Prof. Dr. Lena Oden

65050
Rechnerarchitektur

Lehrveranstaltung
Computersysteme

Leseprobe

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

1. Digitaltechnik und Schaltwerke

Contents

Einschub: Lernziele dieser Kurseinheit

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie in der Lage sein:

- die Darstellung von Zahlen (Integer, Gleitkomma) und Zeichen im Binär- und Hexadezimalsystem zu erklären und ineinander umzurechnen.
- die grundlegenden Logikgatter (AND, OR, NOT, NAND, NOR, XOR, XNOR) und ihre Wahrheitstabellen zu benennen sowie Schaltfunktionen in den drei Darstellungsformen (Wahrheitstabelle, Funktionsgleichung, Schaltplan) anzugeben.
- kombinatorische Standardbausteine (Multiplexer, Demultiplexer, Decoder, Komparator) zu erklären und ihre Rolle in Datenpfaden zu beschreiben.
- den Aufbau einer ALU aus Addierern und MUX herzuleiten und die Funktion von Status-Flags zu erläutern.
- den Unterschied zwischen Latch und Flip-Flop zu erklären und die Bedeutung der Flankensteuerung für Pipelines zu begründen.

Das Ziel dieser ersten Kurseinheit im Kurs Rechnerarchitektur ist es, dass Sie ein besseres Verständnis davon bekommen, *wie* ein Computer im Inneren tatsächlich arbeitet. Dafür müssen wir uns sowohl damit befassen, wie Information in einem Computer gespeichert wird und auf welche Art diese Information durch einen Computer verarbeitet wird.

1.1 Analog vs. Digital: Die Welt der zwei Zustände

Wir leben in einer analogen Welt. Unsere Umwelt ist kontinuierlich: Die Temperatur steigt stufenlos, eine Schallwelle schwingt in unendlich feinen Nuancen. Frühe Rechenmaschinen (Analogrechner) versuchten, diese Welt direkt abzubilden, indem sie z. B. elektrische Spannungen stufenlos variierten (3,4*Volt* standen für den Wert 3,45). Das Problem dabei ist die Realität: Ein analoges Signal ist anfällig für Störungen (Rauschen). Wenn eine Leitung korrodiert ist und aus 3,45 Volt plötzlich 3,42 Volt werden, ist die Information verfälscht. Um Rechensysteme robust zu machen, nutzen wir heute vor allem digitale Signale [1]. *Digital* bedeutet hier diskret: Wir erlauben nicht mehr unendlich viele Werte, sondern nur noch eine begrenzte Menge an fest definierten Zuständen. Im Extremfall, und das ist der Standard in der heutigen Informatik, reduzieren wir das System auf das absolute Minimum mit zwei Zuständen.

- **1 (High):** Spannung liegt an / Strom fließt.
- **0 (Low):** Keine Spannung / Strom fließt nicht.

In der Informatik abstrahieren wir diese physikalischen Spannungen zu den Symbolen 1 und 0.

Das ist die wichtigste Erkenntnis für diesen Kurs: Egal wie komplex eine Anwendung erscheint, ob es sich um ein 4K-Video, eine künstliche Intelligenz oder dieses Skript im PDF-Format handelt: auf der untersten Ebene der Hardware existieren keine Bilder, keine Buchstaben und keine Zahlen. Es gibt nur Milliarden von winzigen Schaltern (Transistoren), die entweder offen (0) oder geschlossen (1) sind.

[1] Tatsächlich erfährt das analoge Rechnen in den letzten Jahren durch neue Speichertechnologien im Bereich wie maschinelles Lernen, wo es manchmal auf Genauigkeit nicht so ankommt, wieder einen Aufwind, zumindest in der Forschung. Bis diese Technologien allerdings Marktreife haben, wird es wohl noch etwas dauern, daher bleiben wir in diesen Grundlagen erst einmal *digital*.

Der Computer ist also nichts anderes als eine gigantische Ansammlung von Schaltern. Die *Magie* entsteht erst dadurch, wie wir diese 0en und 1en interpretieren.

