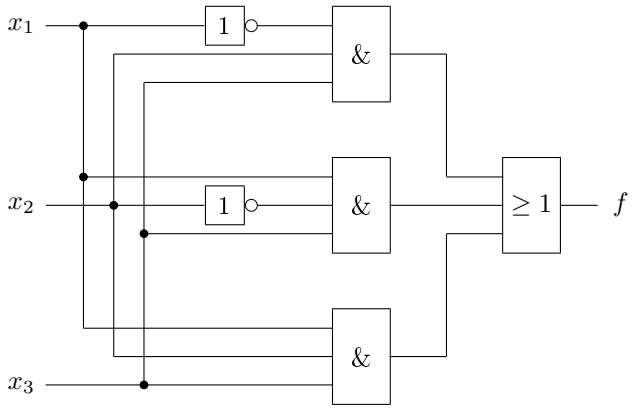
Für Beispiel 2.17 gilt also

$$\begin{aligned} f(x_1, x_2, x_3) &= m_3 + m_5 + m_7 \\ &= \overline{x}_1 \cdot x_2 \cdot x_3 + x_1 \cdot \overline{x}_2 \cdot x_3 + x_1 \cdot x_2 \cdot x_3 \end{aligned}$$

[18] Eine Folgerung aus diesem Satz ist, dass jede beliebige Schaltfunktion mittels der zweistelligen Operationen $+$, $\cdot$ sowie der Negation darstellbar ist.

[19] kanonisch (latein. *canon*, Norm, Regel) bedeutet „dem Kanon entsprechend“, eine Darstellung mit vorgegebenen Eigenschaften

Ein realisierendes Schaltnetz könnte nun aussehen wie folgt:



Der damit verbundene Hardwareaufwand von zwei Invertern und vier AND/OR-Gliedern mit je drei Eingängen lässt sich jedoch (erheblich) reduzieren, wenn man einige Gesetze der Schaltalgebra anwendet, um die Darstellung von f zu vereinfachen:

$$\begin{aligned} f(x_1, x_2, x_3) &= \overline{x}_1 \cdot x_2 \cdot x_3 + x_1 \cdot \overline{x}_2 \cdot x_3 + x_1 \cdot x_2 \cdot x_3 \\ &= (\overline{x}_1 + x_1) \cdot x_2 \cdot x_3 + x_1 \cdot \overline{x}_2 \cdot x_3 \\ &= x_2 \cdot x_3 + x_1 \cdot \overline{x}_2 \cdot x_3 \end{aligned}$$

Diese Realisierung kommt mit nur noch einem Inverter, je einem 2-Eingänge-AND und -OR, sowie einem 3-Eingänge-AND aus, was bereits nahezu einer Halbierung des Hardwareaufwands entspricht.

Es ist somit beim Schaltnetzentwurf von herausragender Wichtigkeit, einen mittels KDNF gewonnenen ersten Entwurf so weit wie möglich zu vereinfachen, d. h. zu *minimieren*. Die vereinfachte (minimierte) Darstellung liegt i. allg. in *nicht-kanonischer* disjunktiver Normalform (DNF) vor. Eine nicht-kanonische Form entsteht durch Kürzung von Literalen[20] aus den Monomen[21].

Minimierung

Das Vereinfachungsverfahren, das im letzten Beispiel angewendet wurde, heißt *Resolutionsregel* und kann wie folgt beschrieben werden:

> Kommen in einer disjunktiven Normalform zwei Monome vor, welche sich in *genau einem* Literal komplementär unterscheiden, so können diese beiden Monome durch ihren gemeinsamen Teil ersetzt werden.

Aufgrund des Idempotenzgesetzes, welches die Verdopplung von Monomen erlaubt, kann die Resolutionsregel ggf. auch mehrmals pro Term angewendet werden. So könnte die Funktion f aus Beispiel 2.17 sogar noch weiter vereinfacht werden:

$$\begin{aligned} f(x_1, x_2, x_3) &= \overline{x}_1 \cdot x_2 \cdot x_3 + x_1 \cdot \overline{x}_2 \cdot x_3 + x_1 \cdot x_2 \cdot x_3 \\ &= \overline{x}_1 \cdot x_2 \cdot x_3 + x_1 \cdot x_2 \cdot x_3 + x_1 \cdot \overline{x}_2 \cdot x_3 + x_1 \cdot x_2 \cdot x_3 \\ &= (\overline{x}_1 + x_1) \cdot x_2 \cdot x_3 + x_1 \cdot (\overline{x}_2 + x_2) \cdot x_3 \\ &= x_2 \cdot x_3 + x_1 \cdot x_3 \end{aligned}$$

[20] Ein *Literal* ist eine Schaltvariable oder die Negation einer Schaltvariablen.
[21] Ein *Monom* ist ein Term, der durch die konjunktive Verknüpfung von Literalen gebildet wird.

Auf diesem Prinzip basiert das Minimierungsverfahren von KARNAUGH – ein visuelles Verfahren, das mit einer Umordnung der grundlegenden Wahrheitstafel in eine 0/1-Matrix beginnt. Diese Matrix heißt KARNAUGH-*Diagramm*[22] und bietet eine gute Übersicht über alle möglichen Resolutionen zu einer gegebenen Schaltfunktion $f : \mathbb{B}^n \to \mathbb{B}$, sofern $n \leq 4$.

Bei einer geraden Anzahl n von Eingangsvariablen verwendet man die Hälfte der Variablen zur Benennung der Spalten, die andere Hälfte zur Benennung der Zeilen. Bei einer ungeraden Anzahl steht für die Zeilenbenennung eine Variable weniger zur Verfügung.

Jeder einzelnen Spalte bzw. Zeile wird eine der möglichen Wertekombinationen der zur Benennung verwendeten Variablen zugeordnet. Dabei ist darauf zu achten, dass sich die Wertekombinationen benachbarter Spalten bzw. Zeilen jeweils nur in *genau einem* Wert unterscheiden. In die Matrixfelder selbst werden die Ausgangswerte der Funktion f eingetragen, die zu den Eingangswertekombinationen gehören, die durch die Spalten- und Zeilenbenennungen vorgegeben sind.

Beispiel 2.18 Das KARNAUGH-Diagramm zur Funktionstafel von Beispiel 2.17 sieht folgendermaßen aus:

x_3 \ x_1, x_2	00	01	11	10
0	0	0	0	0
1	0	1	1	1

Da es ausreicht, in der Matrix nur die Einsen tatsächlich einzutragen, wird auf die Nullen oft verzichtet. □

Jedem Minterm eines einschlägigen Indexes von f entspricht genau eine Eins im KARNAUGH-Diagramm und umgekehrt. Folglich entsprechen zwei benachbarte Einsen zwei Mintermen, welche sich in genau einer komplementären Variablen unterscheiden und auf die somit die Resolutionsregel angewendet werden kann. Zwei solche Einsen bilden einen *Zweierblock*. Diese Beobachtung lässt sich verallgemeinern auf Vierer-, Achter- und Sechzehnerblöcke.

Dies wiederum bedeutet für das Auffinden einer minimierten Darstellung von f:

> Man fasse alle im Diagramm auftretenden Einsen in möglichst große Blöcke der Größe $2^r \times 2^s$ $(r, s \in \{0, 1, 2\})$ zusammen, wähle von diesen soviele aus, dass jede Eins in mindestens einem Block vorkommt, und bilde die Disjunktion der diesen Blöcken entsprechenden Monome.

Für Beispiel 2.18 ergeben sich folgende Blöcke:

x_3 \ x_1, x_2	00	01	11	10
0				
1		1	1	1

[22] nach MAURICE KARNAUGH, geb. 1924, einem US-amerikanischen Physiker, der diese Darstellung 1953 auf der Grundlage einer Arbeit von EDWARD W. VEITCH aus dem Jahr 1952 entwickelte; diese Diagramme werden daher auch häufig als KARNAUGH-VEITCH-Diagramme, kurz **KV-Diagramme**, bezeichnet

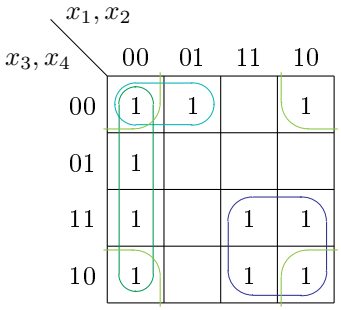
Jedem Block entspricht ein Monom. Man erhält dieses, indem alle im Block konstant vorkommenden Literale konjunktiv verknüpft werden.

Im grün umrandeten Block beobachten wir einheitlich $x_2 = 1$ und $x_3 = 1$. Das entsprechende Monom lautet somit $x_2 \cdot x_3$. Im blau umrandeten Block gilt jeweils $x_1 = 1$ und $x_3 = 1$, d. h. hier wird das Monom $x_1 \cdot x_3$ zugeordnet.

Würde eine Schaltvariable x_i in einem Block konstant immer den Wert $x_i = 0$ annehmen, würde das Literal $\overline{x}_i$ im Monom notiert werden.

Die Bildung von Blöcken darf auch *periodisch* über die Matrixränder hinaus erfolgen, da aufgrund der speziellen Beschriftung die obere und die untere Zeile bzw. die rechte und die linke Spalte der Matrix als benachbart angesehen werden dürfen.

Beispiel 2.19 Gegeben sei folgendes KARNAUGH-Diagramm für eine vierstellige Funktion f:



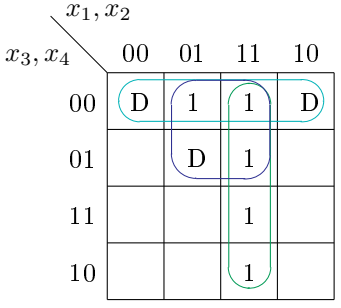
Man erhält in disjunktiver Normalform das Ergebnis

$$f = \overline{x}_1 \cdot \overline{x}_3 \cdot \overline{x}_4 + \overline{x}_1 \cdot \overline{x}_2 + x_1 \cdot x_3 + \overline{x}_2 \cdot \overline{x}_4$$

□

Bisher haben wir stillschweigend nur *vollständig* spezifizierte Funktionen zugrunde gelegt, d. h. Funktionen, bei denen für jede denkbare Eingabekombination immer ein spezifizierter Ausgabewert angegeben werden kann. Dies muss aber nicht in jedem Anwendungsfall zutreffen. So kann es bspw. vorkommen, dass aufgrund äußerer Randbedingungen bestimmte Eingangskombinationen in einer Schaltung gar nicht möglich sind oder der Ausgabewert in manchen Fällen überhaupt nicht relevant ist. Man spricht hier von einer *partiellen* Schaltfunktion und verwendet in der Funktionstabelle bzw. im KARNAUGH-Diagramm für die unspezifizierten Werte die **Don't care**-Belegung oder kurz den Eintrag „D". Bei der Bildung maximal großer Blöcke darf ein D wahlweise als 0 oder als 1 interpretiert werden – je nachdem, wie es besser passt.

Beispiel 2.20 Eine partielle vierstellige Funktion f sei durch folgendes KARNAUGH-Diagramm beschrieben:



Abhängig davon, wie man die *Don't care*-Einträge interpretiert, genügt neben dem grün umrandeten Block genau einer der beiden blau umrandeten Blöcke, um alle Einsen mit maximal großen Blocken zu überdecken. Somit sind zwei gleich gute Lösungen möglich, nämlich entweder

$$f = x_1 \cdot x_2 + \overline{x}_3 \cdot \overline{x}_4$$

oder

$$f = x_1 \cdot x_2 + x_2 \cdot \overline{x}_3.$$

□

Das KARNAUGH-Verfahren ist erweiterbar auf 5- und 6-stellige Funktionen, indem es in eine dreidimensionale Darstellung (einen *Quader* bzw. *Würfel*) übertragen wird. Für höherstellige Funktionen wird die geometrische Darstellung der Nachbarschaftsbeziehungen zu kompliziert.

In diesen Fällen werden tabellarische Verfahren eingesetzt, die auf Rechnern algorithmisch lösbar sind. Ein bekanntes (jedoch nicht effizientes[23]) Verfahren ist die Methode von QUINE und MCCLUSKEY, die ebenfalls auf der Resolutionsmethode basiert, aber in dieser Vorlesung nicht weiter behandelt werden soll.

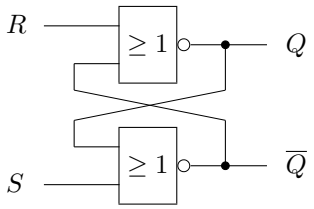
2.2.3 Speicherglieder

Ein *Speicherglied* ist ein Grundbaustein, der den Wert einer Schaltvariablen aufnimmt, aufbewahrt und abgibt. Die technische Realisierung heißt *bistabile Kippschaltung* oder *Flipflop*.

Flipflops

Ein Flipflop ist eine elektronische Schaltung, die zwei stabile Zustände besitzt und die in dem gerade eingenommenen stabilen Zustand solange verharrt, bis sie durch einen von außen kommenden Anstoß in den anderen stabilen Zustand umgeschaltet wird.

Die Grundschaltung eines Flipflops besteht aus zwei NOR-Verknüpfungsgliedern, die über Kreuz rückgekoppelt sind.



Es besitzt die komplementären Ausgänge Q, $\overline{Q}$ und die beiden Eingänge R und S. Werden die Eingänge belegt mit

$$S = 1 \quad \text{und} \quad R = 0,$$

so stellt sich die Ausgangsbelegung

$$\overline{Q} = 0 \quad \text{und} \quad Q = 1$$

ein. Entsprechend führt die Eingangsbelegung $S = 0$ und $R = 1$ zur Ausgangsbelegung $Q = 0$ und $\overline{Q} = 1$. Setzt man $S = R = 0$, so bleibt die bisherige Ausgabe erhalten.

[23] Die Laufzeit des Verfahrens liegt in $\mathcal{O}(n^2\, 3^n \log(n))$.

Interpretiert man Q als den *Zustand* des Flipflops, wird er durch $S = 1$ und $R = 0$ gesetzt und durch $S = 0$ und $R = 1$ rückgesetzt. Daher heißen S und R auch *Set* und *Reset*. Das Flipflop wird als *RS-Flipflop* bezeichnet.

Für $S = R = 1$ gehen beide Ausgänge gleichzeitig auf 0. Wechseln anschließend auch beide Eingänge S und R gleichzeitig auf 0, ist der Ausgangszustand nicht mehr definiert: bei exakt gleich dimensionierten NOR-Verknüpfungsgliedern beginnt das Flipflop zu oszillieren, andernfalls stellt sich ein zwar stabiler, aber nicht vorhersagbarer Zustand ein. Deshalb ist dieser Eingangszustand nicht zulässig.

Die Schaltzustände lassen sich in folgender Tabelle zusammenfassen:

S	R	Q
0	0	wie vorher
0	1	0
1	0	1
1	1	nicht zulässig

Da Umschaltvorgänge Zeit benötigen, liegen Eingabesignale nicht zu allen Zeitpunkten gleichzeitig und stabil an. Oft ist es daher gewünscht, dass ein Flipflop nur zu bestimmten Zeiten auf die Eingabesignale reagiert. Dieses Verhalten kann durch die Verwendung eines *Taktsignals* C realisiert werden, das die Steuereingänge des Flipflops nur zu bestimmten Zeiten freischaltet. Abbildung 2.12 zeigt die Realisierung eines solchen *taktgesteuerten* RS-Flipflops.

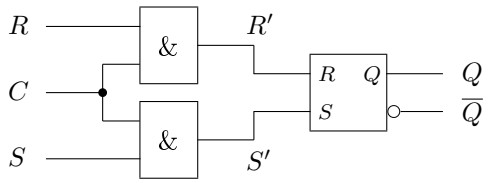


Abbildung 2.12: getaktetes RS-Flipflop

Für $C = 0$ ist $R' = S' = 0$ und das Flipflop speichert den alten Zustand. Für $C = 1$ wird $R' = R$ und $S' = S$. Das Flipflop verhält sich wie ein „normales“ RS-Flipflop.

Wir haben nun gesehen, wie der Zustand eines Flipflops gesetzt bzw. rückgesetzt werden kann. Um den Wert D einer Schaltvariablen zu speichern genügt es,

$$S = D \quad \text{und} \quad R = \overline{D}$$

zu setzen.

Bei der so entstehenden Speicherzelle (*D-Flipflop*, siehe Abbildung 2.13) wird $Q = D$, solange der Takt $C = 1$ ist. In der Phase, in der $C = 0$ ist, bleibt der zuletzt bestehende Ausgangszustand gespeichert.

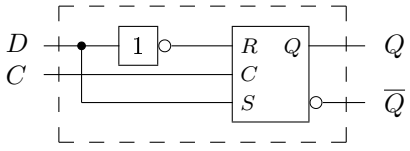


Abbildung 2.13: D-Flipflop

Flipflops, die während der gesamten Zeit, in der $C = 1$ ist, auf den Eingangszustand reagieren, heißen *transparent* oder (takt-) **pegelgesteuert** (engl. *level-triggered*)[24].

Diese Eigenschaft ist für manche Anwendungen unerwünscht. Man benötigt stattdessen Flipflops, bei denen die Eingangsdaten zu genau einem Zeitpunkt übernommen werden, typischerweise bei der steigenden oder bei der fallenden Flanke des Taktsignals.

Für ein Flipflop mit solch einem Verhalten kann man zwei pegelgesteuerte D-Flipflops verwenden, die durch den Takt C komplementär zueinander verriegelt werden. Abbildung 2.14 zeigt diese Anordnung.

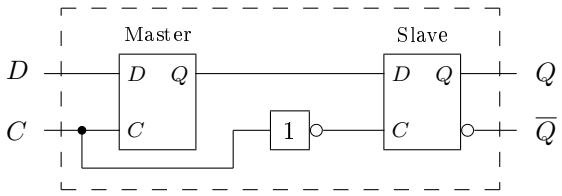


Abbildung 2.14: Master-Slave-D-Flipflop

Solange $C = 1$ ist, schaltet das erste Flipflop (*Master*) den Eingang D transparent auf seinen Ausgang Q_M durch. Da das zweite Flipflop (*Slave*) während dieser Zeit jedoch verriegelt ist, seinen Zustand also unverändert hält, bleibt der Ausgang $Q = Q_S$ auf seinem konstanten (alten) Wert.

Mit dem Wechsel auf $C = 0$ riegelt nun das Master-Flipflop ab, d. h. es speichert genau denjenigen Wert, der zum Zeitpunkt der *fallenden* Flanke des C-Signals am Eingang D anlag. Dieser Wert wird jetzt auch vom Slave übernommen, der sich in dieser Taktphase nun seinerseits transparent verhält, d. h. jede Eingangsänderung durchschaltet. Weitere Eingangsänderungen werden allerdings nicht mehr auftreten, da der Master ja verriegelt ist.

Ein solches Flipflop heißt *Master-Slave-D-Flipflop*. Es verhält sich nach außen als **flankengesteuertes** Flipflop (engl. *edge-triggered*). Durch einen zusätzlichen Inverter vor dem C-Eingang kann das Master-Slave-D-Flipflop so modifiziert werden, dass es seinen Zustand jeweils mit der *steigenden* Taktflanke aktualisiert.

Das Schaltungssymbol eines flankengesteuerten Flipflops markiert den Takteingang mit zwei zusammenlaufenden Schrägstrichen (vgl. Abbildung 2.15).

