Informatik-Skript für die E-Phase

[Jarek Mycan]

23. August 2025

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Der Begriff Informatik

Das Wort **Informatik** setzt sich aus zwei Teilen zusammen: *Information* und *Automatik*. Ursprünglich wurde der Begriff in den 1950er Jahren in Frankreich geprägt („informatique“) und fand später auch im deutschsprachigen Raum Verbreitung. In anderen Sprachen wird der Bereich meist „Computer Science“ genannt, was die technische Seite stärker betont.

1.1.1 Information

Der erste Bestandteil, *Information*, bezieht sich auf Daten, die eine Bedeutung tragen. Information ist also nicht einfach nur eine Ansammlung von Zeichen oder Zahlen, sondern sie entsteht erst durch die *Interpretation* von Daten in einem bestimmten Kontext. Beispielsweise ist die Zahl „42“ zunächst nur ein einzelner Wert. Wird sie jedoch im Zusammenhang mit einer Temperaturangabe, einem Alter oder einem Ergebnis verstanden, wird daraus eine Information. Die Informatik beschäftigt sich also damit, Informationen *darzustellen*, *zu speichern*, *zu übertragen* und *zu verarbeiten*.

1.1.2 Automatik

Der zweite Bestandteil, *Matik* (von Automatik), bedeutet, dass diese Informationsverarbeitung durch Maschinen – insbesondere Computer – automatisiert geschieht. Ein zentrales Ziel der Informatik ist es, Verfahren zu entwickeln, die es ermöglichen, Informationen mithilfe von Computern effizient und zuverlässig zu verarbeiten. Dazu gehören das Erstellen von Programmen, das Entwerfen von Algorithmen sowie die Entwicklung von Systemen, die Informationen ohne manuelle Eingriffe verarbeiten können.

1.2 Bedeutung der Informatik

Die Informatik kann somit als *Wissenschaft von der systematischen Verarbeitung von Informationen, insbesondere mit Hilfe von Computern*, verstanden werden.

Sie ist nicht nur eine technische Disziplin, sondern verbindet Elemente aus Mathematik, Ingenieurwissenschaften, Logik und zunehmend auch Sozial- und Geisteswissenschaften. Heute prägt die Informatik nahezu alle Bereiche des täglichen Lebens: von Smartphones und dem Internet über moderne Autos bis hin zu Medizin, Wirtschaft und Wissenschaft.

1.3 Was tun eigentlich Computer?

Computer *verarbeiten Daten*. Sie führen Berechnungen aus, speichern, übertragen und strukturieren Daten — aber sie „verstehen“ keine Bedeutung im menschlichen Sinn. Bedeutung (Information) entsteht erst beim Menschen (oder in einem Modell), wenn Daten in *Kontext* gesetzt werden. Der Kernprozess ist daher zweistufig:

1. **Repräsentation:** Aus *Information* werden *Daten*, indem wir festlegen, wie etwas als Zeichen/Zahlen (Bits) dargestellt wird.

2. **Abstraktion:** Aus *Daten* wird (wieder) *Information*, indem wir Details weglassen, strukturieren und die Daten in einem Modell deuten.

1.3.1 Von Information zu Daten: Repräsentation

Repräsentation bedeutet: Wir legen eine **Abbildung** fest, die etwas Bedeutungsvolles (Information) in ein **Datenformat** überführt, das der Computer verarbeiten kann. Formal kann man das als Funktion auffassen:

$$\text{rep} : \text{Information} \times \text{Kontext} \rightarrow \text{Daten (Bits)}.$$

1.3.2 Von Daten zu Information: Abstraktion

Abstraktion bedeutet: Wir **interpretieren** Daten in einem passenden Modell und **lassen Details weg**, die für die Fragestellung nicht nötig sind. So entsteht Bedeutung. Formal:

$$\text{abs} : \text{Daten (Bits)} \times \text{Modell/Kontext} \rightarrow \text{Information}.$$

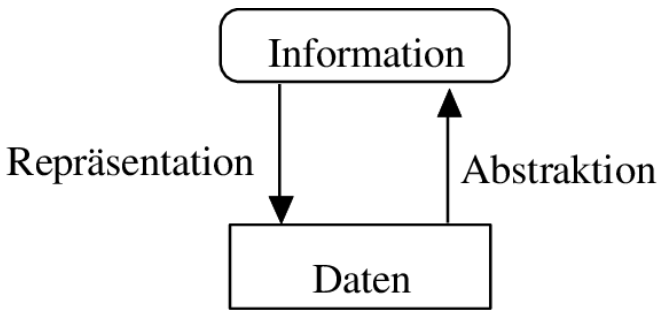

Abbildung 1.1: Wechselspiel zwischen Repräsentation (Information Daten) und Abstraktion (Daten Information).