- Interpretieren wir 32 Schalter als Integer, ergibt 00...0101 die Zahl 5.
- Interpretieren wir sie als ASCII, ist es vielleicht ein Buchstabe.
- Interpretieren wir sie als Befehl (Opcode), sagen wir der CPU: *Addiere jetzt!*

Bevor wir uns damit beschäftigen, wie ein Computer rechnet (Prozessor-Architektur), müssen wir klären, womit er eigentlich arbeitet. Auf einer abstrakten Ebene ist ein Computer keine Maschine, die Strom verarbeitet, sondern eine Maschine, die **Informationen** verarbeitet.

Wir beginnen daher mit der Frage, wie diese Informationen digital dargestellt werden – vom einfachen Bit bis hin zu komplexen Gleitkommazahlen. Anschließend lernen wir, wie sich diese simplen Zustände nutzen lassen, um daraus Logik zu bauen.

Selbsttestaufgabe 1.1 Ein Sensor misst eine Temperatur und gibt sie als analoge Spannung aus. Bei $20C$ liegen $2{,}0V$ an, bei $21C$ liegen $2{,}1V$ an. Durch ein defektes Kabel wird die Spannung um $0{,}08V$ verfälscht.

1. Welchen Temperaturwert würde ein analoges System messen?
2. Erklären Sie, warum ein digitales System diesen Fehler vollständig ignorieren könnte. Welche Eigenschaft macht das möglich?

1.2 Datenformate

Einschub: Notation: Hexadezimal, Nibble, Byte und Word

In diesem Abschnitt und in späteren Kurseinheiten begegnen Ihnen häufig Kurzschreibweisen statt langer Binärfolgen. Die folgende Übersicht erklärt die wichtigsten Einheiten und Notationen.

In der Hardware werden Bits selten einzeln verarbeitet. Um die Effizienz zu steigern, bündelt man sie zu festen Gruppen:

- **Byte:** Die kleinste adressierbare Einheit besteht fast immer aus **8 Bit**. Ein Byte kann $2^8 = 256$ verschiedene Zustände (z. B. die Zahlen 0–255) darstellen.
- **Word (Wort):** Ein Wort entspricht der natürlichen Verarbeitungsbreite eines Prozessors (Registerbreite). Bei modernen Architekturen sind dies meist **32 Bit** (4 Bytes) oder **64 Bit** (8 Bytes).

Diese Bündelung ist der Grund für den Erfolg der **Hexadezimal-Darstellung**. Da $16 = 2^4$ ist, lassen sich exakt 4 Bits (ein sogenanntes *Nibble*) durch eine einzige Hex-Ziffer darstellen. Das Hexadezimalsystem ist ein Zahlensystem zur Basis 16, das die Ziffern 0 bis 9 und die Buchstaben A bis F verwendet. Ein Byte lässt sich somit immer perfekt durch genau zwei Hex-Ziffern beschreiben (z. B. $\texttt{1111 1111}_2 = \texttt{0xFF}_{16}$).

Dies macht Hexadezimal zur idealen *Kurzschrift* für Speicherzustände: kompakter als Binär, aber – im Gegensatz zum Dezimalsystem – lässt es die zugrundeliegende Bit-Struktur (die Byte-Grenzen) sofort erkennen. In der Tabelle 1.1 sind die Zahlen von 1 bis 16 dargestellt.

Umrechnung von Dezimal zu Hexadezimal:

Beispiel: Die Dezimalzahl **419** soll in das Hexadezimalsystem umgerechnet werden.

1. $419 : 16 = 26$, Rest **3** (entspricht hex **3**)
2. $26 : 16 = 1$, Rest **10** (entspricht hex **A**)
3. $1 : 16 = 0$, Rest **1** (entspricht hex **1**)

Nun liest man die Reste von **unten nach oben** ab:

$$419_{10} = 1A3_{16}$$

Dezimal	Binär	Hexadezimal
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10

Tabelle 1.1: Vergleich der Zahlensysteme

Bits allein haben keine inhärente Bedeutung – erst das **Datenformat** legt fest, wie eine Bitfolge zu interpretieren ist: als Zahl, als Zeichen oder als Befehl. Im Folgenden betrachten wir die wichtigsten Formate, denen wir in der Rechnerarchitektur begegnen werden.