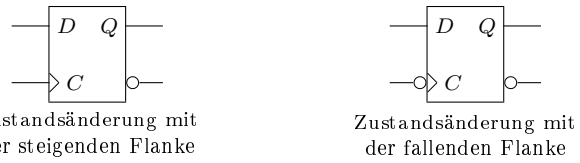


Abbildung 2.15: Schaltungssymbole einflankengesteuerter D-Flipflops

Register

Register sind Speicherelemente, die N-stellige Datenwerte aufnehmen können. Ein einzelnes D-Flipflop stellt somit bereits ein 1-bit-Register dar. Ein N-bit-Register bildet man durch eine Kette von N D-Flipflops, deren Takteingänge miteinander verbunden sind.

Abbildung 2.16 zeigt den *Grundaufbau* eines 4-bit-Registers mit paralleler Ein- und Ausgabe.

[24] Ein pegelgesteuertes Flipflop bezeichnet man auch als *Latch*.

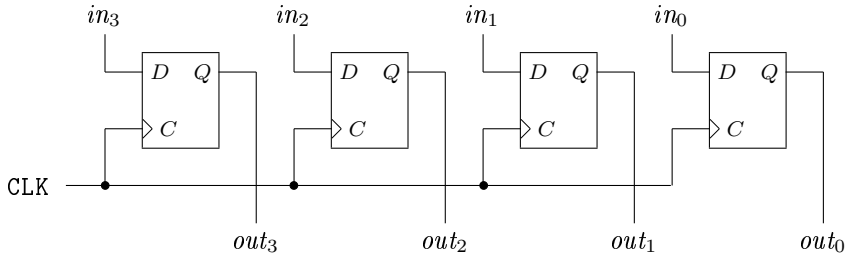


Abbildung 2.16: Grundschaltung eines 4-bit-Registers

Dieser Grundaufbau lässt sich durch uniformes Hinzuschalten weiterer D-Flipflops leicht auf 8-, 16- oder mehr Bit verbreitern.

Typischerweise hat ein Register über die Grundschaltung hinaus weitere Eingänge, wie etwa die Steuereingänge `CLEAR` und `ENABLE`. Während `CLEAR` den Registerinhalt auf 0 setzt, erlaubt $\overline{\texttt{ENABLE}}$ das Register so zu sperren, dass es keine neue Eingabe übernimmt.

Die einfachste Art, dies zu erreichen, ist das Abschalten des Taktsignals durch eine UND-Verknüpfung mit `ENABLE`. Dies wird als *Clock Gating* bezeichnet. Obgleich Clock Gating durchaus Vorteile bei der Leistungsaufnahme mit sich bringt, verursacht es einen Taktversatz (engl. *clock skew*), der i. allg. unerwünscht ist. Empfohlen wird daher ein synchroner Entwurf, der den Takt frei laufen lässt und das `ENABLE` über eine logische Verknüpfung mit den Eingangssignalen realisiert.

Für das i-te D-Flipflop wird dieses Verhalten durch folgende Funktionstabelle beschrieben:

CLEAR	ENABLE	D
0	0	out_i
0	1	in_i
1	0	0
1	1	0

So kann der Grundaufbau aus Abbildung 2.2.3 erweitert werden, indem jedem D-Flipflop ein zusätzliches, stets gleich aufgebautes Schaltnetz vorgeschaltet wird, wie es Abbildung 2.17 zeigt.

Eine zweite Registervariante ist das *Schieberegister*. Hier sind die D-Flipflops so in Reihe geschaltet, dass sie mit jedem Taktereignis ihren Inhalt an ihren linken bzw. ihren rechten Nachbarn weitergeben.

Schieberegister

Der Grundaufbau eines Schieberegisters ist in Abbildung 2.18 gezeigt. Es handelt sich um ein 4-bit-Schieberegister, das seine Daten seriell einliest, d. h. mit jeder steigenden Taktflanke genau ein Bit am seriellen Eingang *in* übernimmt und gleichzeitig alle bisher gespeicherten Bits um eine Position weiter schiebt. Das Bit, das am weitesten vom Eingang entfernt ist, geht verloren. Der gespeicherte Inhalt kann zu jeder Zeit parallel ausgelesen werden.

Auch Schieberegister können durch Steuer- bzw. Selektionsleitungen so angesteuert werden, dass sie spezielle Eigenschaften oder Verwendungszwecke erfüllen. So kann etwa über eine Steuerleitung $\overline{\texttt{L}}/\texttt{R}$ die Schieberichtung eingestellt werden ($\overline{\texttt{L}}/\texttt{R} = 0$ links, $\overline{\texttt{L}}/\texttt{R} = 1$ rechts). Über weitere Selektionsleitungen kann vorgegeben werden, ob das neu eingefügte Bit

- vom seriellen Eingang *in* übernommen werden soll,

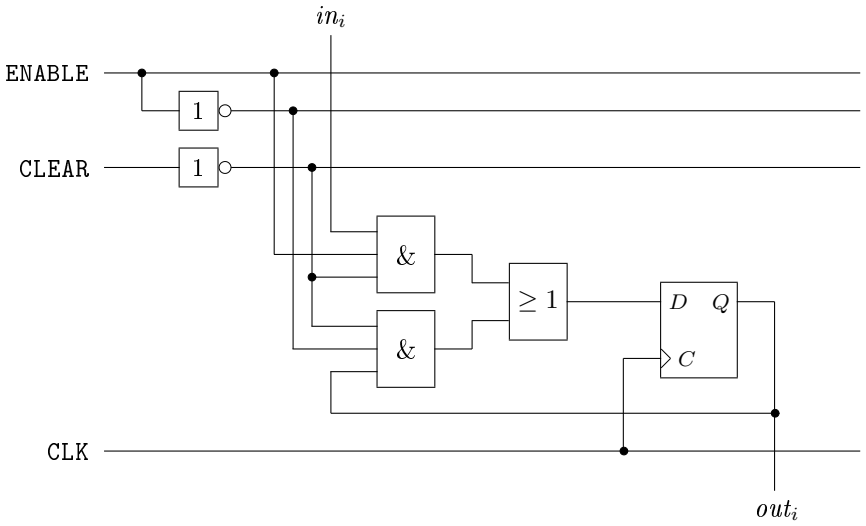


Abbildung 2.17: i-tes Flipflop mit Schaltnetz für `CLEAR` und `ENABLE` vor dem Eingang

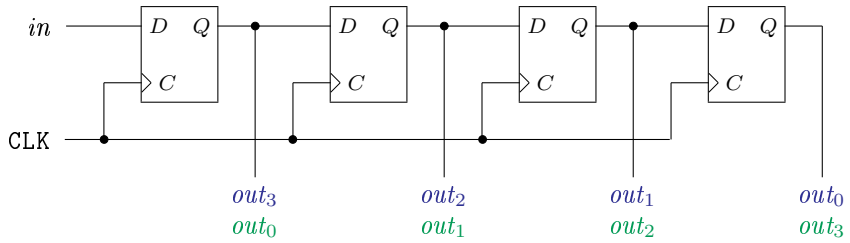


Abbildung 2.18: Grundschaltung eines 4-bit-Rechts- bzw. Linksschieberegisters

- stets den konstanten Wert `0` erhalten soll,
- seinen alten Wert duplizieren soll, oder
- den Wert des am anderen Ende herausgeschobenen Bits übernehmen soll.

Typische Anwendungsfälle für diese Selektionsalternativen sind

- die Seriell/Parallelwandlung von Eingabedaten,
- die Multiplikation mit 2, die dem Linksschieben mit Nachziehen einer `0` entspricht, oder die vorzeichenlose ganzzahlige Division durch 2, die durch ein Rechtsschieben mit Nachziehen einer `0` dargestellt werden kann,
- die vorzeichenerhaltende ganzzahlige Division durch 2, die beim Rechtsschieben den Wert der höchstwertigen Stelle erhalten muss, oder
- die Benutzung des Schieberegisters als Umlaufspeicher, der ermöglicht, Daten seriell an einer Ausgangsleitung auszulesen, ohne sie dabei zu verlieren. Dazu muss das ausgelesene Bit über eine Rückleitung am anderen Ende wieder eingelesen werden. Man nennt dies auch Links- oder Rechts*rotation*.

Zähler

Zähler sind Register-Schaltungen, die einen Zählwert speichern. Jeder Impuls am Zähleingang bewirkt die Änderung des gespeicherten Zählwerts um einen Zählschritt. Der Zählerstand kann über Ausgangsleitungen abgelesen werden.

Bei **Synchronzählern** werden alle Flipflops vom selben Takt gesteuert und schalten somit synchron. Die Berechnung des Zählerstands wird durch zusätzliche Logik vor den Flipflop-Eingängen realisiert.

Abbildung 2.19 zeigt einen modulo-16-Vorwärtszähler, der auf D-Flipflops basiert und reihum die Zählerstände `0000`, `0001`, `0010`, ..., `1111` erzeugt. Der Zählimpulsgeber Φ ist auf die Takteingänge geschaltet.

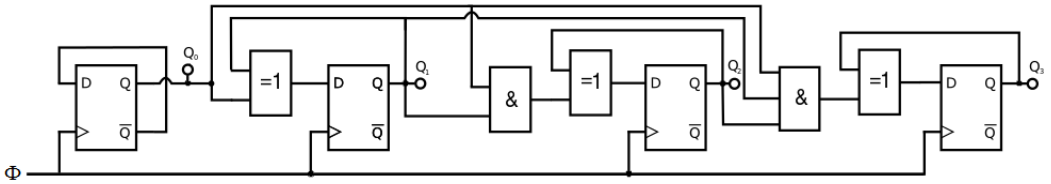


Abbildung 2.19: Synchroner 4-bit-Vorwärtszähler

Welcher Wert beim nächsten Zählimpuls am Eingang des i-ten D-Flipflops liegen muss, leitet sich aus dem jeweils zuvor gespeicherten Zählwert ab:

i	Q_3	Q_2	Q_1	Q_0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

- D_0 ist stets der invertierte vorhergehende Zustand dieses Flipflops, also $\overline{Q}_0$.
- D_1 wird 1, wenn $Q_1 = 0$ und $Q_0 = 1$ war und bleibt 1, wenn $Q_1 = 1$ und $Q_0 = 0$ war.
- D_2 wird 1, wenn $Q_2 = 0$ und $Q_1 = Q_0 = 1$ war und bleibt 1, wenn $Q_2 = 1$ war und Q_1 und Q_0 nicht gleichzeitig 1 waren.
- D_3 wird 1, wenn $Q_3 = 0$ und $Q_2 = Q_1 = Q_0 = 1$ war und bleibt 1, wenn $Q_3 = 1$ war und Q_2, Q_1 und Q_0 nicht gleichzeitig 1 waren.

Üblicherweise werden Zähler noch mit einem `RESET`-Eingang ausgestattet, der sie auf einen Anfangszählwert, i. d. R. Null zurücksetzt. Mit entsprechender Logik wäre es möglich, auch Zähler mit größeren Zählschritten oder unregelmäßigen Zählmustern zu entwerfen.

Synchronzähler sind Spezialfälle von *Schaltwerken*.

2.2.4 Schaltwerke

Schaltwerke (engl. *sequential circuits*) sind Schaltnetze mit der zusätzlichen Fähigkeit, einen *Zustand* zu speichern.

Die Ausgangsvariablen hängen im Unterschied zum Schaltnetz nicht nur ausschließlich von den Eingangsvariablen, sondern auch vom jeweiligen Zustand des Systems ab. Dieser Zustand wird für jeweils eine Taktperiode in einem Register gespeichert.

Der neue Zustand Z^{t+1} bestimmt sich einerseits aus dem alten Zustand Z^t und andererseits aus den Werten der Eingangsvariablen.

Der prinzipielle Aufbau eines Schaltwerks ist in Abbildung 2.20 gezeigt.

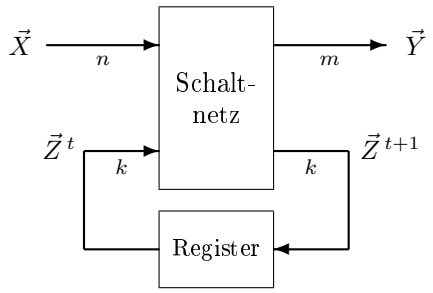


Abbildung 2.20: Prinzipieller Aufbau eines Schaltwerks

Solche Schaltwerke bezeichnet man auch als MEALY-Schaltwerke[25]. Eine Sonderform sind MOORE-Schaltwerke[26], bei denen die Ausgabe *nur* vom Zustand abhängt. Im einfachsten Fall können die Zustandsvariablen hier direkt als Ausgangsvariablen verwendet werden.

MEALY-Schaltwerke werden abstrakt durch *endliche Automaten* beschrieben. Ein MEALY-*Automat* ist ein 6-Tupel

$$\mathcal{A} = (X,\, Y,\, Z,\, \delta,\, \lambda,\, z_0) \tag{2.12}$$

mit

- einer endlichen, nicht-leeren Menge X von Eingabesymbolen (dem *Eingabealphabet*),
- einer endlichen, nicht-leeren Menge Y von Ausgabesymbolen (dem *Ausgabealphabet*),
- einer endlichen, nicht-leeren Menge Z von (inneren) Zuständen mit einem ausgezeichneten *Startzustand* $z_0 \in Z$,
- einer *Zustandsübergangsfunktion* $\delta:\ X \times Z \to Z$,
- und einer *Ausgabefunktion* $\lambda:\ X \times Z \to Y$.

Insbesondere die Funktionen δ, λ können durch eine äquivalente, anschaulichere Darstellung des Automaten beschrieben werden – den *Automatengraphen*.

Beispiel 2.21 Der MEALY-Automat $\mathcal{A} = (\{0,1\},\, \{y,n\},\, \{q_0, p_0, p_1\},\, \delta,\, \lambda,\, q_0)$ mit

δ:

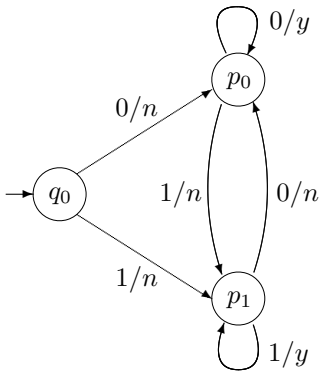
$z\backslash x$	0	1
q_0	p_0	p_1
p_0	p_0	p_1
p_1	p_0	p_1

λ:

$z\backslash x$	0	1
q_0	n	n
p_0	y	n
p_1	n	y

[25] nach GEORGE H. MEALY, 1955
[26] nach EDWARD F. MOORE, 1956

wird durch folgenden Automatengraphen dargestellt:



Wir benutzen allgemein die Markierung a/b für eine Kante vom Zustand z_i in den Zustand z_j, um anzuzeigen, dass $\delta(z_i, a) = z_j$ und $\lambda(z_i, a) = b$ ist.

Die Antwort von $\mathcal{A}$ auf die Eingabe 011000... ist $nnynyy$..., wobei die Zustandsfolge q_0, p_0, p_1, p_1, p_0, p_0, p_0, ... durchlaufen wird. $\mathcal{A}$ gibt somit jeweils aus, ob im Eingabestrom ein wiederholtes Bit auftritt. Man beachte, dass sich p_0 eine 0 und p_1 eine 1 merkt. Der Zustand q_0 ist der Anfangszustand und „erinnert“ sich, dass bisher noch keine Eingabe stattfand. □

Der Automatengraph ist auch die bevorzugte Darstellungsform beim Erstellen einer Schaltwerksspezifikation.

Beispiel 2.22 Ein Getränkemünzautomat verkauft Getränke zum Preis von 3€. Er akzeptiert ausschließlich 1€- oder 2€-Münzen. Sobald der Verkaufspreis erreicht oder überschritten ist, wird ein Getränk ausgegeben. Bei Überzahlung wird das Restgeld ebenfalls ausgegeben.

Abbildung 2.21 zeigt einen entsprechenden Automatengraphen. Dabei bedeuten „-“ keinen Münzeinwurf, „k“ keine Ausgabe, „G“ die Ausgabe eines Getränks und „GR“ die Ausgabe eines Getränks plus Rückgeld.

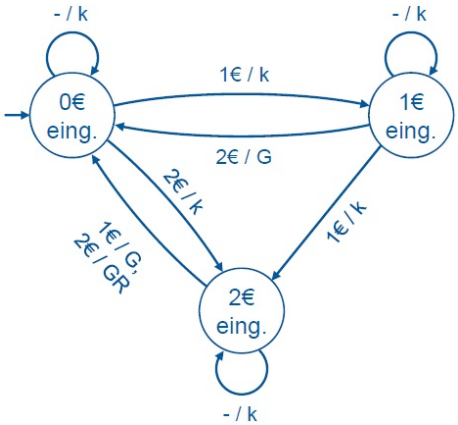


Abbildung 2.21: MEALY-Graph für den Getränkemünzautomaten

Die Eingabe „-“ ist für die Spezifikation per MEALY-Automat an sich noch nicht zwingend erforderlich, wird jedoch bereits explizit vorgesehen, um bei einer technischen Realisierung die Asynchronität von Münzeinwurf und Takt abzubilden. □

Die **technische Realisierung** eines durch einen MEALY-Automaten spezifizierten Schaltwerks beginnt damit, binäre Codierungen für die Symbolmengen X, Y und Z festzulegen.

Für Beispiel 2.22 können folgende Codierungen gewählt werden:

Eingabe:

x	Code
-	00
1€	01
2€	10

Ausgabe:

y	Code
k	00
G	10
GR	11

Zustände:

z	Code
0€ eing.	00
1€ eing.	01
2€ eing.	10

Mit diesen Festlegungen kann im zweiten Schritt die *Automatentabelle* aufgestellt werden:

Zustand $z_1\,z_0$	Eingabe $x_1\,x_0$	Ausgabe $y_1\,y_0$	Folgezustand $z'_1\,z'_0$
0 0	0 0	0 0	0 0
	0 1	0 0	0 1
	1 0	0 0	1 0
	1 1	D D	D D
0 1	0 0	0 0	0 1
	0 1	0 0	1 0
	1 0	1 0	0 0
	1 1	D D	D D
1 0	0 0	0 0	1 0
	0 1	1 0	0 0
	1 0	1 1	0 0
	1 1	D D	D D
1 1	∗ ∗	D D	D D

Aus der Automatentabelle werden im dritten Schritt die Funktionen δ und λ abgeleitet.

Ausgabe

z_1,z_0 \ x_1,x_0	00	01	11	10
00			D	
01			D	1
11	D	D	D	D
10		1	D	1

$$y_1 = x_0z_1 + x_1z_0 + x_1z_1$$

Folgezustände

z_1,z_0 \ x_1,x_0	00	01	11	10
00			D	1
01		1	D	
11	D	D	D	D
10	1		D	

$$z'_1 = x_0z_0 + \overline{x}_0\overline{x}_1z_1 + x_1\overline{z}_0\overline{z}_1$$

z_1,z_0 \ x_1,x_0	00	01	11	10
00			D	
01			D	
11	D	D	D	D
10			D	1

$$y_0 = x_1z_1$$

z_1,z_0 \ x_1,x_0	00	01	11	10
00		1	D	
01	1		D	
11	D	D	D	D
10			D	

$$z'_0 = \overline{x}_0\overline{x}_1z_0 + x_0\overline{z}_0\overline{z}_1$$

Nun kann im vierten und letzten Schritt ein Schaltbild[27] angegeben werden.