1.4 Bits und Bytes

Ein **Bit** (binary digit) ist die kleinste Informationseinheit im Rechner: Es kann genau zwei Zustände annehmen, meist als `0` und `1` notiert. Physikalisch werden Bits z. B. durch zwei Spannungsniveaus, magnetische Ausrichtungen oder Lichtimpulse realisiert.

Einzelne Bits sind für die Verarbeitung jedoch zu fein. Deshalb werden Bits zu **Gruppen** zusammengefasst und *gruppenweise* gelesen/geschrieben:

- **Byte** = 8 Bit (heute die gebräuchlichste Grundeinheit; in den meisten Architekturen zugleich die *kleinste adressierbare Einheit*).
- **Nibble** = 4 Bit (halbes Byte; nützlich z. B. bei Hexadezimaldarstellungen).
- **Wort (Word)** = *architekturabhängige* Arbeitsbreite der CPU (typisch 16, 32 oder 64 Bit).
- **Doppelwort/Quadwort** = Vielfache der Wortbreite (z. B. 32/64/128 Bit).

1.4.1 Wer bestimmt die Lesegruppen?

Hardware: Die *Register- und ALU-Breite* eines Prozessors legt fest, wie viele Bits er in einem Schritt besonders effizient verarbeiten kann (z. B. 32-Bit oder 64-Bit). Auch *Datenbus* und *Cache-Zeilengrößen* begünstigen das Holen/Speichern ganzer Bytes-, Wort- oder Mehrfach-Wort-Blöcke. Moderne CPUs können zusätzlich mit Vektor/SIMD-Einheiten noch größere Pakete (z. B. 128/256/512 Bit) auf einmal verarbeiten.

Software/Betriebssystem: Damit die Hardwarebreite genutzt werden kann, muss das *Betriebssystem* die Architektur unterstützen (32-Bit- oder 64-Bit-Modus, Treiber, Systembibliotheken, ABI). Ein 64-Bit-Prozessor entfaltet seine Vorteile erst vollständig mit einem 64-Bit-Betriebssystem und passenden Programmen; andernfalls arbeitet er im 32-Bit-Kompatibilitätsmodus.

Beispiele.

- **32-Bit-System:** Die CPU arbeitet effizient mit 32-Bit-Wörtern (z. B. `int32`); Zeiger/Adressen sind 32 Bit breit. Daten werden häufig in 32-Bit-Schritten geladen/geschrieben, obwohl der Speicher byteweise adressiert wird.
- **64-Bit-System:** Register und Zeiger sind 64 Bit breit. Das System kann größere Zahlenbereiche adressieren und pro Schritt breitere Daten verarbeiten; trotzdem bleiben Bytes (8 Bit) die kleinste adressierbare Einheit.

1.5 Größe der Daten

1 k	=	1024 Bit	=	2^{10}	(k = Kilo)
1 M	=	1024×1024 Bit	=	2^{20}	(M = Mega)
1 G	=	$1024 \times 1024 \times 1024$ Bit	=	2^{30}	(G = Giga)
1 T	=	$1024 \times 1024 \times 1024 \times 1024$ Bit	=	2^{40}	(T = Tera)
1 P	=	$1024 \times 1024 \times 1024 \times 1024 \times 1024$ Bit	=	2^{50}	(P = Peta)
1 E	=	$1024 \times 1024 \times 1024 \times 1024 \times 1024 \times 1024$ Bit	=	2^{60}	(E = Exa)
1 Z	=	$1024 \times 1024 \times 1024 \times 1024 \times 1024 \times 1024 \times 1024$ Bit	=	2^{70}	(Z = Zetta)
1 Y	=	$1024 \times 1024 \times 1024 \times 1024 \times 1024 \times 1024 \times 1024 \times 1024$ Bit	=	2^{80}	(Y = Yotta)

1.6 Textdarstellung

Von ASCII zu Unicode — warum überhaupt?