1.2.1 Ganzzahldatenformate (Integers)

Ganzzahlen können unterschiedlich lang (z. B. 8, 16, 32 oder 64 Bit) und jeweils mit Vorzeichen (*signed*) oder ohne Vorzeichen (*unsigned*) definiert sein.

Arbeitet man nur mit positiven Zahlen, also Zahlen ohne Vorzeichen, entspricht die Binärdarstellung einfach der Darstellung im dualen Zahlensystem, wie in der Tabelle 1.2 in der 2. Spalte für positive Zahlen dargestellt. Interessanter wird es jedoch, wenn man auch negative Zahlen zulässt. Die häufigste Darstellungsform für vorzeichenbehaftete Zahlen ist das Zweierkomplement.

Das Zweierkomplement:

Dies ist das Standardformat für vorzeichenbehaftete Zahlen. Um das Zweierkomplement einer Zahl zu bilden, folgt man der Regel: *Invertieren und Eins addieren.* Das bedeutet, dass zunächst jedes Bit invertiert wird und anschließend Eins addiert wird.

Dezimal	Binär	Hexadezimal
1	b00000001	0x01
2	b00000010	0x02
8	b00001000	0x08
39	b00100111	0x27
-1	b11111111	0xFF
-39	b11011001	0xD9

Tabelle 1.2: Zahlendarstellung für 8-Bit-Zahlen. Positive Werte sind in vorzeichenloser Binärdarstellung angegeben (identisch mit dem Zweierkomplement); negative Werte ausschließlich im Zweierkomplement.

Beispiel 1.1 **Wandlung von $+5$ in -5 in 8 Bit**

1. $+5$ binär darstellen: `0000 0101`
2. Bits invertieren (Einerkomplement): `1111 1010`
3. 1 addieren: `1111 1011`

Das Ergebnis `1111 1011` repräsentiert die Zahl -5.

Der große Vorteil dieser Darstellung ist die Implementierung in Hardware: Die Subtraktion kann auf eine einfache Addition zurückgeführt werden ($A - B = A + (-B)$). Ein Rechenwerk (ALU, darauf werden wir im Laufe des Kurses noch eingehen) muss also nicht *wissen*, ob es addiert oder subtrahiert; die Logik bleibt dieselbe und eine Addition und eine Subtraktion können durch dieselbe Hardware implementiert werden. Wir wollen es an einem Beispiel zeigen:

Beispiel 1.2 $7 - 5 = 2$

Wir rechnen $7 + (-5)$ im 8-Bit-System:

```
   0000 0111  ( 7)
+  1111 1011  (-5)
--------------
 (1)0000 0010  (Übertrag wird verworfen)
```

Das Ergebnis ist `0000 0010`, was exakt der Dezimalzahl 2 entspricht.

1.2.2 ASCII und BCD

Neben reinen Zahlenwerten müssen Computer auch Texte und dezimale Ziffernfolgen verarbeiten können. Hierfür haben sich spezialisierte Formate etabliert.

- **ASCII (American Standard Code for Information Interchange):** Dies ist das grundlegende Format zur Textrepräsentation. Jedem Schriftzeichen (Buchstaben, Ziffern, Satzzeichen und Steuerzeichen) wird ein fester Zahlenwert zugeordnet, der