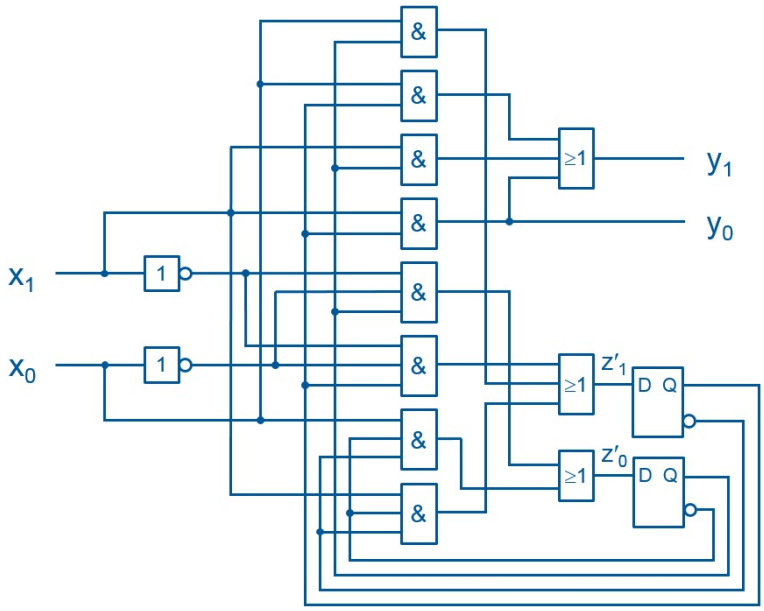


Abbildung 2.22: Schaltbild zu Beispiel 2.22

Die Minimierung der Ausgabe- und der Zustandsübergangsfunktion, die wir hier per KARNAUGH-Verfahren durchgeführt haben, ist wichtig, jedoch nicht die alleinige Möglichkeit der Minimierung. Eine ebenso wichtige Methode ist die *Zustandsreduktion*. Hier prüft man mit geeigneten Algorithmen, ob es äquivalente Zustände im Automaten gibt. Zustände werden als äquivalent betrachtet, wenn sie für dieselbe (beliebige) Folge von Eingabesymbolen auch jeweils dieselbe Folge von Ausgabesymbolen erzeugen. Äquivalente Zustände können zu einem einzigen Zustand zusammengefasst werden. Schließlich bietet auch die geschickte Wahl der Codierungen, insbesondere der *Zustandscodierung*, noch weiteres Optimierungspotential. Auch dafür sind bewährte (heuristische) Algorithmen bekannt. Auf die Verfahren zur *Automatenminimierung* soll in dieser Einführungsvorlesung jedoch nicht mehr eingegangen werden.

2.3 VON NEUMANN-Rechner

1942 begannen J. PRESPER ECKERT und JOHN W. MAUCHLY an der Universität von Pennsylvania mit der Entwicklung eines Rechners, den sie ENIAC (*Electronic Numerical Integrator and Computer*) nannten und der als der erste programmierbare vollelektronische digitale Rechner gilt. Die Dateneingabe erfolgte über dekadische Drehschalter und der Datenfluss eines Programms musste mittels Steckverbindungen manuell eingerichtet werden. Nachdem der ENIAC 1946 offiziell in Betrieb ging und der US-Armee zur Berechnung ballistischer Tabellen diente, wurde schnell deutlich, dass das manuelle Umprogrammieren des ENIAC durch Umstecken von Verbindungskabeln zu umständlich war. Diese Schwäche sollte beim Nachfolgemodell EDVAC (*Electronic Discrete Variable Automatic Computer*) vermieden werden. Im Zuge der Planung des EDVAC ergab sich ein Kontakt zwischen zwei Entwicklern aus der ENIAC-Gruppe, ARTHUR W. BURKS und HERMAN H. GOLDSTINE, und JOHN VON NEUMANN, einem anerkannten Mathematikprofessor an der Princeton

[27] In Abbildung 2.22 wurden die Takteingänge der D-Flipflops aus Gründen der Übersichtlichkeit unterdrückt. Sie sind aber selbstverständlich vorhanden.

University. Aus der gemeinsamen Diskussion entstand ein Konzeptpapier für einen *speicherprogrammierten* Rechner [10], das als Grundstein der konventionellen Rechnerarchitektur angesehen wird.

Das VON NEUMANN-Konzept bildet trotz aller Weiterentwicklung und Verfeinerung auch heute noch die Grundlage für fast alle modernen Rechner. Es basiert auf folgenden Merkmalen:

- Ein Rechner wird logisch und räumlich in folgende Funktionseinheiten strukturiert:
  1. in ein *Speicherwerk*, in dem Programme, Daten und (Zwischen-) Ergebnisse gespeichert werden,
  2. in ein *Rechenwerk*, in dem arithmetische und logische Verknüpfungen durchgeführt werden,
  3. in ein *Leitwerk* oder *Steuerwerk*, das einen Programmablauf steuert und
  4. in ein oder mehrere *Ein-/Ausgabewerke*, über die Daten und Programme ein- und ausgegeben werden.

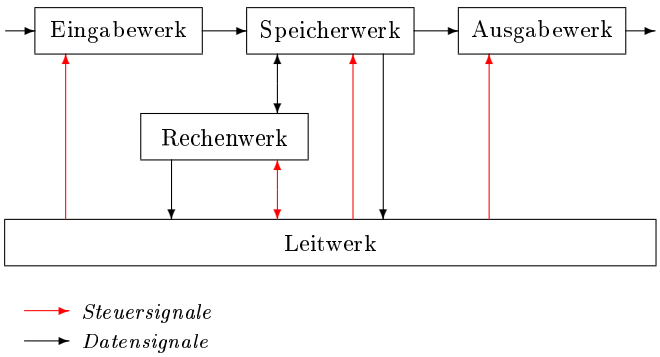


Abbildung 2.23: Struktur eines VON NEUMANN-Rechners

- Die Struktur des Rechners ist *unabhängig* von speziellen, zu bearbeitenden Problemen. Vielmehr wird für jedes Problem eine Bearbeitungsvorschrift, das *Programm*, von außen eingegeben und im Speicher abgelegt. Erst dieses Programm macht den Rechner arbeitsfähig.
- Programme und von diesen benötigte Daten sowie Zwischen- und Endergebnisse werden in *einem einheitlichen* Speicher abgelegt.
- Der Speicher besteht aus fortlaufend numerierten Speicherplätzen fester Wortlänge. Die Nummer heißt *Adresse*. Der Inhalt eines Speicherplatzes ist über diese Adresse abrufbar.
- Befehle eines Programms werden im allgemeinen aus aufeinanderfolgenden Speicherplätzen geholt. Diese *sequentielle* Verarbeitung kann jedoch durch Sprungbefehle oder datenabhängige Verzweigungen unterbrochen werden.

Die konzeptionelle Struktur nach VON NEUMANN wurde mit den ersten Realisierungen Schritt für Schritt verfeinert. So sind im Urkonzept je zwei Werke durch Stichleitungen miteinander verbunden. Sehr bald ging man jedoch dazu über, die Werke über eine gemeinsame Datenschiene (*Bus*) miteinander zu verbinden[28]. Ein Bus ist ein Bündel von parallelen Leitungen, über das Daten, Befehle, Adressen oder Steuersignale übertragen werden können.

[28] Der Bus als Sammelschiene taucht erstmals Mitte der 1960er Jahre auf (DEC PDP-8).

Des Weiteren legte das klassische Konzept das Modell der Akkumulatormaschine[29] zugrunde, wurde jedoch sehr bald auf das Modell der Registermaschine[30] erweitert.

Das Rechen- und das Leitwerk werden zusammen als *Zentraleinheit* (engl. *Central Processing Unit*, CPU) bezeichnet. Das Eingabe- und das Ausgabewerk werden zu einem einheitlichen E/A-Werk zusammengefasst, wobei „das“ E/A-Werk als Stellvertreter für viele prinzipiell gleichartige E/A-Werke zu verstehen ist – in herkömmlichen Arbeitsplatzrechnern gibt es je ein E/A-Werk für Tastatur und Maus, für den Monitor, die Festplatte, das optische Laufwerk, die Netzwerkschnittstelle, die USB-Schnittstelle, etc. pp.

Abbildung 2.24 zeigt das schematische Gesamtbild eines entsprechend modifizierten VON NEUMANN-Rechners. Die einzelnen Werke werden in den anschließenden Abschnitten detaillierter betrachtet.

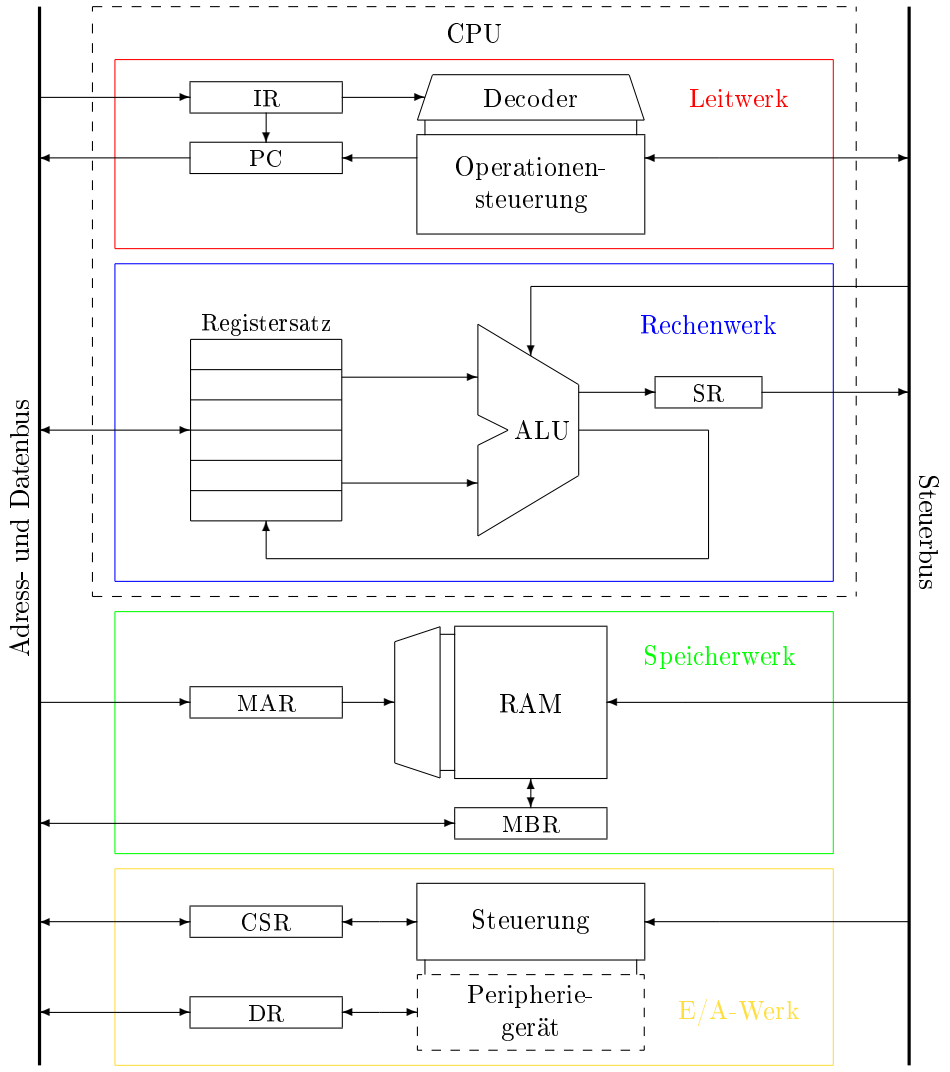


Abbildung 2.24: Schematisches Gesamtbild eines VON NEUMANN-Rechners

[29] Eine *Akkumulatorarchitektur* verfügt nur über ein einziges Rechenregister, das *Akkumulator* (AC) genannt wird. Jeder Befehl bezieht sich implizit immer auf AC. Der Befehl `ADD 100` bedeutet bspw. AC := AC + Memory[100].

[30] Eine *Registerarchitektur* stellt mehrere Rechenregister zur Verfügung. Typische Befehle weisen bis zu drei Operanden aus: ein Zielregister und zwei Quellregister. Der Befehl `ADD R3,R2,R1` bedeutet bspw. R3 := R2 + R1.

2.3.1 Leitwerk

Das Verhalten eines Rechners wird durch die Ausführung gespeicherter Maschinenprogramme bestimmt. Die Steuerung des Programmablaufs ist Aufgabe des Leitwerks (engl. *control unit*). Das Leitwerk holt die Maschinenbefehle aus dem Speicher, entschlüsselt sie und steuert ihre Ausführung. Zudem koordiniert das Leitwerk die Zusammenarbeit der einzelnen Werke.

Die wichtigsten Bestandteile eines Leitwerks sind

- ein *Befehlsregister* (engl. *instruction register*, IR) zur Speicherung des Befehls, der aktuell zur Ausführung ansteht,
- ein *Befehlszähler* (engl. *program counter*, PC)[31], der angibt, welcher Befehl als nächstes aus dem Speicher geholt werden soll,
- ein *Befehlsdecodierer* zur Entschlüsselung eines Befehls und
- die *Operationensteuerung* oder *Ablaufsteuerung*[32], die schließlich die Ausführung jedes Befehls steuert.

Die Steuerung des Programmablaufs erfolgt bei einem VON NEUMANN-Rechner Befehl für Befehl nach einem gleichbleibenden Muster – dem sog. **Befehlszyklus** oder VON NEUMANN-*Zyklus*. Ein Befehlszyklus besteht aus einer festen Abfolge verschiedener Phasen:

- In der **Befehlsholphase** (engl. *Instruction Fetch*) wird der Befehl, der durch den Befehlszähler (PC) adressiert ist, aus dem Hauptspeicher in das Befehlsregister (IR) geladen. Der Speicherzugriff erfolgt mit Hilfe der beiden Speicherwerksregister MAR und MBR, die die Schnittstellen zum Adress- bzw. zum Datenbus darstellen. Darüberhinaus wird standardmäßig der Befehlszähler um einen Befehl weitergezählt.
- In der anschließenden **Entschlüsselungsphase** (engl. *Decode*) werden aus dem Operationscode des Maschinenbefehls die durchzuführende Operation sowie die Art des Zugriffs auf die benötigten Operanden abgeleitet und entsprechende Steuer- und Auswahlsignale für die Operationensteuerung oder die Werke generiert.
- Die folgende **Operandenholphase** (engl. *Load*) sorgt dafür, dass alle für die Befehlsausführung notwendigen Operanden im Rechenwerk bereitgestellt werden. Für manche Operanden ist hierfür (wie in der Befehlsholphase) ein Speicherzugriff durchzuführen. Unterstützt der Befehlssatz auch indizierte Adressierungsarten[33], können zusätzliche Adressberechnungen notwendig werden.
- Die **Ausführungsphase** (engl. *Execute*) führt die mit dem jeweiligen Befehl assoziierte Operation aus. Ihre Steuerung kann je nach Befehlsgruppe unterschiedlich komplex sein. *Schiebebefehle* können unmittelbar im Register ausgeführt werden. *Transportbefehle* führen lediglich ein Kopieren von Registerinhalten durch. In ähnlicher Weise besteht die Ausführung von *Sprungbefehlen* darin, das Sprungziel in den Befehlszähler zu kopieren. Bei *bedingten* Sprungbefehlen hängt die Durchführung dieser Operation von einer Bedingung ab. Diese Bedingung bezieht sich ausschließlich auf den Zustand eines oder mehrerer Flags im Statusregister (SR) des Rechenwerks. Arithmetische und logische *Verknüpfungsbefehle* benötigen die ALU, die das Ergebnis berechnet und an ihrem Ausgang bereitstellt.

[31] Auch die Bezeichnung *instruction pointer* (IP) ist üblich, so z. B. in der Intel-Architektur.

[32] In der Literatur wird uneinheitlich sowohl für die Operationensteuerung, als auch für das gesamte Leitwerk gelegentlich der Begriff *Steuerwerk* gebraucht.

[33] Bei der indizierten Adressierung wird auf die aus anderen Adressangaben gebildete Adresse noch ein Index addiert.

Während RISC-Architekturen sich meist damit begnügen, nur die Grundrechenarten der ersten Stufe (Addition, Subtraktion) in Hardware zu implementieren, unterstützen CISC-Architekturen im Allgemeinen auch die Grundrechenarten der zweiten Stufe (Multiplikation, Division). Hier ist dann i. d. R. eine iterative Operationensteuerung erforderlich, d. h. im Rahmen eines Maschinenbefehls werden wiederkehrende Elementaroperationen abgearbeitet.

- Die abschließende **Rückschreibephase** (engl. *Write back* oder *Store*) dient der Speicherung des Ergebnisses am dafür vorgesehenen Ziel im Speicher oder im Registersatz.

Der Befehlszyklus wird durch die Operationensteuerung betrieben. RISC-Architekturen realisieren die Operationensteuerung meist *festverdrahtet* (engl. *hard-wired*) als Schaltwerk. In CISC-Architekturen ist es üblich, *mikroprogrammierte*[34] Operationensteuerungen zu nutzen.

2.3.2 Rechenwerk

Die zentralen Komponenten des Rechenwerks (engl. *execution unit*) sind der Registersatz und die arithmetisch-logische Einheit. Der Registersatz (engl. *register set*) umfasst alle allgemeinen Rechenregister, welche durch Maschinenprogramme direkt angesprochen werden können. Die arithmetisch-logische Einheit (ALU, engl. *Arithmetic-Logic Unit*) berechnet arithmetische und logische Verknüpfungen. Sie ist ein reines Schaltnetz, das Eingänge für zwei Operanden und für Steuersignale zur Auswahl der ALU-Operation sowie Ausgänge für das Ergebnis und für das Statuswort besitzt.

Das Statuswort wird im *Statusregister* SR abgelegt. Dieses Register enthält (mindestens) vier wichtige Statusbits, sog. *flags*:

CF Das **Übertragsbit** (engl. *Carry Flag*) zeigt bei einer Addition an, ob ein Übertrag in der höchstwertigen Stelle auftritt bzw. bei einer Subtraktion, ob dort ein Übertrag (*borrow*) benötigt wird.

ZF Das **Nullbit** (engl. *Zero Flag*) zeigt an, ob das Ergebnis der letzten Rechenoperation gleich Null war.

SF Das **Vorzeichenbit** (engl. *Sign Flag*) zeigt an, ob das Ergebnis der letzten Rechenoperation negativ war, d. h. bei einer vorzeichenbehafteten Berechnung entspricht es dem höchstwertigen Bit des Ergebnisses.

OF Das **Überlaufbit** (engl. *Overflow Flag*) zeigt eine Bereichsüberschreitung an. Bei einem Rechner, der mit Wortlänge N im Zweierkomplement arbeitet, wird dieses Bit gesetzt, wenn das positive Ergebnis größer als $2^{N-1} - 1$ oder das negative Ergebnis kleiner als -2^{N-1} ist.

2.3.3 Speicherwerk

Das Speicherwerk hat die Aufgabe, sowohl das aktuell ausgeführte Programm als auch die Daten, die im Zuge der Ausführung dieses Programms benötigt oder erzeugt werden, zwischenzuspeichern.

Die zentrale Komponente des Speicherwerks ist der Haupt- bzw. Arbeitsspeicher (engl. *main memory*, RAM, *random access memory*). Die Schnittstelle zu den anderen Werken wird gebildet durch das Speicheradressregister MAR (engl. *memory address register*) und das Speicherpufferregister MBR (engl. *memory buffer register*). Bei einem Lesezugriff muss das MAR die Speicheradresse enthalten, aus der gelesen werden soll. Das gelesene Datenwort wird vom Hauptspeicher im MBR

[34] Eine mikroprogrammierte Operationensteuerung zerlegt jeden Maschinenbefehl in eine Folge von Mikrooperationen. Diese Folge wird als Mikroprogramm bezeichnet und in einem schnellen Mikroprogrammspeicher abgelegt, von wo aus sie durch ein Mikroprogrammwerk (eine Art Rechner im Rechner) abgerufen und ausgeführt wird.

abgelegt, von wo aus es über den Bus in andere Werke weiter transferiert werden kann. Bei einem Schreibzugriff enthält das MAR ebenfalls die Zieladresse und das MBR das zu schreibende Datenwort. Die Schreib- oder Lese-Richtung wird dem Speicher durch ein Steuersignal $\overline{\mathrm{R}}/\mathrm{W}$ angezeigt. Meist besitzen Speicherchips auch noch einen *Chip Select*-Eingang CS, der genau dann aktiv geschaltet wird, wenn eine gültige Speicheradresse anliegt und ein Speicherzugriff durchzuführen ist.