Frühe Computersysteme nutzten **ASCII** (American Standard Code for Information Interchange, 1960er Jahre). ASCII ist ein 7-Bit-Zeichensatz mit 128 Zeichen: lateinische Grundbuchstaben A–Z/a–z, Ziffern, Satzzeichen und Steuerzeichen (z. B. Zeilenumbruch). Für englischen Text genügte das, *aber*: Umlaute (ä, ö, ü), Akzente (é), das Euro-Zeichen (€), kyrillisch (Ж), griechisch (Ω), arabisch (م), asiatische Schriften (日), Emojis (😊) *fehlten*. „é“ (NFC) vs. „é“ (NFD) bzw. „e + ◌́“. *fehlten*.

Als Übergang entstanden viele **„erweiterte ASCII“-Codepages** (8-Bit, 256 Zeichen), z. B. `ISO-8859-1`, `Windows-1252`, `KOI8-R`. Jede deckte *einen* Sprachraum ab. Ergebnis: Inkompatibilitäten (sogenanntes *Mojibake*), weil dieselben Bytewerte je nach Codepage andere Zeichen bedeuten.

Unicode (seit 1991) löst dieses Grundproblem: *ein* weltweiter Zeichensatz für *alle* Schriftsysteme, Symbole, technische Zeichen und Emojis. Ziel: *„Jedem Zeichen ein eindeutiger Codepunkt“* — unabhängig von Sprache, Plattform oder Anwendung.

Begriffe sauber trennen

- **Codepunkt** (Unicode): eine Nummer in der Form `U+XXXX` (z. B. `U+00E4` = „ä“, `U+20AC` = „€“, `U+1F60A` = „□“).
- **Kodierung** (Encoding): konkrete Regel, wie Codepunkte in *Bytes* umgesetzt werden (z. B. UTF-8, UTF-16, UTF-32).
- **Graphem-Cluster**: was der Mensch als „ein Zeichen“ wahrnimmt (z. B. „e“ + kombinierender Akzent `U+0301` → „é“, oder Familien-Emoji aus mehreren Codepunkten).

Wichtige Unicode-Kodierungen

- **UTF-8** (variabel, 1–4 Byte pro Codepunkt): ASCII-Zeichen bleiben 1 Byte (abwärtskompatibel), alle anderen werden als 2–4 Byte kodiert. Heutzutage Standard im Web, in Dateien und Schnittstellen.

- **UTF-16** (variabel, 2 oder 4 Byte): Basis-Mehrsprachige Ebene (BMP) meist 2 Byte; Supplementärzeichen (z. B. viele Emojis) als Surrogatpaare (4 Byte). Achtung auf Byte-Reihenfolge (*Endianness*) und optionales *BOM*.
- **UTF-32** (fix 4 Byte): einfacher, aber speicherintensiv; praktisch v. a. intern in manchen Systemen.

Historische Entwicklung in Kürze

1963 ASCII (7-Bit) standardisiert Grundzeichen und Steuerzeichen.

1980er Viele 8-Bit-Codepages (`ISO-8859-x`, `Windows-125x`) — regionale Lösungen, wenig kompatibel.

1991+ Unicode-Projekt: *ein* universeller Zeichensatz; Trennung von *Zeichen* (Codepunkte) und *Kodierung* (UTFs).

2000er+ UTF-8 setzt sich global durch (Internet, Linux/Unix, moderne Apps und Protokolle).

Wie wird Unicode praktisch genutzt?

- **Dateien und Protokolle:** Textdateien, JSON, HTML, E-Mails, Datenbanken — fast überall ist UTF-8 üblich. Wichtig: *Encoding angeben* (z. B. HTTP `Content-Type`, HTML `<meta charset=ütf-8»`, DB-Kollation).
- **Betriebssysteme:** Dateinamen und Konsolen sind (je nach System) Unicode-fähig; moderne Terminals verstehen UTF-8.
- **Programmiersprachen:** Python, Java, JavaScript, C# u. a. arbeiten intern mit Unicode-Zeichenketten; I/O nutzt meist UTF-8.

Kapitel 2

Zahlensysteme

2.1 Was ist ein Zahlensystem?

Ein **Zahlensystem** legt fest, wie Zahlen durch *Ziffern* und deren *Stellenwert* dargestellt werden. In *Stellenwertsystemen* (positionalen Systemen) hat jede Stelle einen Wert, der von der *Basis* b abhängt:

$$(d_k d_{k-1} \ldots d_1 d_0)_b = d_k \cdot b^k + d_{k-1} \cdot b^{k-1} + \ldots + d_1 \cdot b^1 + d_0 \cdot b^0,$$

wobei $0 \leq d_i < b$ gilt. Beispiele: Dezimal ($b = 10$), Binär ($b = 2$), Oktal ($b = 8$), Hexadezimal ($b = 16$). Nicht-positionale Systeme (z. B. römische Zahlen) kennen keinen einheitlichen Stellenwert und sind für Rechenalgorithmen unpraktisch.