genau ein Byte (8 Bit) belegt[2].
  - *Beispiel:* Der Buchstabe 'A' entspricht dezimal 65 (`0x41`), das Leerzeichen dezimal 32 (`0x20`).
  - In einem 32-Bit-Wort können somit bis zu vier ASCII-Zeichen gleichzeitig gespeichert und übertragen werden.
- **BCD (Binary Coded Decimal):** BCD wird vor allem dort eingesetzt, wo exakte Dezimalwerte ohne Rundungsfehler verarbeitet werden müssen (z. B. in kaufmännischen Anwendungen oder bei digitalen Anzeigen). Anstatt die gesamte Zahl binär zu wandeln, wird jede **einzelne Dezimalziffer** separat in 4 Bits (einem Nibble) codiert. Man unterscheidet zwei Speicherformen:
  - **Ungepacktes BCD:** Jedes Byte speichert nur eine Dezimalziffer in den unteren 4 Bits; die oberen 4 Bits bleiben meist ungenutzt (oft als `0000` oder `1111` aufgefüllt).
  - **Gepacktes BCD:** Jedes Byte speichert zwei Dezimalziffern (eine im oberen und eine im unteren Nibble). Dies halbiert den Speicherbedarf gegenüber der ungepackten Form.

Beispiel 1.3 Darstellung der Zahl 1548 in BCD

Die Dezimalziffern sind $1, 5, 4, 8$. Binär codiert als 4-Bit-Blöcke: `0001, 0101, 0100, 1000`.
- **Gepackt (16 Bit):** `0001 0101 | 0100 1000` (Hex: `0x1548`)
- **Ungepackt (32 Bit):** `.... 0001 | .... 0101 | .... 0100 | .... 1000`

1.2.3 Gleitkommadatenformate (Floats)

Für die Darstellung von sehr großen Zahlen (z. B. in der Astronomie) oder sehr kleinen Brüchen (z. B. in der Quantenphysik) reichen Ganzzahlen nicht aus. Hier nutzen wir den **IEEE-754-Standard** für die Darstellung von sogenannten *Gleitkommazahlen*. Gleitkommazahlen heißen so, weil das Komma nicht an einer festen Stelle steht, sondern durch einen Exponenten flexibel über die Ziffernfolge *„gleiten"* kann, um sowohl riesige als auch winzige Werte effizient darzustellen.

Eine Gleitkommazahl f wird mathematisch wie folgt beschrieben:

$$f = (-1)^s \cdot 1.m \cdot 2^{e-b}$$

Dabei ist:
- s **(Sign):** Das Vorzeichenbit (0 für positiv, 1 für negativ).
- m **(Mantisse):** Die Nachkommastellen der normalisierten Binärzahl. Die führende Eins (vor dem Komma) wird nicht gespeichert (Hidden Bit), um Platz zu sparen.
- e **(Exponent):** Der gespeicherte, verschobene Exponent.
- b **(Bias):** Ein fester Verschiebewert, damit der Exponent ohne extra Vorzeichenbit auch negative Werte (für kleine Brüche) annehmen kann.

[2] Der originale ASCII-Standard (ANSI X3.4-1968) ist ein 7-Bit-Code und umfasst 128 Zeichen. Das achte Bit wurde erst in der *Extended ASCII*-Erweiterung genutzt, um weitere Zeichen darzustellen. In modernen Systemen hat *Unicode* (z. B. als UTF-8 kodiert) ASCII weitgehend abgelöst, bleibt aber für die Zeichen 0–127 vollständig kompatibel.

Vergleich der Genauigkeiten

In modernen Systemen sind vor allem zwei Varianten des IEEE-754-Standards gebräuchlich:

Typ	**Breite**	**Exponent** (e)	**Mantisse** (m)	**Bias** (b)
Single Precision	32 Bit	8 Bit	23 Bit	127
Double Precision	64 Bit	11 Bit	52 Bit	1023

Tabelle 1.3: Vergleich Single vs. Double Precision nach IEEE-754.