Der interne Ablauf eines Schreib- oder Lesevorgangs wird nicht zentral vom Leitwerk, sondern lokal im Speicher selbst gesteuert.

2.3.4 Realisierung einer einfachen Zentraleinheit

Grundlagen für die in diesem Abschnitt studierte Realisierung sind der Befehlssatz von Brookshear aus Abschnitt 2.1.7 und das ebenfalls dort eingeführte Rechnermodell.

Spezialregister PC und IR

Die Register der Zentraleinheit sind realisiert wie in Abschnitt 2.2.3 und verfügen über einen `ENABLE`-Steuereingang, der im Folgenden jeweils mit `_ie` (für *input enable*) abgekürzt wird. Der Befehlszähler (Abb. 2.25) ist als Synchronzähler realisiert. Der Zählimpuls Φ wird als Signal `PC_incr` eingesteuert. Ein Rücksetzen des Zählers ist über das Signal `PC_reset` möglich. Der Wert eines Registers wird über einen 8-fach-Tristate-Bustreiber auf den Systembus geschaltet. Das Steuersignal hierfür heiße jeweils `_oe` (für *output enable*).

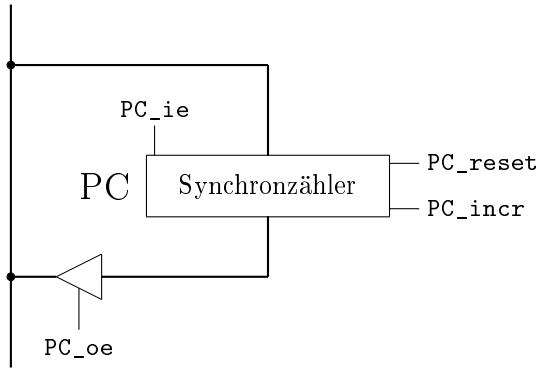


Abbildung 2.25: Befehlszähler

Das Befehlsregister (Abb. 2.26) ist als Doppelregister angelegt, das für beide Hälften einen eigenen `ENABLE`-Eingang besitzt. Bei Lade-, Speicher- und Sprungbefehlen befindet sich in der unteren Doppelregisterhälfte eine Adresse bzw. eine Konstante, die zur weiteren Befehlsausführung auf den Bus zurückgeführt werden muss.

Der Befehlszähler und das Befehlsregister spielen eine wichtige Rolle in der ersten Phase eines Befehlszyklus – der *Befehlsholphase*. Die Befehlsholphase wird (wie alle anderen Phasen des Befehlszyklus auch) als Teil der Operationensteuerung, d. h. als Schaltwerk realisiert.

Operationensteuerung

Um die Befehlsholphase im Einzelnen nachvollziehen zu können, ist es noch notwendig, das Speicherwerk eine Ebene detaillierter zu betrachten. Abbildung 2.27 zeigt seinen Aufbau. Das Spei-

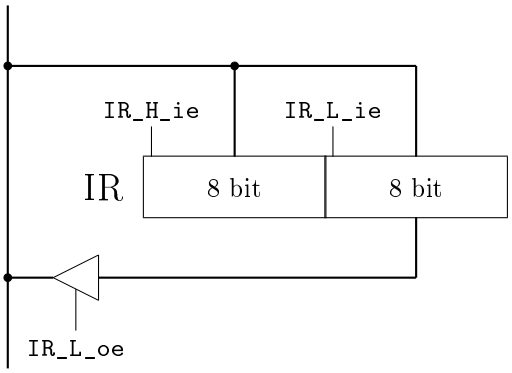


Abbildung 2.26: 16-bit-Befehlsregister

cherwerk verwendet einen[35] 1-Multiplexer[36], um den MBR-Eingang wahlweise mit dem Datenbus oder dem RAM-Ausgang zu verbinden.

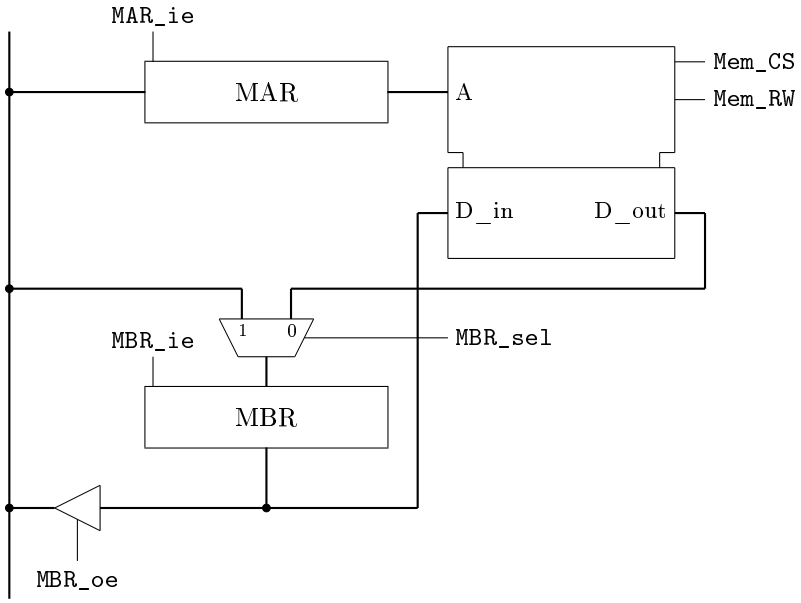


Abbildung 2.27: Speicherwerk

Damit sind alle Steuersignalleitungen eingeführt, die in der Befehlsholphase benötigt werden.

Abbildung 2.28 gibt einen Ausschnitt aus einem MEALY-Automatengraphen wieder, der die **Befehlsholphase** beschreibt. Im Automatengraphen sind nur diejenigen Steuersignale aufgeführt, die für diesen Automaten relevant sind. Zur Verdeutlichung werden Signale, die sich bei einer Transition ändern, farblich hervorgehoben.

[35] Genau genommen sind es acht 1-Multiplexer, da jede Stelle im Register ihren eigenen Multiplexer erfordert.

[36] Ein n-Multiplexer (n-MUX) schaltet denjenigen Eingang, dessen Nummer als Dualzahl über die n Selektoreingänge angelegt wird, auf den Ausgang durch (siehe dazu auch eine Übungsaufgabe).

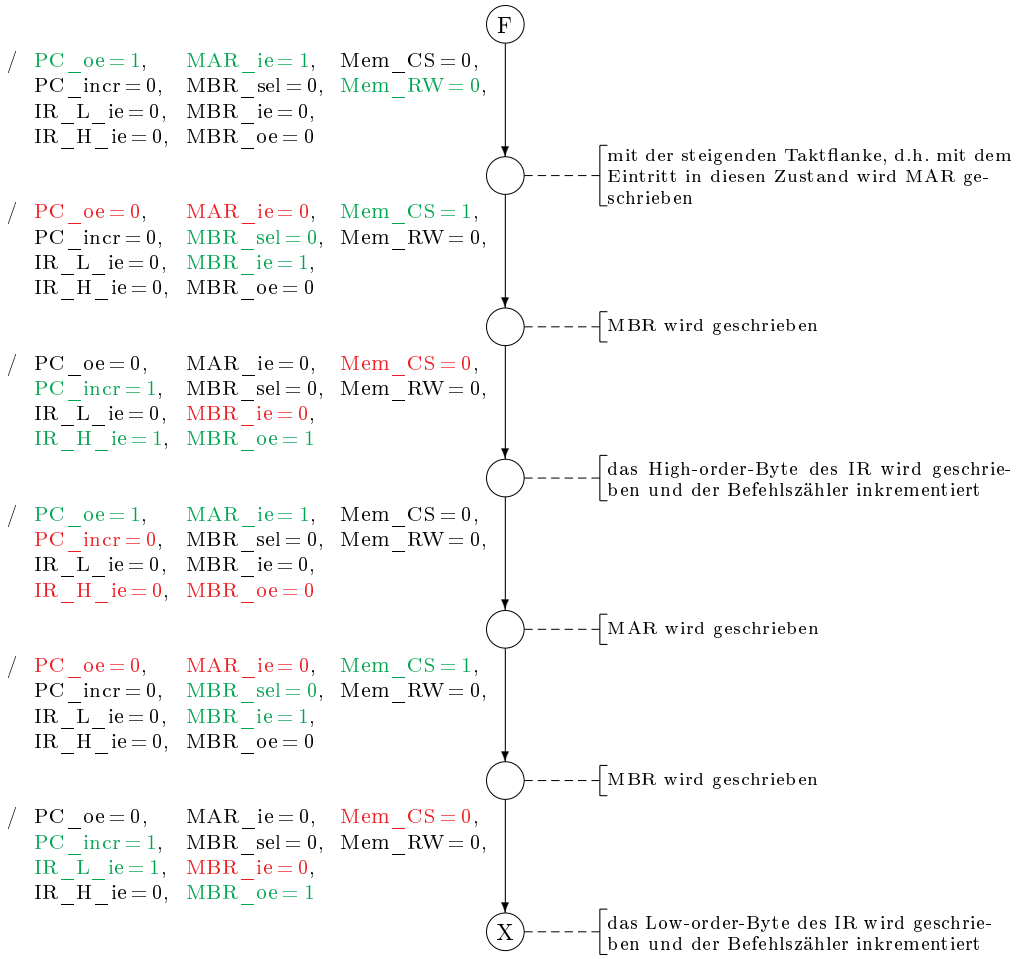


Abbildung 2.28: Befehlsholphase (Teil der Operationensteuerung)

Die **Entschlüsselungsphase** kann in einer RISC-Architektur mit fester Befehlslänge durch einen einfachen Binär-zu-1-aus-n-Decodierer realisiert werden[37]. In CISC-Architekturen mit variierenden Befehlslängen wird zunächst nur der Operationscode entschlüsselt und dann entschieden, welche Operandeninformation wohin nachzuladen ist.

Für unsere Zentraleinheit genügt ein einfacher Decodierer. Sein Eingang wird direkt mit den IR-Bitpositionen $15..12$ verbunden, in denen der Opcode jedes Befehls steht. Damit kann der Decodierer unmittelbar die entschlüsselten Auswahlsignale `_exec` erzeugen, die der Operationensteuerung anzeigen, welche Operation im Rahmen des jeweils aktuellen Befehls auszuführen ist.

Bei der Operandenauswahl sind einige Eigenarten des Befehlssatzes zu beachten: so zeigt ein Blick auf Tabelle 2.4, dass das Register für den Zieloperanden bei den meisten Befehlen durch die IR-Bitpositionen $11..8$ adressiert wird. Es gibt jedoch Ausnahmen: der `MOVE`-Befehl verwendet die Bitpositionen $3..0$ und die Befehle `STORE`, `JUMP` und `HALT` haben gar kein Zielregister. Zur Behandlung dieser Ausnahmen generiert der Decoder ein Operandenauswahlsignal `dst_sel` (das in diesem Fall identisch mit `MOVE_exec` sein könnte).

Ähnlich verhält es sich beim ersten Quelloperanden: er wird bei den meisten Befehlen durch die IR-Bitpositionen $7..4$ adressiert. Lediglich der `STORE`-, der `JUMP`- und der `ROTATE`-Befehl nutzen hierfür die Bitpositionen $11..8$ (bei letzterem ist der erste Quelloperand zugleich der Zieloperand).

[37] Einen solchen Decodierer haben wir in einer Übungsaufgabe entwickelt.

Ein Auswahlsignal für diese Sonderfälle heiße `src1_sel`. Die Befehle `LOAD` und `LOADI` adressieren bzw. nennen ihren Quelloperanden direkt im IR und `HALT` hat gar kein Quelloperanden.

Zur Adressierung des zweiten Quelloperanden nutzen alle Befehle (die ihn überhaupt benötigen) die IR-Bitpositionen 3 .. 0. Als Ausnahme erkennt man hier den `JUMP`-Befehl, der als zweiten Operanden implizit den Wert aus Register 0 verwendet. Dies kann über ein Auswahlsignal `src2_zero` gesteuert werden (das in diesem Fall identisch mit `JUMP_exec` wäre).

Abbildung 2.29 zeigt einen entsprechenden Befehlsdecodierer.

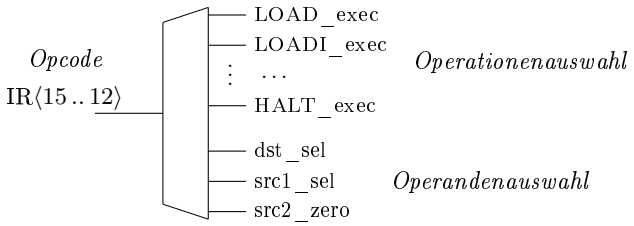


Abbildung 2.29: Befehlsdecodierer

Die restlichen drei Phasen des Befehlszyklus können für unsere Zentraleinheit zu einer einzigen **Ausführungsphase** zusammengefasst werden. Um die Ausführungsphase der unterschiedlichen Befehle nachvollziehen zu können, ist es notwendig, auch das Rechenwerk eine Ebene detaillierter zu betrachten. Abbildung 2.30 zeigt eine verfeinerte Ansicht, die alle notwendigen Multiplexer und Steuerleitungen enthält.

Die wesentlichen Funktionsblöcke des Rechenwerks sind der Registersatz und die ALU. Hinzu kommt das Statusregister SR, das zur Realisierung von Brookshears Befehlssatz lediglich das ZF bereitstellen muss. Das Erzeugen des ZF erfolgt durch eine NOR-Verknüpfung aller 8 bit des ALU-Ausgangs. Dies wird hier durch den Funktionsblock FlagGen dargestellt.

Der Zugriff auf den Registersatz erfolgt über Registernummern, die im jeweils aktuellen Befehl als Operandenadressen angegeben sind. Sie werden dem Registersatz direkt aus dem IR über Multiplexer zugeführt, die die bei der Decodierung angesprochenen Ausnahmen berücksichtigen [11]. Intern steuern die Quellregisternummern Multiplexer (4-MUX), die den jeweiligen Registerausgang auf die ALU-Eingänge durchschalten. Um die Register schreiben zu können, sind ihre Eingänge alle mit `dst_data` verbunden. Das Steuersignal `EN` wird über einen 1:16-Demultiplexer (4-DMUX), der von `dst_reg` gesteuert wird, auf die `ENABLE`-Eingänge der Rechenregister verteilt.

Die ALU muss neben der arithmetischen Addition die logischen Verknüpfungen AND, OR und XOR sowie die Rechtsrotation beherrschen. Diese Funktionen werden durch Selektorsignale angewählt, die hier als ein Signalbündel `ALU_op` dargestellt werden. Eine solche ALU wird anschließend entwickelt.

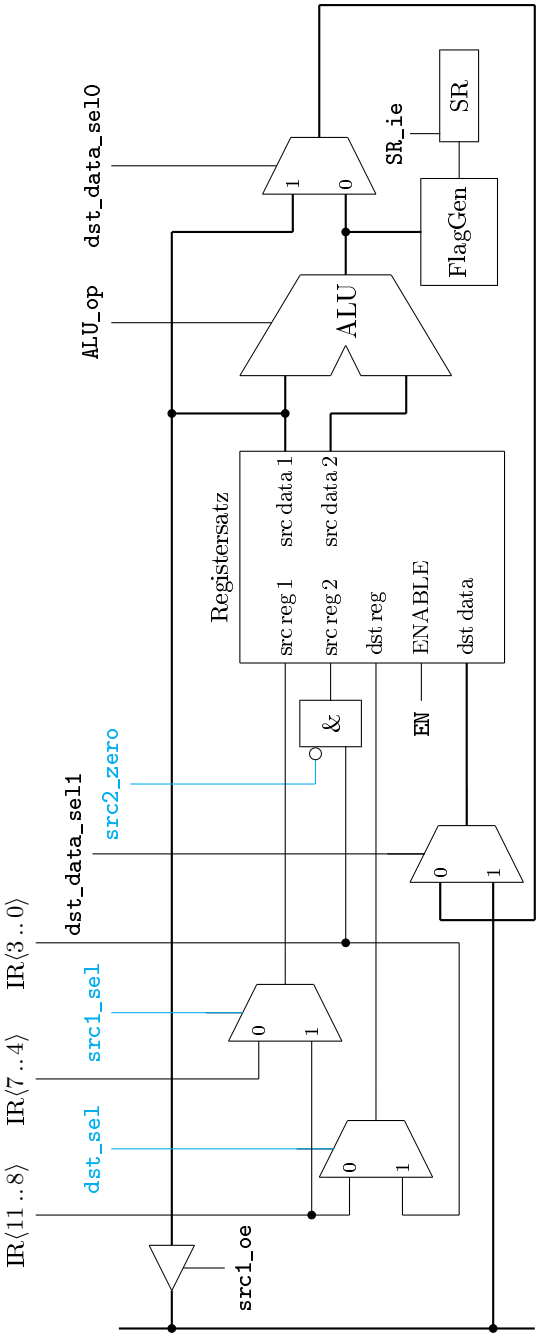


Abbildung 2.30: Rechenwerk

Die folgenden Abbildungen 2.31 bis 2.37 zeigen jeweils einen Ausschnitt des MEALY-Automaten, der die festverdrahtete Operationensteuerung für unsere Zentraleinheit realisiert. Es werden nicht alle in Tabelle 2.4 gelisteten Befehle entwickelt, sondern nur eine exemplarische Auswahl. Der Übersichtlichkeit halber werden in jedem Automaten(teil)graphen nur diejenigen Steuerleitungen angezeigt, die dort auch relevant sind. Die gesamte Operationensteuerung setzt sich aus all diesen Teilgraphen einschließlich des Teilgraphen für die Befehlsholphase aus Abbildung 2.28 zusammen. Konnektoren sind die Zustände F und X.