Beispiel 1.4 Detaillierte Wandlung von $12,78125$ in IEEE-754 (32-Bit)

Um eine Dezimalzahl in das IEEE-754-Format umzurechnen, gehen wir in fünf exakten Teilschritten vor:

1. **Vorzeichen bestimmen:**
   Da die Zahl $+12,78125$ positiv ist, setzen wir das Vorzeichenbit $\boxed{s = 0}$.
2. **Getrennte Binärwandlung:**
   - **Vorkommastelle (12):** Durch wiederholtes Teilen durch 2 erhalten wir $12 = 1100_2$.
   - **Nachkommastelle (0,78125):** Diese wird wiederholt mit 2 multipliziert. Der Wert vor dem Komma (0 oder 1) ergibt die Binärziffer:
     - $0,78125 \cdot 2 = \mathbf{1},5625 \rightarrow 1$ merken
     - $0,5625 \cdot 2 = \mathbf{1},125 \rightarrow 1$ merken
     - $0,125 \cdot 2 = \mathbf{0},25 \rightarrow 0$ merken
     - $0,25 \cdot 2 = \mathbf{0},5 \rightarrow 0$ merken
     - $0,5 \cdot 2 = \mathbf{1},0 \rightarrow 1$ merken (Ende, da Rest 0)

     Daraus folgt: $0,78125_{10} = 0,11001_2$.

   Zusammengesetzt ergibt sich die vorläufige Binärzahl: $1100,11001_2$.
3. **Normalisierung:**
   Wir verschieben das Komma so, dass genau eine Eins davor steht. Dazu schieben wir es um 3 Stellen nach links:

   $$1100,11001 \rightarrow 1,10011001 \cdot 2^3$$

   Der wahre Exponent ist also $E = 3$. Die Ziffern hinter dem Komma bilden unsere Mantisse.
4. **Berechnung des Bias-Exponenten (e):**
   Für 32-Bit gilt $b = 127$. Der zu speichernde Wert ist:

   $$e = E + b = 3 + 127 = 130$$

   Die Zahl 130 wird binär dargestellt: $\boxed{130_{10} = 10000010_2}$.
5. **Zusammensetzen des 32-Bit-Worts:**

S (1 Bit)	**Exponent** (8 Bit)	**Mantisse** (23 Bit)
0	10000010	10011001000000000000000

Das Ergebnis in Hexadezimal-Kurzschrift lautet: `0x414C8000`.

Eingeschränkte Genauigkeit und Darstellungsfehler

Ein häufiges Missverständnis ist die Annahme, dass jede im Dezimalsystem exakt darstellbare Zahl auch im IEEE-754-Format exakt gespeichert werden kann. Dies ist jedoch nicht der Fall.

Das Problem kommt durch die binären Brüche zustande. Ein Bruch lässt sich in einer Basis nur dann exakt darstellen, wenn sein Nenner ein Produkt der Primfaktoren dieser Basis ist. Dies führt dazu, dass eine im Dezimalsystem darstellbare Zahl dies nicht im Binärsystem ist.

- Im **Dezimalsystem** (Basis $10 = 2 \cdot 5$) lassen sich Brüche wie $1/2, 1/5$ und $1/10$ exakt darstellen.
- Im **Binärsystem** (Basis 2) lassen sich *nur* Brüche exakt darstellen, deren Nenner eine **Zweierpotenz** ist $(1/2, 1/4, 1/8, \dots)$.

Beispiel 1.5 Die Zahl 0,1

Betrachten wir die vermeintlich einfache Dezimalzahl $0,1$ $(1/10)$:

- $0,1 \cdot 2 = \mathbf{0},2$
- $0,2 \cdot 2 = \mathbf{0},4$
- $0,4 \cdot 2 = \mathbf{0},8$
- $0,8 \cdot 2 = \mathbf{1},6$
- $0,6 \cdot 2 = \mathbf{1},2 \rightarrow$ ab hier wiederholt sich der Zyklus (**periodisch**!)

Das Ergebnis im Binärsystem ist $0,0\overline{0011}_2$. Da die Mantisse im IEEE-754-Format jedoch auf 23 Bit (Single) bzw. 52 Bit (Double) begrenzt ist, muss diese unendliche Periode irgendwann abgebrochen werden.

Dies führt dazu, dass $0,1$ im Computer geringfügig größer oder kleiner gespeichert wird, als es der mathematischen Wahrheit entspricht [3].