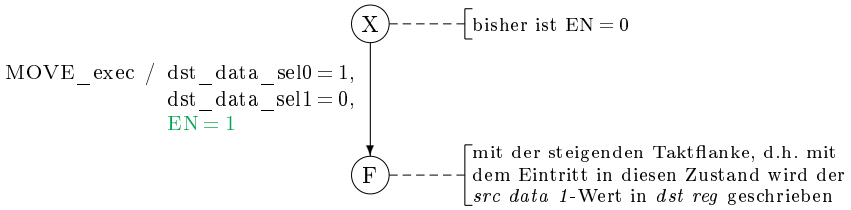


Abbildung 2.31: Operationensteuerung für den MOVE-Befehl

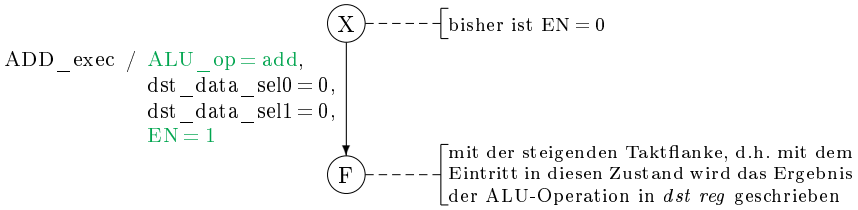


Abbildung 2.32: Operationensteuerung für die Addition

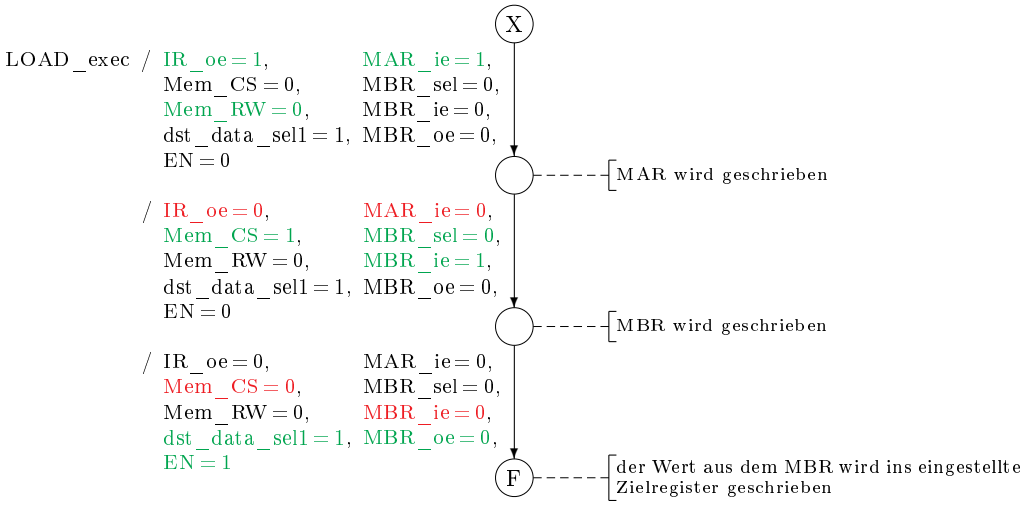


Abbildung 2.33: Operationensteuerung für den LOAD-Befehl

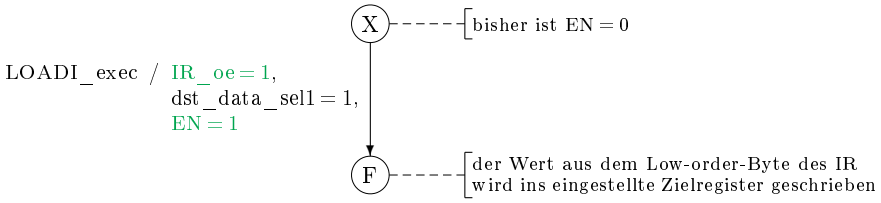


Abbildung 2.34: Operationensteuerung für den LOADI-Befehl

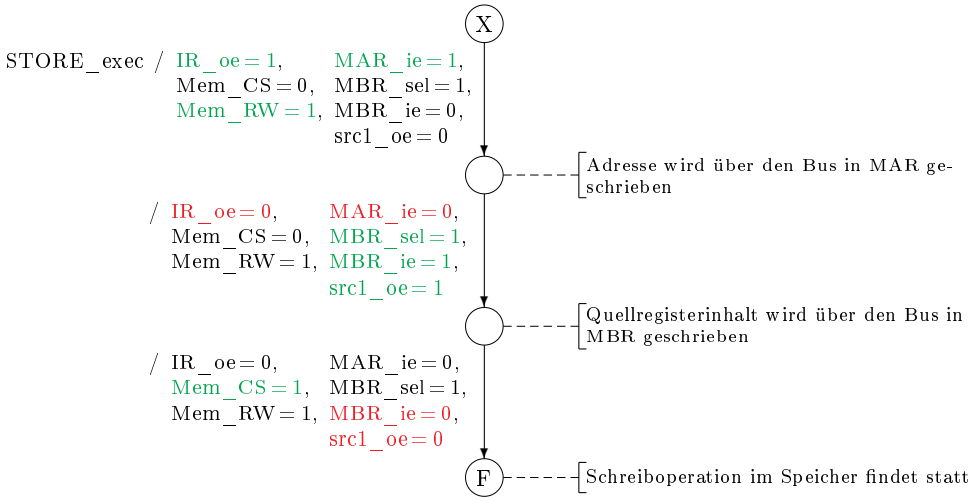


Abbildung 2.35: Operationensteuerung für den STORE-Befehl

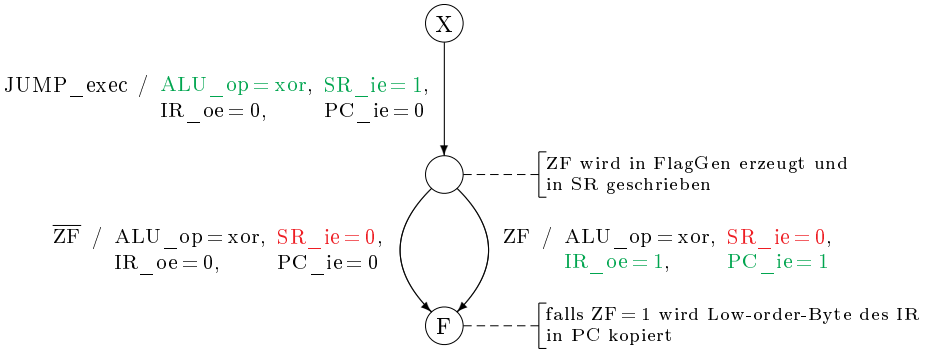


Abbildung 2.36: Operationensteuerung für den Sprungbefehl

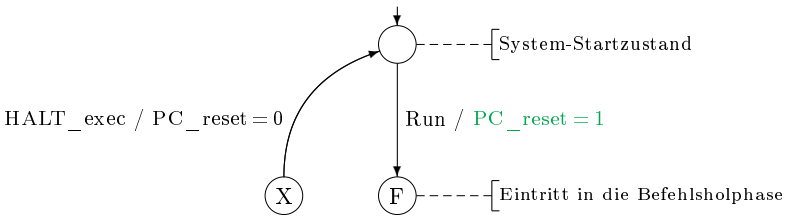


Abbildung 2.37: Operationensteuerung für den HALT-Befehl

ALU

Eine ALU ist ein reines Schaltnetz, das Eingänge für zwei Operanden und den Steuercode für die ALU-Operation besitzt. Ihre Ausgänge liefern das Ergebnis sowie neue Zustände für einzelne Flags des Statusregisters. Der Kern einer ALU ist ein *Addiernetz*.

Das Prinzip der **Addition** mit Binärzahlen hatten wir in Abschnitt 2.1.3 betrachtet. Die Rechenregel für die Addition der i-ten Stelle zweier Binärwörter a und b können wir durch folgende Funktionstabelle darstellen:

a_i	b_i	c_{i+1}	s_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Offenbar gilt:

$$\begin{aligned} c_{i+1} &= a_i \cdot b_i \\ s_i &= a_i \oplus b_i \end{aligned}$$

Ein entsprechendes Schaltnetz heißt *Halbaddierer* und ist in Abbildung 2.38 gezeigt.

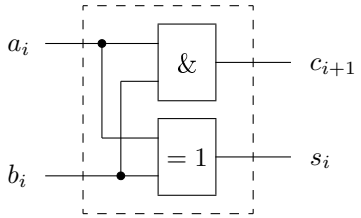


Abbildung 2.38: Halbaddierer

Ein Halbaddierer kann unmittelbar eingesetzt werden, um das niederwertigste Bit einer Summe zu berechnen. Bei allen höherwertigen Bits muss zusätzlich noch der Übertrag aus ihrer jeweils vorhergehenden Stelle berücksichtigt werden.

Ein Baustein, der dieses Schaltverhalten realisiert, heißt *Volladdierer*. Mit etwas Überlegung erkennt man, dass ein Schaltnetz für einen Volladdierer wie folgt aufgebaut werden kann: ein erster Halbaddierer addiert die beiden Operandenbits (Ergebnis s'_i). Zu diesem Ergebnis wird durch einen zweiten Halbaddierer das Übertragsbit c_i der vorhergehenden Stelle addiert (Ergebnis s''_i bzw. s_i). Ein Übertrag c_{i+1} in die nächsthöhere Stelle kann bei beiden Additionen anfallen, jedoch nie bei beiden gleichzeitig. Es genügt somit eine ODER-Verknüpfung der Übertragsausgänge der beiden Halbaddierer. Abbildung 2.39 zeigt das Schaltbild eines Volladdierers.

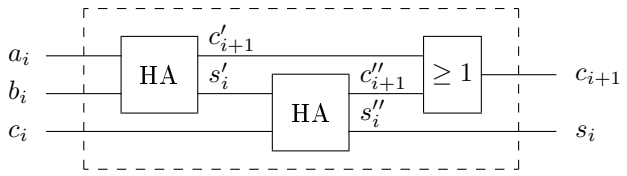


Abbildung 2.39: Volladdierer

Damit kann bereits ein Addiernetz konstruiert werden. Einen Paralleladdierer[38] für 4-stellige Binärzahlen zeigt Abbildung 2.40. Erweiterungen für mehr Stellen sind leicht vorstellbar. Da bei dieser Realisierung der Übertrag durch das gesamte Schaltnetz „rieselt“, bezeichnet man solche Addierer als *Ripple-Carry Adder*.

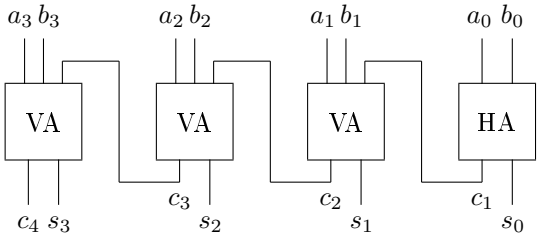


Abbildung 2.40: Ripple-Carry-Addierer

Wenden wir uns nun der Implementierung der zweiten arithmetischen Rechenoperation zu – der **Subtraktion**. In Abschnitt 2.1.4 hatten wir festgestellt, dass die Subtraktion auf die Addition zurückgeführt werden kann, indem das Zweierkomplement des Subtrahenden auf den Minuenden addiert wird.

Ein gegebenes Addiernetz kann somit leicht um die Zusatzfunktion „Subtraktion“ erweitert werden. Eine Steuerleitung $\overline{\texttt{add}}/\texttt{sub}$ zeigt an, ob addiert (0) oder subtrahiert (1) werden soll. Indem die Stellen des zweiten Operanden mit dieser Steuerleitung bitweise XOR-verknüpft werden, wird erreicht, dass der zweite Operand im Falle $\overline{\texttt{add}}/\texttt{sub} = 0$ (als Summand) unverändert auf den Eingang der ALU durchgeschaltet wird, hingegen im Falle $\overline{\texttt{add}}/\texttt{sub} = 1$ (als Subtrahend) stellenweise invertiert wird. Die anschließende Addition einer 1 wird einfach dadurch erreicht, dass im Addiernetz ein Übertragseingang c_0 in die niederwertigste Stelle geschaffen wird, d. h. der Halbaddierer in Stelle 0 in Abbildung 2.40 durch einen Volladdierer ersetzt wird. Bei einer Subtraktion muss $c_0 = 1$ gesetzt werden, bei einer Addition hingegen auf $c_0 = 0$ bleiben. Somit kann der c_0-Eingang direkt mit der Steuerleitung $\overline{\texttt{add}}/\texttt{sub}$ verbunden werden (siehe Abbildung 2.41).

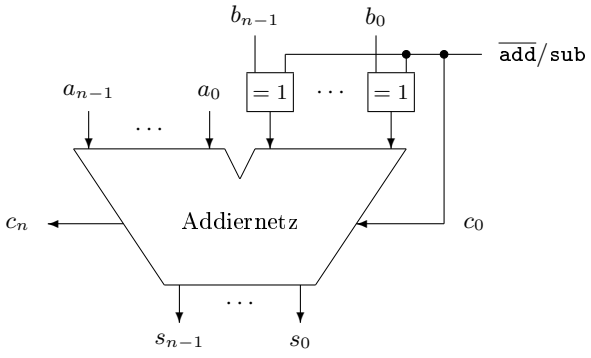


Abbildung 2.41: Additions- und Subtraktionsnetz

Die dritte arithmetische Rechenoperation ist die **Multiplikation**, die allerdings von Brookshears Befehlssatz (genauso wie dieSubtraktion und die Division) gar nicht unterstützt wird. Das Prinzip, nach dem eine Multiplikationsoperation implementiert werden kann, leitet sich unmittelbar aus der schriftlichen Methode ab, die in Abschnitt 2.1.3 dargestellt wurde. Dabei wird die Multiplikation auf eine *wiederholte Addition* abgebildet, bei der das jeweils erzeugte Zwischenergebnis nach jedem Schritt um eine Stelle nach rechts geschoben wird. Für ein Beispiel kann das Buch von Patterson

[38] Bei einer parallelen Addition werden alle Stellen gleichzeitig gebildet.

und HENNESSY [11] (Kap. 3.3) herangezogen werden. In entsprechender Weise könnte die **Division** auf eine *wiederholte Subtraktion* des Divisors vom Dividenden abgebildet werden, nachdem bei jedem Schritt der Dividend zuerst um eine Stelle nach links geschoben wird. Auch hierzu kann ein Beispiel in [11] nachgeschlagen werden (Kap. 3.4).

Offenbar genügt es also, dass die ALU als einzige wesentliche *arithmetische* Operation die Addition realisiert. Hinzu kommen allerdings noch *logische* Operationen.

Logische Operationen auf Registern verstehen sich als *bitweise* Verknüpfungen, d. h. je zwei gleichrangige Bitpositionen werden unabhängig von anderen Bitpositionen gemäß eines vorgegebenen BOOLEschen Operators miteinander verknüpft.

Die Erweiterung eines Additions- und Subtraktionsschaltnetzes zu einer ALU, die auch logische Verknüpfungen bearbeiten kann, folgt einem ähnlichen Gedanken wie der Aufbau eines Ripple-Carry-Addierers aus Volladdierern. Das Grundelement ist hier eine **1-bit-ALU**, die aus drei Teilschaltnetzen besteht:

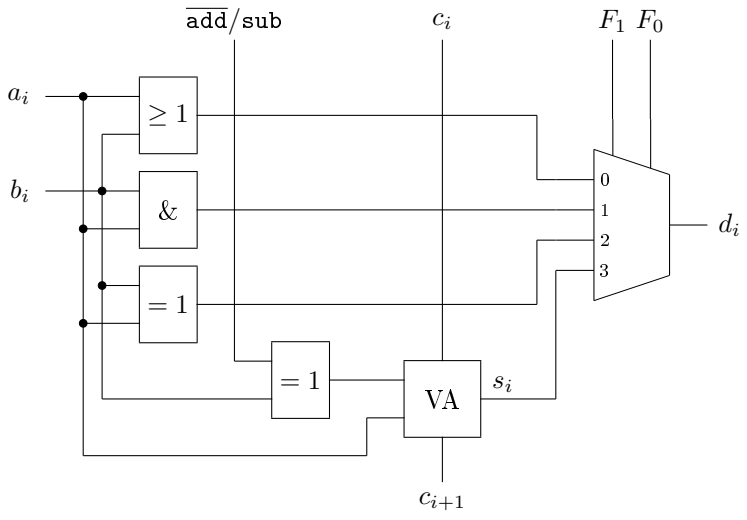
- einem Volladdierer mitsamt XOR-Verknüpfungsglied für die Komplementbildung zur Subtraktion
- einer Gruppe von Verknüpfungsgliedern für BOOLEsche Operationen
- und einem Multiplexer, der je nach Belegung seiner Selektoreingänge das Ergebnis einer bestimmten Operation auf den Ausgang durchschaltet.

Durch die Verkettung genügend vieler 1-bit-ALUs lässt sich eine ALU beliebiger Wortbreite aufbauen.

Zur Realisierung der ALU unserer Zentraleinheit genügt eine Kette von acht 1-bit-ALUs, die neben der Addition/Subtraktion die drei logischen Operationen OR, AND und XOR unterstützen. Die Auswahl der jeweiligen Operation werde über folgende Auswahlcodes getroffen:

	F_1	F_0
OR	0	0
AND	0	1
XOR	1	0
add/sub	1	1

Das entsprechende Schaltbild sieht aus wie folgt:



Die letzte noch nicht behandelte Operation ist die Rechtsrotation. Hierfür wird ein Schaltnetz eingesetzt, das als *Barrel Shifter* bezeichnet wird und das Bitfolgen in konstanter Zeit um eine beliebige Stellenanzahl verschiebt oder rotiert. Ein passender Barrel Shifter wird in einer Übungsaufgabe entwickelt und könnte zusätzlich noch in die ALU integriert werden.

2.3.5 Hauptspeicher

Der Hauptspeicher ist ein *ortsadressierbarer* Schreib-/Lese-Speicher (engl. *Random Access Memory*, RAM), d. h. auf jeden Speicherort kann *wahlfrei* zugegriffen werden (ohne dass vorher – wie bei klassischen Magnetbandspeichern – andere Speicherpositionen sequentiell durchlaufen werden müssen). Jeder Speicherort ist über seine Adresse eindeutig identifiziert und umfasst ein Wort bzw. (bei byte-adressiertem Speicher) ein Byte.

Der prinzipielle Aufbau eines RAM ist in Abbildung 2.42 wiedergegeben. Die Auswahl eines Speicherorts geschieht durch einen *1-aus-n*-Adressdecodierer, der die als Dualzahl gegebene Adresse auflöst und genau diejenige Wortleitung aktiviert, die der Adresse entspricht. Alle anderen Wortleitungen bleiben inaktiv.

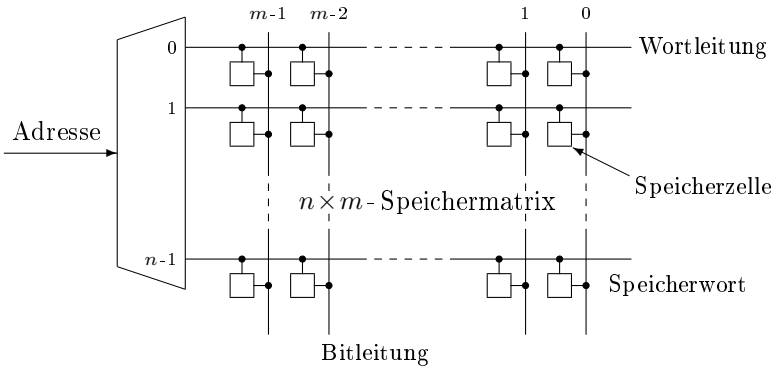


Abbildung 2.42: Prinzipieller Aufbau eines Hauptspeichers

Bei großen Speichern wird die technische Realisierung des 1-aus-n-Decodierers kritisch. Ein byteadressierter Speicher der Größe 1 MByte benötigt bereits 1 Million Wortleitungen. Es ist daher vorteilhaft, einen Speicher intern in Zeilen und Spalten zu organisieren (siehe Abbildung 2.43). Die ursprüngliche Adresse gliedert sich dann in zwei Teile, von denen einer die Zeilennummer, der andere die Spaltennummer beschreibt. In diesem Fall wären nurmehr ein Zeilen- und ein Spaltendecodierer für jeweils 1000 Wortleitungen erforderlich.

Um den Adressbus schmaler gestalten zu können, werden in manchen Architekturen die Zeilen- und die Spaltenadresse nacheinander über dieselben Adressleitungen übertragen („Adressmultiplexing“). In diesem Fall muss mittels zweier Steuersignale RAS (*Row Address Strobe*) und CAS (*Column Address Strobe*) angezeigt werden, welcher Adressteil gerade anliegt.

Schreib-/Lese-Speicher werden gemäß ihrer technischen Umsetzung in zwei Gruppen eingeteilt:

- **statische** RAM-Bausteine (SRAM) speichern die Information in Zellen, die aus einer Flipflop-Grundschaltung bestehen,
- **dynamische** RAM-Bausteine (DRAM) speichern die Information als elektrische Ladung in einem Kondensator. Das Lesen einer Speicherzelle bedingt in der Regel das Entladen des Kondensators („zerstörendes Lesen“), so dass danach der gelesene Wert wieder geschrieben

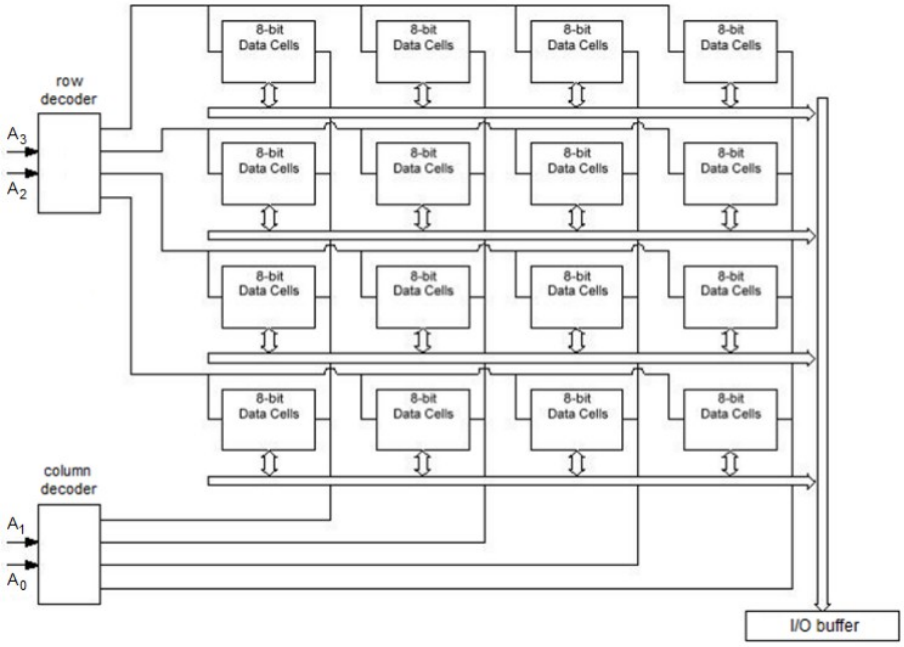


Abbildung 2.43: Speicherorganisation in Zeilen und Spalten

werden muss. Außerdem geht durch unvermeidliche Leckströme kontinuierlich Ladung verloren, weshalb DRAM-Zellen in regelmäßigen Abständen wieder aufgefrischt werden müssen (*Refresh*). Ein Refresh-Zyklus ist etwa alle 8 bis 64 ms erforderlich. Obgleich das häufig erscheint, ist ein DRAM nur rund 1% seiner Betriebszeit mit Refreshing beschäftigt.

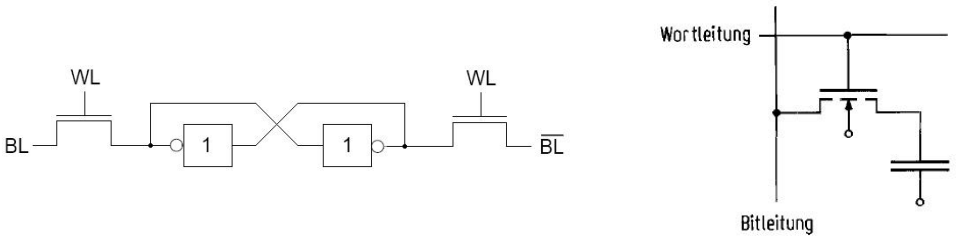


Abbildung 2.44: SRAM- und DRAM-Zelle

Man erkennt in Abbildung 2.44, dass eine SRAM-Zelle sechs Transistoren benötigt, während eine DRAM-Zelle mit einem Transistor und einem Kondensator auskommt. Beide werden auf einem Halbleiterchip sogar gemeinsam als 1-Transistor-Zelle realisiert (siehe Abbildung 2.45). Im Vergleich zu einer SRAM-Zelle benötigt die DRAM-Zelle nur $\frac{1}{4}$ der Fläche und erlaubt somit höhere Packungsdichten. Durch die vergleichsweise langsamen Ladevorgänge des Kondensators und das notwendige Refreshing ist ein DRAM jedoch langsamer. Zudem ist sein Energieverbrauch (und damit die Abwärme) höher.

Typischerweise verwendet man die SRAM-Technik heute zur Realisierung von *Cache-Speichern* (siehe Abschnitt 2.4.1) während Hauptspeicher in DRAM-Technik gebaut werden.

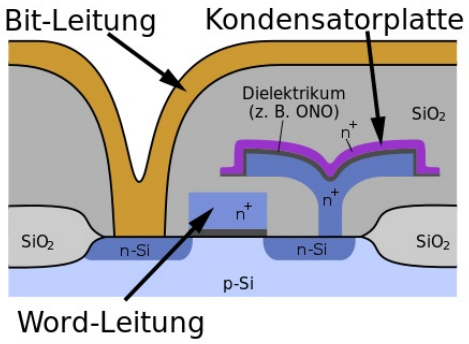


Abbildung 2.45: Aufbau einer DRAM-Zelle in 3D-Stack-Technologie [12]

Entwicklungsstufen der Speichertechnologie

Beide Techniken, SRAM und DRAM arbeiten grundsätzlich asynchron, d. h. sie führen ihre Operationen (wie ein Schaltnetz) in dem Moment aus, in dem die relevanten Steuerleitungen entsprechend gesetzt werden. Eine Synchronisation auf den Systemtakt erfolgt durch die Pufferung von Adresseingängen, Steuersignalen und Ein-/Ausgabedaten in getakteten Registern.

Unter der Bezeichnung **SDRAM** (*synchronous* DRAM) hat sich eine DRAM-Variante etabliert, die – unter Ausnutzung der zeitlichen Regelmäßigkeit einer synchronen Schnittstelle – imstande ist, ein komplexeres Verhaltensmuster zu unterstützen. Wegen der Rückschreibenotwendigkeit erfordert jeder Lesevorgang eigentlich zwei Taktzyklen. Ein SDRAM-Speicher wird so gebaut, dass er zwei sog. *Speicherbänke* gleichzeitig benutzt. Der Adressraum wird so auf die beiden Speicherbänke abgebildet, dass aufeinanderfolgende Speicherworte in jeweils der anderen Speicherbank liegen. Wird nun ein ganzer Speicherblock („*burst*“) auf einmal ausgelesen, kann mit *jedem* Taktzyklus das jeweils nächste Speicherwort bereitgestellt werden – abwechselnd aus je einer Speicherbank, während in der anderen das Rückschreiben stattfindet. Dass es sinnvoll ist, gleich einen ganzen Block von Daten bereitzustellen, wird im Zusammenhang mit Cache-Speichern noch deutlich werden (siehe Abschnitt 2.4.1).

Obgleich das Verschränken von zwei Speicherbänken ausreichend wäre, wird heute oft eine höhere Anzahl von Bänken verwendet (4, 8 oder 16), um die Wahrscheinlichkeit eines Bankwechsels bei wahlfreier (zufälliger) Adressierung auch für kurze Burstlängen zu erhöhen.

Eine weitere Variante ist der **DDR-SDRAM** (*double data rate* SDRAM), der jeweils mit der steigenden *und* mit der fallenden Taktflanke die nächsten Speicherwörter liefert, also *zwei* pro Taktzyklus. Dies wird realisiert, indem ein interner Ein-/Ausgabepuffer verwendet wird, der *doppelt* so breit ist wie die Datenbusbreite und erlaubt, mit einem einzigen Lesezugriff gleich die 2-fache Datenmenge vorweg aus dem Speicher bereitzustellen. Man bezeichnet diese Technik als *2-fach Prefetch*. Bei einem Schreibzugriff müssen umgekehrt zwei Datenbustransfers vorweg erfolgt sein, um den internen Puffer zu belegen.

Die Weiterentwicklung ist der **DDR2-SDRAM**. Bei dieser Variante werden die jeweils nächsten Daten ebenfalls mit beiden Taktflanken bereitgestellt, allerdings kann der Ein-/Ausgabepuffer mit einer doppelt so schnellen (externen) Taktrate gelesen werden, wie er mit der (internen) Taktrate gefüllt wird. Dies erfordert bereits einen 4-fach Prefetch.

Schließlich sind DDR3- und (seit 2014) DDR4-SDRAMs die direkte Fortentwicklung der DDR2-Technologie. Sie unterstützen die vier- bzw. achtfache (externe) Taktfrequenz und bedingen einen 8-fach bzw. (eigentlich) einen 16-fach Prefetch. Durch Optimierungen des inneren Aufbaus[39] genügt beim DDR4 jedoch weiterhin ein 8-fach Prefetch.

[39] DDR4 erreicht schnellere Zugriffszeiten u. a. durch eine reduzierte Betriebsspannung von 1.2 V, die auf Wortlei-

2.3.6 Ein-/Ausgabewerke

Die E/A-Werke bilden die Schnittstelle zur Peripherie. Aufgrund der spezifischen Signale und Übertragungsabläufe können Peripheriesysteme nicht direkt an den Systembus angeschlossen werden, sondern bedürfen individueller *Anpassungsbausteine* (engl. *peripheral interface adapter, I/O controller*).

Der prinzipielle Aufbau eines einzelnen E/A-Werks ist im Gesamtbild des VON NEUMANN-Rechners in Abbildung 2.24 dargestellt. Ein *Datenregister* (DR) dient der Pufferung der Ein-/Ausgabedaten zur Anpassung der unterschiedlichen Arbeitsgeschwindigkeiten von Prozessor und Hauptspeicher einerseits und dem angeschlossenen Peripheriegerät andererseits. Es unterstützt aber auch eine evtl. notwendige Umsetzung paralleler Daten in serielle Daten oder digitaler Daten in analoge Daten. Ein Steuer/Status-Register (CSR, *Control/Status Register*) ermöglicht es, unterschiedliche Betriebsarten des Peripheriegeräts auszuwählen und dessen augenblicklichen Betriebszustand anzuzeigen.

Das Ansprechen der Register DR und CSR eines E/A-Werks erfolgt (alternativ) auf eine von zwei Arten:

- im Falle der sog. **Port-based I/O** wird mit jedem E/A-Register eine *Portnummer* assoziiert. Über spezielle Maschinenbefehle erfolgt ein Lese- oder Schreibzugriff auf einen bestimmten Port.

  Der Intel-Befehlssatz enthält hierfür beispielsweise folgende Befehle:

Befehle zur Ein-/Ausgabe		
`in`	`al,port`	liest ein `BYTE` vom angegebenen Port in `al`
`out`	`port,ax`	schreibt ein `WORD` aus `ax` in den angegebenen Port

- bei der sog. **Memory-mapped I/O** werden die E/A-Register in den Hauptspeicher-Adressraum abgebildet. Erscheint eine solche „umgewidmete“ Hauptspeicheradresse auf dem Adressbus, wird sie vom Speicherwerk ignoriert. Stattdessen reagiert die Steuerung des entsprechenden E/A-Werks. Ein Zugriff auf die Register kann ohne spezielle Maschinenbefehle genauso wie ein Hauptspeicherzugriff programmiert werden.

Der Zugriff auf Ein- und Ausgabedaten kann auf verschiedene Arten gesteuert werden. Man unterscheidet drei wesentliche Methoden:

- *programmgesteuerte* Ein-/Ausgabe,
- *unterbrechungsgesteuerte* Ein-/Ausgabe und
- die Methode des *direkten Speicherzugriffs* (engl. *Direct Memory Access*, DMA).

Bei der **programmgesteuerten** E/A wird die Datenübertragung durch die Ausführung von Schreib-/Lesezugriffen auf das DR des Bausteins an geeigneten Stellen im Programm vorgesehen. Dazu fragt das Programm in regelmäßigen oder unregelmäßigen Abständen das CSR daraufhin ab, ob der Schnittstellenbaustein bereit ist, ein Datum zu übertragen (diese Vorgehensweise wird auch als *Polling* bezeichnet).

tungen durch Ladungspumpen wieder verstärkt wird, und durch eine Verkürzung der Wortbreite pro Speicherbank, was im Gegenzug allerdings die Zahl der Speicherbänke erhöht. Durch besondere Maßnahmen zur Verringerung der Leistungsaufnahme (bspw. temperaturabhängige Bestimmung des Refreshing-Intervalls, invertierte Abspeicherung von Daten, wenn dadurch Ladungswechsel reduziert werden) kann die Temperatur gering gehalten werden. Dies ist wichtig, da MOS-Kapazitäten mit der Temperatur ansteigen und Schaltzeiten mit der Kapazität wachsen.

Beispiel 2.23 Folgender Assemblerprogrammausschnitt beschreibt die Abfrage des Tastaturcontrollers eines Standard-PC. Das DR wird über Port `60h`, das CSR über Port `64h` angesprochen. Das CSR fasst mehrere Flags zusammen. An der niederwertigsten Bitposition des CSR befindet sich das OUTB-Flag, das genau dann gesetzt ist, wenn ein neues Ausgabe-Byte im DR bereit steht.

```
kbRead:   in    al, 64h    ; read status byte
          test  al, 1      ; test OUTB flag
          jz    kbRead     ; wait for OUTB = 1
          in    al, 60h    ; read data byte
          ...
```
□

Offenbar führt das Programm hier eine Schleife aus, in der lediglich das CSR abgefragt wird, bis schließlich neue Daten vorliegen. Über diese Abfrage hinaus tut die CPU nichts anderes. Man spricht von „aktivem Warten“ oder *busy waiting*. So wird zwar eine schnelle Reaktion auf einen erwarteten Transfer erreicht, aber man verbraucht Rechenleistung für die Wartephase. Dieses Verfahren ist nur dann gerechtfertigt, wenn der Rechner seine Aufgabe ohne das erwartete Datum nicht weiter ausführen kann.

Im Gegensatz dazu wird bei der **unterbrechungsgesteuerten** E/A jede Datenübertragung vom E/A-Werk bei der CPU über ein Steuersignal `IRQ` (*interrupt request*) angefordert. Das Leitwerk unterbricht daraufhin zum nächstmöglichen Zeitpunkt (also nach vollständiger Ausführung des aktuellen Befehls) sein augenblicklich bearbeitetes Programm und springt in ein spezielles Unterprogramm zur Unterbrechungsbehandlung (*interrupt service routine*, ISR).

Das jeweils anzuwendende Unterprogramm wird der CPU vom E/A-Werk durch die sog. IVN (*Interrupt Vector Number*) zur Verfügung gestellt[40]. Die IVN ist bspw. über bestimmte Bits im CSR auslesbar. Sie entspricht einem Index in den *Interruptvektor*.

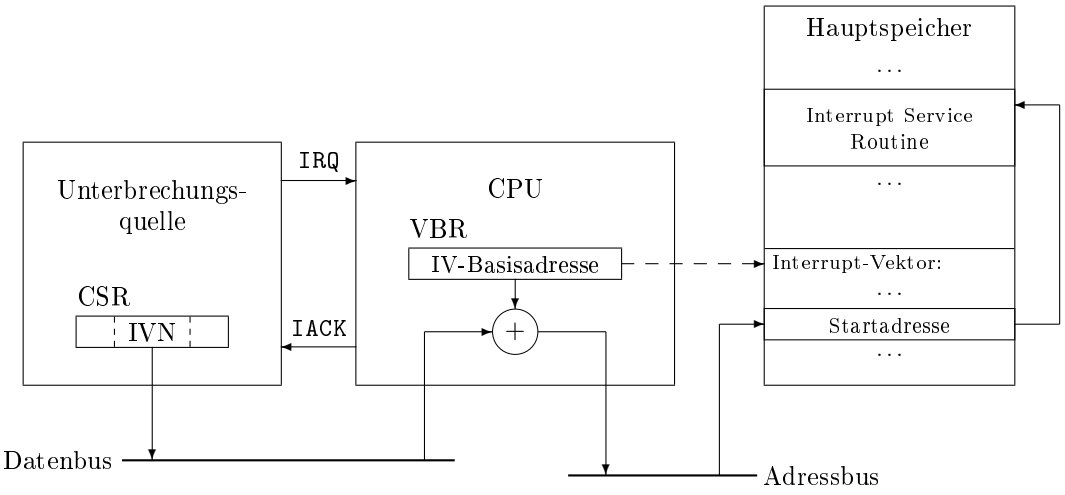


Abbildung 2.46: Berechnung der Einsprungadresse von Unterbrechungsprogrammen

Im Interruptvektor werden die Startadressen sämtlicher Unterbrechungsprogramme als Tabelle im Hauptspeicher (residenter Teil) zusammengefasst. Die Adresse, an der diese Vektortabelle beginnt, ist die Interruptvektor-Basisadresse. Sie wird in einem Basisadressregister VBR (*Vector Base Register*) gespeichert (siehe Abbildung 2.46).

[40] Dies kann bspw. erfolgen, indem von der CPU eine Steuerleitung `IACK` (*interrupt acknowledge*) aktiviert wird. Sie wird durch alle E/A-Werke durchgeführt. Das erste E/A-Werk in dieser Kette, das einen `IRQ` ausgelöst hat, schaltet mit der nächsten steigenden Taktflanke die relevante IVN auf den Datenbus.

Die Ausführung der ISR ist sehr ähnlich zur Ausführung eines Unterprogramms. Zu Beginn werden der (unterbrochene) Befehlszähler, das Statusregister und alle Rechenregister auf dem Stack gesichert. Am Ende – mit dem abschließenden `IRET`-Befehl – wird der Zustand vor der Unterbrechung durch Rückspeicherung vom Stack wieder hergestellt.

In den meisten modernen Rechnersystemen wird der Verwaltungsaufwand für Unterbrechungen von der CPU in einen separaten *Interrupt-Controller* verlagert. Aus Sicht der CPU gibt es dann nur noch diese einzige Interruptquelle. Der Interrupt-Controller selbst besitzt mehrere Eingänge für die einzelnen E/A-Werke. Bei Bedarf können mehrere Interrupt-Controller kaskadiert werden. Der Interrupt-Controller leistet ggf. eine Priorisierung bei mehrfachen Unterbrechungen bzw. erlaubt höher priorisierten Unterbrechungen, die laufende Verarbeitung niedriger priorisierter Unterbrechungen zu unterbrechen. Meist kann ein Interrupt-Controller (vom Systemprogrammierer) so programmiert werden, dass einzelne Unterbrechungen *maskiert*, d. h. (bis zur Aufhebung ihrer Maskierung) unterdrückt werden können.

Der Vorteil der unterbrechungsgesteuerten E/A besteht darin, dass die CPU zwischen den einzelnen Unterbrechungsbehandlungen frei ist, andere Operationen auszuführen, was insbesondere bei der E/A mit langsamen Peripheriegeräten (z. B. Tastaturen) vorteilhaft ist. Der Datentransfer selbst wird jedoch im Rahmen der Unterbrechungsbehandlung weiterhin über die CPU gesteuert.

Die dritte Methode, der **Speicherdirektzugriff** (DMA) befreit die CPU vollständig vom Datentransfer. Er wird stattdessen von einem speziellen Steuerbaustein abgewickelt, der als *DMA-Controller* (DMA-C) bezeichnet wird. Ein DMA-C kann ein oder mehrere E/A-Werke steuern.

Typischerweise arbeitet ein DMA-C im sog. Blockmodus (engl. *burst mode*), in dem jeweils ein ganzer Block von Daten auf einmal übertragen wird. Zu Beginn eines Datentransfers wird der DMA-C von der CPU „programmiert“, d. h. mit der Startadresse eines Datenbereichs im Hauptspeicher, der Anzahl der zu übertragenden Bytes und der Transferrichtung (Lesen/Schreiben) versorgt. Danach wickelt der DMA-Controller den gesamten Datentransfer selbständig über den Systembus ab (vgl. Abbildung 2.47). Der Bus ist ihm exklusiv zugeteilt.