Ein bekanntes Phänomen in fast allen Programmiersprachen ist daher:

```
0.1 + 0.2 != 0.3  // ergibt oft z.B. 0.30000000000000004
```

Für Hardware-Architektinnen bedeutet dies: Rechenwerke für Gleitkommazahlen müssen komplexe Strategien zur **Rundung** implementieren (z. B. Round-to-nearest-even), um diese unvermeidbaren Fehler zu minimieren. Dies ist ein Grund, warum sogenannte FPUs (Floating Point Units) so viel Chipfläche beanspruchen.

Spezialwerte nach IEEE-754

Der IEEE-754-Standard reserviert bestimmte Bitmuster im Exponenten, um mathematische Ausnahmesituationen und Grenzwerte darzustellen. Dies stellt sicher, dass ein Prozessor bei Berechnungen wie einer Division durch Null nicht einfach „anhält“, sondern ein definiertes Ergebnis liefert.

Die Steuerung erfolgt primär über den **Exponenten**:

[3] Ob das Ergebnis geringfügig größer oder kleiner ist hängt von der Rundungsstrategie der Hardware ab. Die am häufigsten verwendete Strategie (Default) in modernen Prozessoren ist Round to Nearest (Even).

- **Exponent = 00...0:** Repräsentiert die Null oder denormalisierte Zahlen.
- **Exponent = 11...1:** Repräsentiert Unendlichkeit oder Fehlerzustände (NaN).

Null und das Vorzeichen

Interessanterweise erlaubt IEEE-754 zwei Arten von Null: $+0$ und -0.

- Bitmuster: Exponent und Mantisse sind komplett Null.
- Das Vorzeichenbit s bestimmt, ob es $+0$ oder -0 ist. In der Praxis verhalten sich beide meist identisch, was jedoch zeigt, wie strikt die Hardware-Logik dem Bit-Format folgt.

Unendlichkeit (Infinity)

Wird ein positiver Wert durch Null geteilt oder übersteigt das Ergebnis den darstellbaren Wertebereich (Overflow), resultiert dies in *Infinity*.

- **Bitmuster:** Exponent = `11...1`, Mantisse = `00...0`.
- Man unterscheidet gemäß dem Vorzeichenbit zwischen $+\infty$ und $-\infty$.

NaN (Not a Number)

Ein *NaN* tritt auf, wenn eine Operation mathematisch nicht definiert ist, wie etwa $0/0$ oder die Quadratwurzel aus einer negativen Zahl (wir haben hier keine komplexen Zahlen, diese werden i.d.R. in Software definiert).

- **Bitmuster:** Exponent = `11...1`, Mantisse $\neq$ `00...0`.
- Sobald ein *NaN* in einer Rechnung auftaucht, bleibt das Ergebnis für alle weiteren Rechenschritte *NaN* („Fehlerfortpflanzung“).

Denormalisierte Zahlen (Subnormal Numbers)

Diese Zahlen füllen die Lücke zwischen der kleinsten normalisierten Zahl und der Null.

- **Bedingung:** Exponent = `00...0`, Mantisse $\neq$ `00...0`.
- **Besonderheit:** Hier entfällt die implizite führende Eins ($1.m$). Stattdessen wird die Mantisse als $0.m$ interpretiert. Dies erlaubt ein „sanftes Unterlaufen“ (*Gradual Underflow*) bei sehr kleinen Werten, ist für die Hardware-Rechenwerke jedoch deutlich aufwendiger zu berechnen als der Normalfall.

Wert	**Exponent (e)**	**Mantisse (m)**
Null (± 0)	$00\ldots0$	$00\ldots0$
Denormalisiert	$00\ldots0$	$\neq 0$
Normalisiert	$00\ldots0 < e < 11\ldots1$	beliebig
Unendlich ($\pm\infty$)	$11\ldots1$	$00\ldots0$
NaN (Not a Number)	$11\ldots1$	$\neq 0$

Tabelle 1.4: Zusammenfassung der Spezialwerte im IEEE-754 Format.

Damit schließen wir den Abschnitt der Informationsdarstellung ab. Wir haben gesehen, wie aus einfachen Bits komplexe Zahlen und Logikvorgaben werden. Im nächsten Abschnitt werden wir diese Bits „in Bewegung setzen“ und lernen, wie physikalische Gatter daraus Ergebnisse berechnen.

Selbsttestaufgabe 1.2 1. Wandeln Sie die Dezimalzahl -42 in die 8-Bit-Zweierkomplement-Darstellung um. Geben Sie alle Zwischenschritte an und notieren Sie das Ergebnis zusätzlich als Hexadezimalzahl.

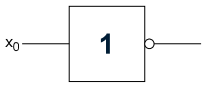
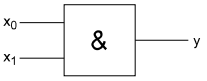
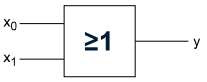
Operator	Schaltzeichen	Benennung
$\neg$		Nicht-Glied (Not)
$\wedge$		UND-Glied (AND)
$\vee$		ODER-Glied (OR)

Tabelle 1.5: Grundoperatoren und Schaltzeichen

2. Warum kann die Dezimalzahl 0,1 im IEEE-754-Format nicht exakt dargestellt werden? Begründen Sie anhand der Eigenschaften des Binärsystems.

1.3 Logikgatter und Schaltalgebra

Nachdem wir definiert haben, dass digitale Systeme mit den Zuständen 0 und 1 arbeiten, stellt sich die Frage: Wie verarbeiten wir diese Zustände?

In der Digitaltechnik arbeitet man hier mit sogenannten Logikgattern – der kleinsten funktionalen Baueinheit einer digitalen Schaltung. Man kann es sich wie einen elektronischen Schalter vorstellen, der nach festen logischen Regeln entscheidet, ob am Ausgang Strom fließt (Zustand 1 oder „High“) oder nicht (Zustand 0 oder „Low“). Ein Gatter hat meist zwei oder mehr Eingänge und genau einen Ausgang. Es verarbeitet die ankommenden Signale basierend auf der *Booleschen Algebra*.

Um komplexe Schaltungen zu entwerfen, benötigen wir ein mathematisches Werkzeug, mit dem wir Logik formulieren und vereinfachen können, denn jedes eingesparte Gatter bedeutet weniger Stromverbrauch und höhere Geschwindigkeit.

Dieses Werkzeug ist die **Schaltalgebra**. Sie ist die Anwendung der Boole'schen Algebra auf die Menge $\boldsymbol{B} = \{0, 1\}$. Die Verbindung zur Realität ist dabei direkt:

- Die Werte **0** und **1** entsprechen physikalischen Spannungspegeln (Low/High).
- Die mathematischen Operatoren entsprechen den elektrotechnischen **Gattern**.

1.3.1 Die Grundbausteine

Die gesamte digitale Welt lässt sich auf nur drei grundlegende Operationen zurückführen. Tabelle 1.5 zeigt die mathematische Schreibweise und die zugehörigen genormten Schaltzeichen nach DIN EN 60617. Mit diesen einfachen Schaltzeichen lässt sich theoretisch jede beliebige Logik darstellen.

1. **UND** (Konjunktion, $\wedge$): Der Ausgang ist nur 1, wenn alle Eingänge 1 sind.
2. **ODER** (Disjunktion, $\vee$): Der Ausgang ist 1, wenn mindestens ein Eingang 1 ist.
3. **NICHT** (Negation, $\overline{a}$): Der Ausgang ist das Gegenteil des Eingangs.

Im weiteren Verlauf werden wir zusätzliche Schaltzeichen kennenlernen, die für den Entwurf digitaler Schaltungen essenziell sind. Interessanterweise lassen sich viele dieser

Gatter, etwa NAND oder NOR, hardwareseitig mit weniger Transistoren realisieren als das klassische UND-Gatter.

Hinweis: Die vollständigen Rechenregeln der Schaltalgebra werden in Abschnitt 1.4.5 im Zusammenhang mit der Vereinfachung von Schaltfunktionen systematisch eingeführt.