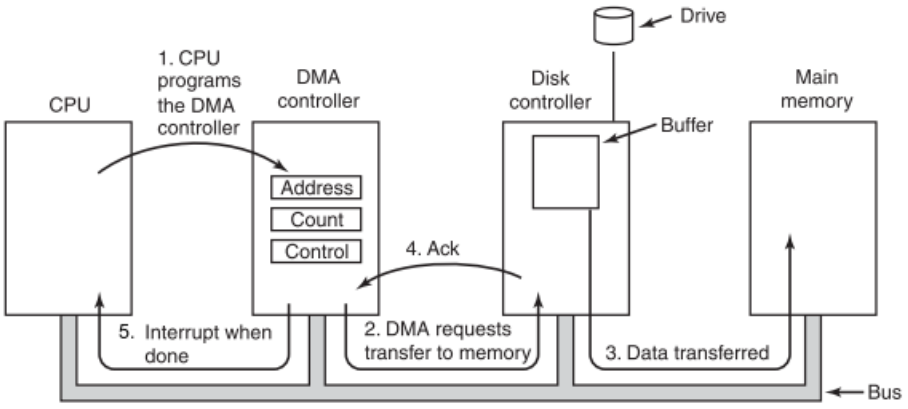


Abbildung 2.47: Ablauf eines DMA-Transfers

Ein DMA-C entlastet die CPU zwar vom Datentransfer, nicht aber von der Gerätesteuerung. Auf Ausnahmesituationen in den Peripheriegeräten muss immer noch die CPU reagieren.

2.4 Fortgeschrittene Konzepte in heutigen Rechnerarchitekturen

2.4.1 Cache-Speicher

In der Pionierzeit war die CPU die langsamste Einheit des Rechners. Bei heutigen Rechnern kann die CPU inzwischen Daten und Befehle deutlich schneller verarbeiten, als sie aus dem (vergleichsweise langsamen) Hauptspeicher über den (vergleichsweise langsamen) Systembus bereit gestellt werden können. Dabei ist es durchaus kritisch, dass Daten und Befehle sich denselben Bus teilen[41]. Hauptspeicher und Bussystem bestimmen somit die Verarbeitungsgeschwindigkeit des gesamten Rechners. Dieses Problem wird als VON-NEUMANN-*Flaschenhals* bezeichnet. Zur Abschwächung des VON-NEUMANN-Flaschenhalses verwenden heutige Rechnerarchitekturen meist sog. *Cache-Speicher*.

Für die meisten Anwendungen gilt die sog. *Lokalitätseigenschaft* (*zeitliche* Lokalität: wird einmal auf eine Speicheradresse zugegriffen, so wird mit hoher Wahrscheinlichkeit innerhalb kurzer Zeit erneut darauf zugegriffen[42], *räumliche* Lokalität: wird einmal auf eine Speicheradresse zugegriffen, so wird mit hoher Wahrscheinlichkeit innerhalb kurzer Zeit auch auf benachbarte Speicheradressen zugegriffen).

Dies rechtfertigt die Idee, einen kleinen Auszug aus dem (billigen, langsamen DRAM-) Hauptspeicher in einem (schnellen, teuren SRAM-) *Pufferspeicher* oder *Cache*-Speicher[43] zur Verfügung zu stellen. Dieser Speicher, der logisch zwischen der Zentraleinheit und dem Hauptspeicher liegt, wird meist *prozessornah*, d. h. auf dem CPU-Chip untergebracht.

Aufbau und Funktionsweise eines Caches

Ein Cache ist in Zeilen organisiert, die im Grundaufbau aus zwei Elementen bestehen: der Hauptspeicheradresse eines Datenworts und dem Datenwort selbst (vgl. Abbildung 2.48). Zusätzlich besitzt jede Zeile ein Flag V – das *Valid*-Bit – das anzeigt, ob in der jeweiligen Zeile ein gültiger Eintrag steht.

Bei jedem **Lesezugriff** der CPU wird immer zuerst versucht, aus dem Cache zu lesen. Über einen Adressvergleich mit den Adressfeldern aller gültigen Zeilen wird geprüft, ob das adressierte Datenwort im Cache vorhanden ist. Falls ja, spricht man von einem Treffer (engl. *hit*) und das Datenwort wird vom Cache bereitgestellt. Im Fall eines Fehlschlags (engl. *miss*) muss ein Zugriff auf den Hauptspeicher erfolgen. Das dort gelesene Datenwort wird üblicherweise (dem Lokalitätsprinzip entsprechend) auch gleich in den Cache kopiert.

Bei einem **Schreibzugriff** wird ebenfalls zuerst der Cache geprüft. Im Fall eines *miss* muss das Datenwort in den Hauptspeicher geschrieben werden. Geschieht dies ohne Einbeziehen des Caches, so spricht man von einem *write-around* (oder auch *write-no-allocate*). Wird (dem Lokalitätsprinzip entsprechend) außerdem auch eine Kopie im Cache angelegt, bezeichnet man dies als *write-allocate*.

Im Fall eines *hit* muss das Datenwort, um die Datenkonsistenz (*Kohärenz*) zu erhalten, sowohl im Cache als auch im Hauptspeicher geschrieben werden. Auch hierbei gibt es zwei Möglichkeiten:

- Beim *Durchschreibeverfahren* (engl. *write-through*) wird jeder Schreibvorgang auf beiden Speichern durchgeführt. Damit ist die Kohärenz stets gewährleistet, der Vorteil des schnellen Zugriffs über den Cache geht jedoch verloren, da in jedem Fall auf den Hauptspeicher gewartet werden muss.

[41] Es gibt andere Rechnerarchitekturen, bspw. die *Harvard-Architektur*, bei der für Daten und Befehle getrennte Busse und Speicher verwendet werden.

[42] Messungen zeigen, dass im Mittel ca. 10% des Programmcodes für ca. 90% der Laufzeit verantwortlich ist.

[43] *Cache* = (dt.) verstecktes Lager

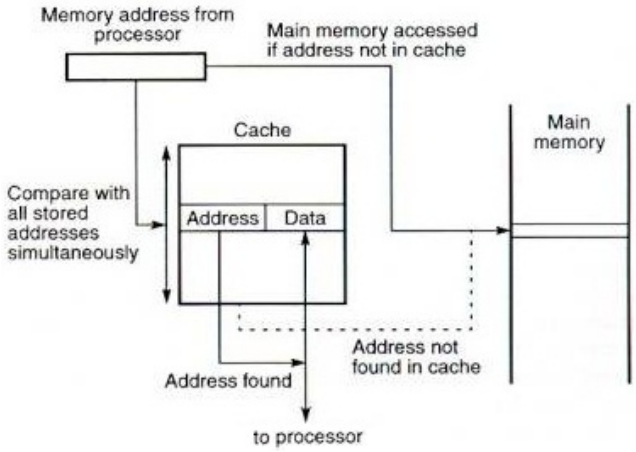


Abbildung 2.48: Prinzipieller Aufbau eines Cache-Speichers (nach [13])

- Beim *Rückschreibeverfahren* (engl. *write-back*) wird das Datenwort zunächst nur in den Cache geschrieben und durch ein Flag D – das *Dirty*-Bit – als geändert gekennzeichnet. Erst zu einem späteren Zeitpunkt wird das Datenwort in den Hauptspeicher zurückgeschrieben. Diese Vorgehensweise ist dann problematisch, wenn andere Rechnerkomponenten parallel auf den Hauptspeicher zugreifen können. Hier bedarf es zusätzlicher *Cache-Kohärenz-Protokolle*, die im folgenden Abschnitt 2.4.2 angesprochen werden.

Es hat sich gezeigt, dass bei den meisten Rechneranwendungen die kombinierten Strategien *write-allocate*/*write-back* die beste Effizienz bieten. Beide Strategien setzen auf das Lokalitätsprinzip und gehen davon aus, dass weitere Zugriffe auf das zu schreibende Datenwort erfolgen werden.

Reale Caches speichern in einer Zeile üblicherweise nicht nur ein einziges Datenwort ab, sondern Blöcke aufeinanderfolgender Datenwörter, die auch als Einheit vom/zum Hauptspeicher übertragen werden. Ein Block wird als *Cache line* bezeichnet und hat eine typische Größe von 32 bis 256 Bytes (vgl. Abbildung 2.49).

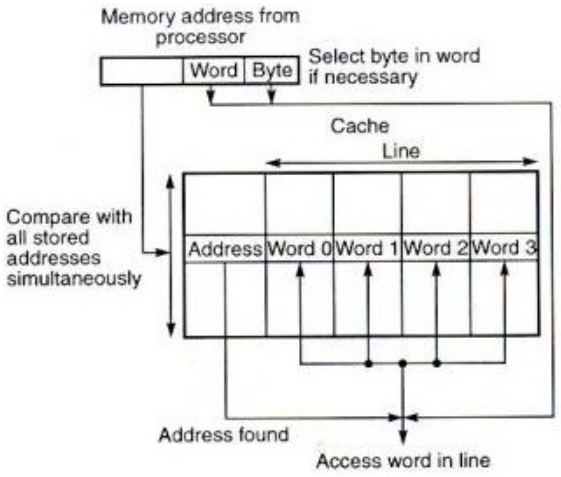


Abbildung 2.49: Cache line (aus [13])

Organisation eines Caches

Da ein Cache nur einen Auszug aus dem Hauptspeicher repräsentieren kann, ist es von großer Bedeutung, eine günstige Abbildungsfunktion vom Hauptspeicher auf den Cache zu finden, d. h. festzulegen, wo im Cache welcher Hauptspeicherblock abgelegt wird.

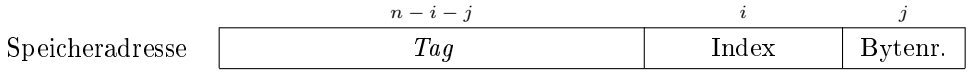
Die in Abbildung 2.48 implizierte Abbildungsfunktion bietet die Möglichkeit, jeden Hauptspeicherblock in jeder beliebigen Zeile im Cache zu speichern. Diese Organisationsform bezeichnet man als **vollassoziative Abbildung** (engl. *fully associative cache*). Sie erfordert einen hohen Suchaufwand, da bei jedem Zugriff alle Adressen überprüft werden müssen. Sie bietet aber die größte Flexibilität, da der Cache stets Daten aufnehmen kann, solange noch irgendwo eine Zeile frei ist. Sind alle Zeilen belegt, muss mit dem nächsten *miss* eine *Verdrängungsstrategie* angewendet werden. Typische Strategien sind:

- *Round Robin* (RR), ersetzt in fester (zyklischer) Reihenfolge,
- *Least Recently Used* (LRU), ersetzt diejenige Zeile, auf die am längsten nicht mehr zugegriffen wurde.

Bei Einsatz des Rückschreibeverfahrens ist vor der Verdrängung einer Cache-Line das D-Bit zu prüfen und ggf. ein Rückschreiben in den Hauptspeicher durchzuführen.

Eine zweite Abbildungsfunktion ist die **Direktabbildung** (engl. *direct mapping*). Sie stellt das umgekehrte Extrem dar: jeder Hauptspeicherblock kann nur in genau einer einzigen möglichen Zeile im Cache abgelegt werden. Der Index dieser Zeile errechnet sich direkt aus der Hauptspeicheradresse.

Geht man von einem Cache aus, der 2^i Zeilen Raum bietet und der eine Blockgröße (*line size*) von 2^j Bytes unterstützt, so werden von der Original-Speicheradresse A (n-stellig, $n > i + j$) die i Stellen $A_{i+j-1}, \ldots, A_j$ direkt als Index für den Cache benutzt.



Natürlich gibt es nun mehrere Hauptspeicherblöcke, die denselben Index haben und somit in dieselbe Cache-Line gehören. Um erkennen zu können, welcher dieser Hauptspeicherblöcke tatsächlich gerade im Cache gespeichert wird, genügt es, die höchstwertigen $n - i - j$ Stellen $A_{n-1}, \ldots, A_{i+j}$ als „Etikett“ (engl. *tag*) im Adressfeld abzulegen.

Sobald ein Hauptspeicherblock in den Cache geholt wird, dessen Index identisch mit einem bereits im Cache befindlichen Block ist, verdrängt er den älteren Block.

Mit dieser Abbildungsfunktion wird der Hardwareaufwand für den Tag-Vergleicher minimiert. Andererseits ist die Ausnutzung des Caches u. U. nicht befriedigend, da Cachezeilen frei bleiben, wenn keine Zugriffe über ihre Indizes stattfinden. Umgekehrt kann es zu sogenanntem *Flattern* kommen (engl. *cache thrashing*), wenn auf zwei (oder mehr) Datenblöcke mit gleichem Index ein häufig wechselnder Zugriff erfolgt. Eine solche Situation ist aufgrund des Lokalitätsprinzips zwar nicht zu erwarten, aber möglich.

Einen Mittelweg beschreitet die **n-fach satz-assoziative Abbildung** (engl. *n-way set associative mapping*). Sie unterteilt den Cache in n Teilcaches, von denen jeder einzelne nach dem Direktabbildungsprinzip arbeitet. Somit können n Hauptspeicherblöcke mit demselben Index aber unterschiedlichen Tags im Cache gespeichert werden. Typische Werte für n sind 2, 4 oder 8. Eine Auswahllogik entscheidet, in welchen dieser Teilcaches im Falle eines *miss* zu scheiben ist. Diese Logik arbeitet üblicherweise ebenfalls mit den oben bereits erwähnten Verdrängungsstrategien RR und LRU.

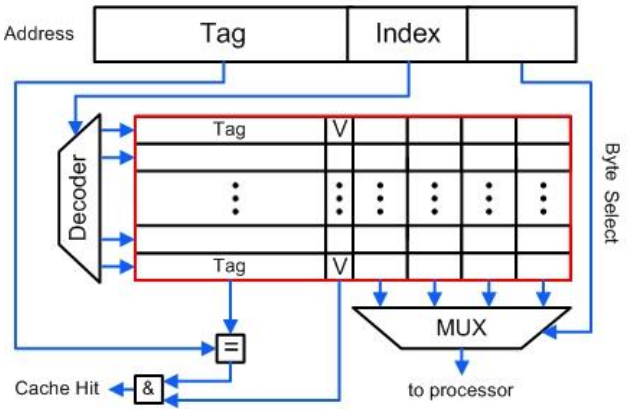


Abbildung 2.50: Aufbau eines direkt abbildenden Caches

Mehrstufiges Caching

Bei der Dimensionierung eines Caches gilt es, einen guten Kompromiss zwischen kurzer Zugriffszeit und hoher Trefferrate zu finden: größere Caches haben günstigere Trefferraten, aber eine längere Latenz. Um diesen Kompromiss zu verbessern, benutzen heutige Rechner *mehrere Ebenen* von Caches. Häufig wird ein kleiner, schnellerer Cache – der *Primärcache* oder **Level-1**-Cache (L1) – durch einen größeren, langsameren Cache – den *Sekundärcache* oder **Level-2**-Cache (L2) – bedient.

Insbesondere in Mehrprozessorsystemen und Mehrkern-Architekturen wird sogar eine dritte Ebene (L3) verwendet, die vor allem der Sicherstellung der Cache-Kohärenz dient, d. h. dem Abgleich der Inhalte der individuellen Caches aller Prozessoren bzw. Kerne im Fall von Schreiboperationen (siehe auch Abschnitt 2.4.2).

Abbildung 2.51 zeigt typische Speichergrößen und Antwortzeiten für die unterschiedlichen Cache-Ebenen im Kontext einer typischen *Speicherhierarchie*. Die Speicherhierarchie veranschaulicht die Vielfalt der in einer Rechnerarchitektur verwendeten Speicherformen, an der Spitze die CPU-Register (wenig Kapazität, sehr schnell, sehr teuer), bis hinunter zum Offline-Massenspeicher (immense Kapazität, langsam, preiswert).

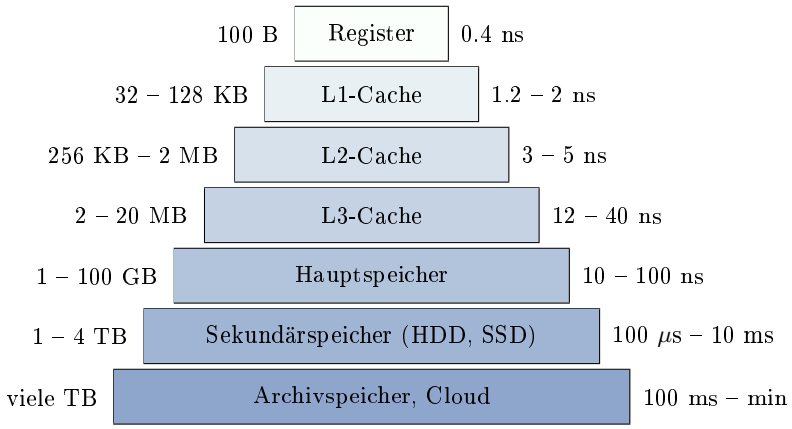


Abbildung 2.51: Speicherpyramide mit typischen Kapazitäten und Zugriffszeiten

2.4.2 Mehrkern-Architekturen

Bis etwa ins Jahr 2005 wurde die stetige Steigerung der Rechnerleistung im Wesentlichen durch zunehmend komplexere Transistorstrukturen auf dem Prozessorchip[44] und eine regelmäßige Erhöhung der Taktfrequenz erreicht. Mit der Taktrate stieg auch die elektrische Leistungsaufnahme P an. Bei CMOS-Schaltkreisen gilt

$$P \sim C \cdot U_V^2 \cdot f$$

wobei C die pro Schaltvorgang umgeladene Kapazität (über alle Transistoren des Chips gerechnet), U_V die Betriebsspannung und f die Taktfrequenz bezeichnet. Die aufgenommene Leistung wird bei einem Chip vollständig in Abwärme umgewandelt. Die Taktsteigerung konnte durch immer größere Kühlkörper, stärkere Ventilatoren und ein Absenken der Betriebsspannung lange aufgefangen werden. Bei einer Taktrate von etwa 4 GHz erreichten diese Maßnahmen aber ihre Grenzen. Als Lösung setzte sich die *Mehrkern-Architektur* durch.

Ein Mehrkernrechner besitzt mehrere Zentraleinheiten („Kerne“, engl. *cores*) auf einem *einzigen* Prozessor-Chip[45].

Die Vorteile verdeutlicht Abbildung 2.52, die auf *Intel*-Angaben basiert [14]. Ein Zweikernprozessor kann gegenüber einem Einkernprozessors mit jeweils um 15% reduzierter Taktrate und Betriebsspannung betrieben werden, hat dadurch die gleiche Leistungsaufnahme, steigert aber die Rechenleistung (fast) im selben Maße wie die Chipfläche erhöht wird.

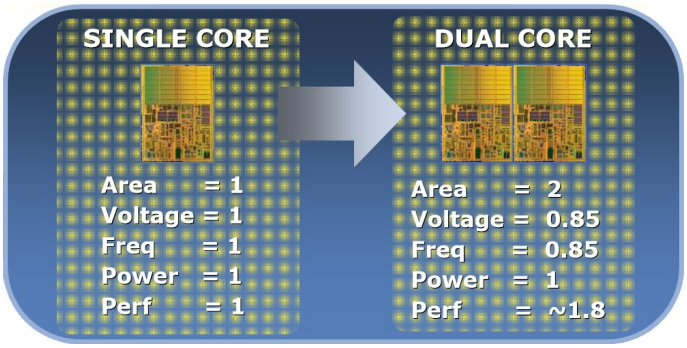


Abbildung 2.52: Vergleich Einkern-/Zweikernprozessor

Die Kommunikation der Kerne erfolgt üblicherweise über einen *gemeinsamen Cache-Speicher*. Ein typisches Modell einer Cache-Struktur in Mehrkern-Architekturen zeigt Abbildung 2.53.

Während jeder Kern über seinen privaten L1-Cache verfügt, um schnellsten Zugriff auf häufig benutzte Daten und Befehle zu haben, dienen die L2-Caches als gemeinsamer Speicher für Gruppen von Kernen. Schließlich sorgt ein allen Kernen gemeinsamer L3-Cache dafür, dass die Anzahl der Zugriffe auf den Hauptspeicher möglichst gering bleibt.

Eine zentrale Aufgabe ist die Gewährleistung der Cache-Kohärenz. Man verwendet *Cache-Kohärenz-Protokolle*, die dafür sorgen, dass jeder Cache beim Zugriff auf ein Datum stets den (über alle Kerne betrachtet) zuletzt geschriebenen Wert liefert. Ein bekanntes Protokoll ist das MESI-Protokoll, das hier aber nicht weiter vertieft werden soll.

Mehrkernarchitekturen stellen den derzeit populärsten Ansatz dar, die stetig wachsenden Bedarfe für höhere und höchste Rechenleistungen zu erfüllen. Zukünftig sollen auf einem einzigen

[44] Das Moore sche Gesetz besagt, dass sich aufgrund der technologischen Entwicklung die Integrationsdichte, also die Anzahl der Transistoren pro Flächeneinheit auf einem integrierten Schaltkreis, alle 18 Monate verdoppelt.

[45] Hierin liegt der Unterschied zu einem *Mehrprozessorrechner*, der *mehrere* Chips verwendet (jeweils Einkern- oder Mehrkern-Prozessorchips).

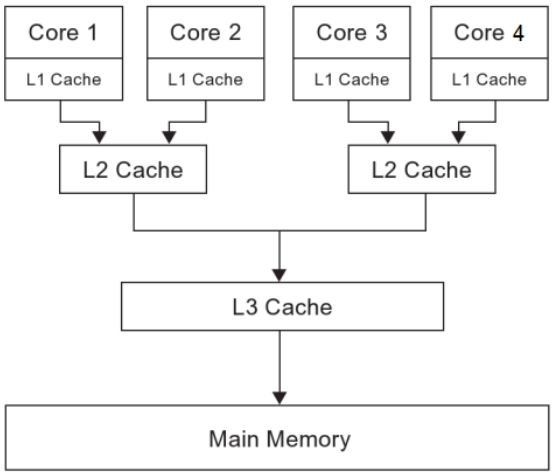


Abbildung 2.53: Cache-Struktur in Mehrkernarchitekturen

Chip Hunderte Kerne untergebracht werden. Man spricht bei solch hohen Zahlen von **Manycore**-Prozessoren.

Ihr volles Potential entfalten Mehrkernarchitekturen allerdings nur dann, wenn die Software die von der Hardware zur Verfügung gestellte Parallelität auch nutzt. Für Einkernrechner entwickelte Software wird auf einem Mehrkernrechner mit hoher Wahrscheinlichkeit langsamer laufen als zuvor, weil nur ein einziger Kern genutzt wird, der im Vergleich niedriger getaktet ist. *Parallele Programmierung wird zur zwingenden Maßgabe.*

Manycore-Prozessoren sind deswegen auch weniger für den Verbrauchermarkt gedacht, sondern eher für hochgradig parallele Anwendungen wie bspw. Simulationen von physikalischen, chemischen, molekularbiologischen und meteorologischen Modellen oder Analysen von sehr großen, oft unstrukturierten Datenmengen aus verschiedenen Quellen (*Big Data*).

2.4.3 Befehlspipelining

Befehlspipelining ist eine Technik zur Beschleunigung der Befehlsausführung, die heute in nahezu allen Rechnern verwendet wird. Die (einfache) Idee besteht darin, die fünf Phasen eines Befehlszyklus, die für jeden Befehl immer wieder in der gleichen Weise durchlaufen werden (vgl. Abschnitt 2.3.1), wie bei einer Fließbandfertigung zeitlich überlappend auszuführen. Der dadurch erzielbare Beschleunigungseffekt wird in Abbildung 2.54 schnell deutlich.

Eine CPU arbeitet dadurch zur selben Zeit an mehr als einem Befehl, ohne dass eine Vervielfachung von Werken notwendig wird (wie es etwa bei Mehrprozessorsystemen der Fall ist).Wichtig ist, dass alle fünf Phasen gleich lange dauern. Dies wird über einen speziellen *Phasentakt* erreicht.

Bei gefüllter Pipeline wird mit jedem Phasentaktzyklus ein Befehl fertiggestellt. Die Wirkung ist so, als ob jeder Befehl nur einen einzigen Phasentaktzyklus zur Ausführung benötigen würde, obgleich die Ausführungszeit eines Befehls insgesamt[46] nicht reduziert wird.

Der Leistungsgewinn durch Pipelining kann wie folgt ermittelt werden: bezeichne k die Anzahl der Stufen einer Pipeline (bzw. die Anzahl der Phasen einer Befehlsausführung) und τ die Zykluszeit des Phasentakts, so beträgt die Gesamtzeit für die Ausführung von n Befehlen

$$T = (k + n - 1)\ \tau.$$

[46] Verglichen mit der Ausführungszeit ohne Pipelining ist die über alle Pipelinestufen kumulierte Befehlsausführungszeit aufgrund der Angleichung der Phasen tatsächlich etwas länger.

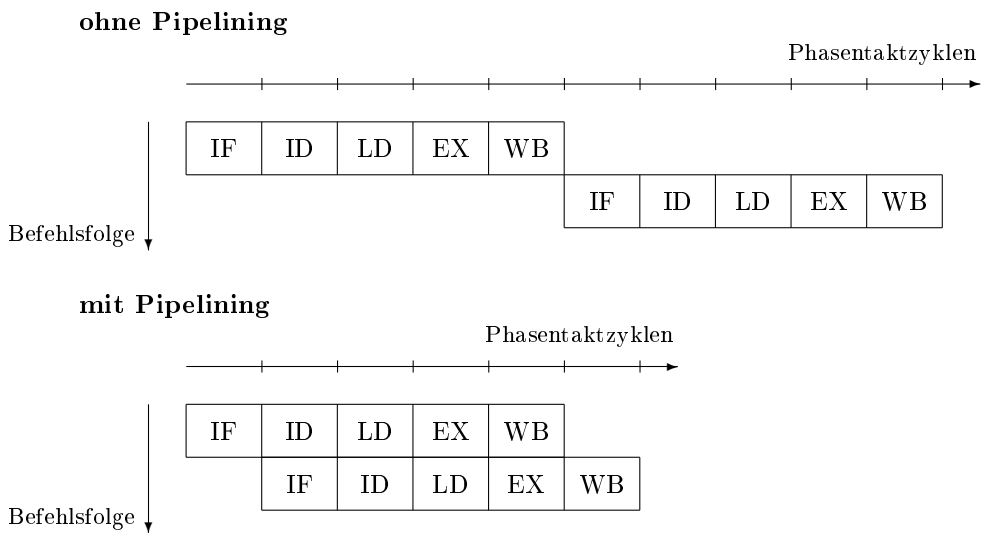


Abbildung 2.54: fünfstufige Pipeline
(IF *Instruction Fetch*, ID *Decode*, LD *Load*, EX *Execute*, WB *Write Back*)

Abbildung 2.54 hilft, diese Formel nachzuvollziehen: der erste Befehl benötigt für seine gesamte Ausführung k Zyklen. Der zweite Befehl befindet sich zu diesem Zeitpunkt in der letzten Phase der Pipeline und benötigt nur mehr 1 Zyklus zu seiner Fertigstellung. Ist dies erreicht, befindet sich der dritte Befehl in seiner letzten Phase, usw. Außer dem ersten Befehl benötigt jeder der $(n-1)$ übrigen Befehle also je 1 weiteren Zyklus.

Vergleicht man diese Zeit mit der „normalen“ Ausführungszeit $T_{oP} = nk\tau$, so ergibt sich der Leistungsgewinn

$$\frac{T_{oP}}{T} = \frac{nk}{k+n-1}.$$

Der Grenzwert dieses Ausdrucks für sehr große n lautet

$$\lim_{n\to\infty}\left(\frac{nk}{k+n-1}\right) = k,$$

d. h. der Leistungsgewinn könnte theoretisch durch die Wahl einer entsprechend großen Stufenzahl beliebig skaliert werden.

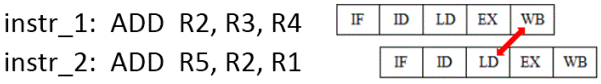
Um die Stufenzahl zu erhöhen, müssten die einzelnen Phasen in *Teilphasen* aufgeteilt werden. Insbesondere die (wegen des Speicherzugriffs langsamen) Befehls- und Operandenholphasen sowie die Rückschreibephase bieten sich für eine feinere Unterteilung an. Diese Untergliederung in Teilphasen nennt man **Superpipelining**. Da die Teilphasen auch kürzere Durchlaufzeiten benötigen, kann die Taktrate der Pipeline zudem erhöht werden.

Leider lässt sich die Befehlsabarbeitung jedoch nicht in beliebig viele Stufen unterteilen. Neben der Unteilbarkeit atomarer Operationen ist ein Hauptgrund das zunehmende Auftreten von Konflikten aufgrund von diversen Abhängigkeiten. Man unterscheidet

- Datenflusskonflikte,
- Steuerflusskonflikte und
- Ressourcenkonflikte.

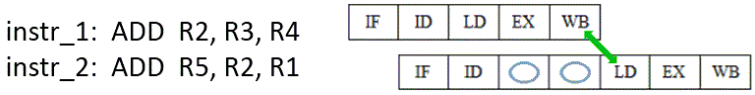
Datenflusskonflikte beziehen sich auf logische Abhängigkeiten sequentieller Befehle, die durch die überlappende Ausführung der Befehle zu falschen Ergebnissen führen können.

Prozessoren, die ihre Befehle streng nacheinander abarbeiten[47], müssen eine Form der Datenabhängigkeit beachten, die als *Read-After-Write*-Abhängigkeit (**RAW**) bezeichnet wird. Eine RAW-Abhängigkeit besteht, wenn ein Operand verändert und in einer Folgeanweisung wieder gelesen wird, z. B.:



Der zweite Befehl holt in seiner Operandenholphase bereits R2, während der erste Befehl R2 noch gar nicht geschrieben hat.

Eine Lösung dieses Problems ist eine künstliche Verlängerung der vorhergehenden Pipelinestufe durch Wartezyklen.



Das Einfügen von Wartezyklen wird *Pipeline Stall* genannt.

Ein einfaches Verfahren, das RAW-Konflikte erkennen und beheben kann, ist das *Scoreboard*-Verfahren. Hier wird mit jedem Register ein zusätzliches Kennzeichnungsbit assoziiert – das *Valid*-Bit. In der Dekodierphase wird für jeden Befehl erkannt, ob ein Ergebnis zu schreiben ist und welches Register ggf. betroffen ist. Dieses wird auf „invalid“ gesetzt und erst nach Abschluss der Rückschreibephase wieder auf „valid“ zurückgesetzt. Muss ein anderer Befehl auf ein als ungültig markiertes Register zugreifen, wird seine Operandenholphase solange hinausgezögert, bis der benötigte Registerinhalt gültig ist.

Steuerflusskonflikte entstehen bei Sprungbefehlen. Zum Holen des Folgebefehls wird hier beim Befüllen der Pipeline von der Fetch-Stufe das Sprungziel bereits benötigt, bevor es berechnet ist (der vorherige Befehl, also der Sprungbefehl, befindet sich ja noch in der Decode-Stufe).

Im Falle eines *unbedingten* Sprungbefehls bedeutet dies, dass die Pipeline mit dem Erkennen des Sprungbefehls abgebrochen werden muss und erst nach der Berechnung des Sprungziels[48] neu befüllt werden kann. Alle Befehle, die bis zum Abbruch in die Pipeline eingeflossen sind, müssen zuvor verworfen werden (dies wird als *Pipeline Flush* bezeichnet). Eine einfache Lösung, um Hardwareaufwand zur Realisierung von Pipeline Flushes zu vermeiden, ist die *Delayed Branch*-Methode. Hier wird einfach dafür gesorgt, dass ein Compiler/Assembler nach jedem Sprungbefehl eine ausreichende Anzahl an NOP-Befehlen einfügt, bis sichergestellt ist, dass der Befehlszähler für die anschließende Fetch-Phase das richtige Sprungziel nennt.

Im Falle eines *bedingten* Sprungbefehls ist diese Lösung aber nicht optimal. Es könnte ja sein, dass bei nicht-erfüllter Bedingung gar nicht gesprungen werden muss. Somit ist ein Flush der Pipeline in diesem Fall gar nicht erforderlich. Ob ein bedingter Sprung ausgeführt (*taken*) oder nicht ausgeführt (*not taken*) werden muss, wird erst nach der Auswertung der Sprungbedingung klar, d. h. nach der Execute-Phase.

Zur Verringerung der Zeitverluste bei einem Steuerflusskonflikt kann man versuchen vorauszusagen, ob ein bedingter Sprung erfolgen wird oder nicht. Man nennt dies *Sprungvorhersage* (**Branch Prediction**). Die Vorhersage ist nur mit einer gewissen Wahrscheinlichkeit richtig.

[47] In sog. *superskalaren* Architekturen werden Befehle teilweise in einer anderen Reihenfolge ausgeführt, als sie im Programmcode stehen, um die Stufen der Pipeline besser auszulasten.

[48] Dies ist bspw. bei relativer Adressierung notwendig und erfolgt in der EX-Stufe.

Die Sprungvorhersage erfolgt so früh wie möglich, also typischerweise in der Decode-Phase, und wird durch ein Spezialwerk geleistet – die *Branch Prediction Unit* (BPU).

Das Arbeitsprinzip der BPU beruht darauf, dass ein Automat durch Sprünge, die in der Vergangenheit erfolgt sind, auf Voraussagen für die Zukunft „trainiert“ wird. So wird für die letzten ca. 128 bis 512 bedingten Sprungbefehle, die jeweils durch ihre Hauptspeicheradresse identifiziert werden, in einem speziellen Cache (*Branch History Table*, BHT) ein Zustandscode gespeichert, der aussagt, ob bei der nächsten Ausführung ein Sprung eher stattfindet (*taken*) oder eher nicht stattfindet (*not taken*).

Ein solcher Automat mit vier Zuständen (2-bit Zustandscode) ist in Abbildung 2.55 dargestellt.

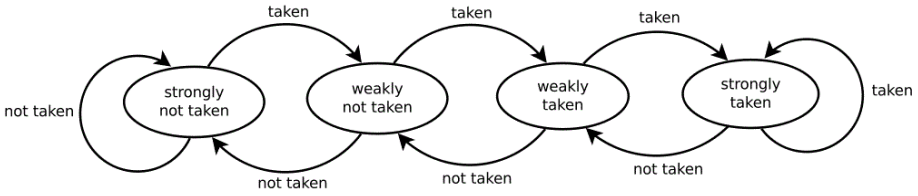


Abbildung 2.55: MOORE-Automat zur Sprungvorhersage

In den Zuständen *strongly taken* und *weakly taken* tippt der Automat beidemal auf „Sprung“, in den zwei anderen Zuständen jeweils auf „kein Sprung“. Abhängig vom tatsächlichen Ausgang der späteren Berechnung (*taken* oder *not taken*) verändert er anschließend seinen Zustand. Ergibt sich, dass ein Sprung entgegen der *not taken*-Vorhersage doch ausgeführt werden muss, so ist trotz allem ein Pipeline Flush erforderlich.

Auch für die Sprungvorhersage gibt es mittlerweile verbesserte Verfahren, die allerdings im Rahmen dieser Lehrveranstaltung nicht mehr behandelt werden sollen.

Die letzte Konfliktform bilden **Ressourcenkonflikte**, die immer dann vorliegen, wenn eine Stufe der Pipeline Zugriff auf ein Betriebsmittel benötigt, das bereits von einer anderen Stufe belegt wird.

Typische Ressourcenkonflikte treten auf, wenn mindestens zwei Pipelinestufen die ALU benötigen (z. B. zur Adressberechnung in der Load-Phase und für eine arithmetische Operation in der Execute-Phase) oder einen Hauptspeicherzugriff durchführen müssen (z. B. in der Fetch-Phase und in der Load-Phase).

Ihre Behandlung erfolgt genauso wie bei Datenflusskonflikten durch Pipeline Stalls, ihre Erkennung durch Scoreboarding.

Literaturverzeichnis

[1] Peter J. Denning, Douglas E. Comer, David Gries, Michael C. Mulder, Alan Tucker, A. Joe Turner, Paul R. Young: Computing as a Discipline, Communications of the ACM, Vol. 32, No. 1 (1989) 9 - 23, http://denninginstitute.com/pjd/GP/CompDisc.pdf.

[2] Alan M. Turing: On Computable Numbers, with an Application to the Entscheidungsproblem, Proc. of the London Mathematical Society, Vol. s2-42, No. 1 (1937) 230 - 265, https://londmathsoc.onlinelibrary.wiley.com/doi/epdf/10.1112/plms/s2-42.1.230.

[3] Uwe Schöning: Theoretische Informatik – kurz gefasst, Spektrum Akademischer Verlag, 5. Auflage (2008).

[4] Einige Beispiele zur Turingmaschine, https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.040/Formale_Methoden_der_Informatik/Übungen/blatt08-zusatz_2.pdf.

[5] Matthias Jantzen: Berechenbarkeit und Komplexität, Unterlagen zur Vorlesung, Universität Hamburg (2001), https://www.informatik.uni-hamburg.de/TGI/lehre/vl/WS0102/F3/F3script.pdf.

[6] Arno Schwarz: Prinzipielle Grenzen der Berechenbarkeit, Unterrichtsmaterialien für die Jahrgangsstufe 12, Ministerium für Bildung, Kultur und Wissenschaft Saarland, https://www.saarland.de/dokumente/thema_bildung/Prinzipielle_Grenzen_der_Berechenbarkeit.pdf.

[7] Pierluigi Crescenzi, Viggo Kann: A compendium of NP optimization problems, KTH Stockholm (2005), http://www.nada.kth.se/~viggo/wwwcompendium/.

[8] J. Glenn Brookshear: Computer Science, An Overview, 12$^{\text{th}}$ ed., Addison-Wesley (2015).

[9] J.-F. Lin, Y.-T. Hwang, M.-H. Sheu, C.-C. Ho: A Novel High-Speed and Energy Efficient 10-Transistor Full Adder Design, IEEE Transactions on Circuits and Systems, Vol. 54, 5 (2007) 1050-1059.

[10] John von Neumann: First Draft of a Report on the EDVAC, Moore School of Electrical Engineering, University of Pennsylvania (1945), https://archive.org/details/firstdraftofrepo00vonn.

[11] David A. Patterson, John L. Hennessy: Computer Organisation and Design, 5$^{\text{th}}$ ed., Morgan Kaufmann (2014).

[12] wikimedia commons, https://commons.wikimedia.org/wiki/File:DRAM-Zelle_(stack).svg.

[13] Hongqing Liu, Stacy Weng, Wei Sun: Introduction of Cache Memory, CMSC 411, Summer 2001, University of Maryland, http://www.cs.umd.edu/class/fall2001/cmsc411/proj01/cache/cache.pdf.

[14] Baskaran Ganesan: Introduction to Multi-Core,
`http://www.ecs.umass.edu/ece/andras/courses/ECE668/Mylectures/Introduction_to_Multi_Core.pdf`.