2.2 Dezimalsystem ($b = 10$)

Das **Dezimalsystem** nutzt die Ziffern 0–9. Es ist heute *weltweit das dominante System für den Alltag und das schulische Rechnen*. Historisch hängt das vermutlich mit dem Zählen an zehn Fingern zusammen. Beispiel:

$$(5073)_{10} = 5 \cdot 10^3 + 0 \cdot 10^2 + 7 \cdot 10^1 + 3 \cdot 10^0.$$

2.3 Andere Zahlensysteme in der Welt

Sexagesimalsystem ($b = 60$)

Das **Babylonische Sexagesimalsystem** prägt uns bis heute: *Zeitmessung* ($60\,\text{s} = 1\,\text{min}$, $60\,\text{min} = 1\,\text{h}$) und *Winkelmaße* (Grad–Bogenmaß mit Minuten und Sekunden). Rechnen erfolgt im Alltag dennoch meist dezimal; die Einteilung selbst ist aber sexagesimal.

Vigesimalsystem ($b = 20$)

In Teilen der Welt gab und gibt es **Zwanzigersysteme** (Basis 20). Sprachliche Spuren finden sich z. B. in Zahlwörtern einiger Sprachen (Restbestände wie „vier-mal-zwanzig“ für 80). Auch hier wird formal in der Schule und in modernen Anwendungen überwiegend dezimal gerechnet.

Duodezimalsystem ($b = 12$)

Das **Zwölfersystem** hat gute Teilbarkeit (2,3,4,6). Reste davon sieht man bei Dutzend/Groß, Uhren (12 Stunden), Maßeinheiten aus der Geschichte. Für maschinelles oder schulisches Rechnen dominiert aber 10.

Fazit zur Frage: „Rechnet man irgendwo ernsthaft nicht-dezimal?“

Menschen rechnen heute fast überall *dezimal*, mit kulturellen Resten anderer Basen in speziellen Domänen (Zeit, Winkel, Maße). **Maschinen** (Computer) *rechnen binär*. Darauf basiert die Notwendigkeit weiterer Basen in der Informatik (Oktal/Hex als kompakte Binärdarstellung).

2.4 Binär, Oktal, Hexadezimal – warum in der Informatik?

Binärsystem ($b = 2$)

Digitale Schaltungen kennen zwei stabile Zustände (z. B. „aus“/„ein“). Deshalb arbeitet Hardware *binär*.

$$(101010)_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 42.$$

Oktalsystem ($b = 8$) und Hexadezimalsystem ($b = 16$)

Beide sind für Menschen *kompakte Schreibweisen* von Binärzahlen:

- 1 **Oktal-Ziffer** entspricht **3 Bit** (weil $8 = 2^3$).
- 1 **Hex-Ziffer** entspricht **4 Bit** (weil $16 = 2^4$).

Darum lassen sich Binärzahlen leicht gruppieren:

$$\underbrace{1010}_{\text{A}}\ \underbrace{1010}_{\text{A}} = (\text{AA})_{16} = (10101010)_2.$$

Hex ist heute Standard in Programmierung, Speicher-Dumps, Farbwerten (#FF00AA), Adressen usw. Oktal sieht man u. a. noch bei UNIX-Rechten (z. B. 0755).

2.5 Umrechnungen zwischen Basen

Von Dezimal in eine Basis b (Divisionsrest-Methode)

Beispiel: 93_{10} nach Binär.

$$\begin{array}{r|l} 93 : 2 = 46 \text{ Rest } 1 & \\ 46 : 2 = 23 \text{ Rest } 0 & \\ 23 : 2 = 11 \text{ Rest } 1 & \\ 11 : 2 = 5 \text{ Rest } 1 & \Rightarrow\ (93)_{10} = (1011101)_2. \\ 5 : 2 = 2 \text{ Rest } 1 & \\ 2 : 2 = 1 \text{ Rest } 0 & \\ 1 : 2 = 0 \text{ Rest } 1 & \end{array}$$

Von Basis b nach Dezimal (Horner-Schema)

Beispiel: $(2A)_{16}$ mit $A = 10$:

$$(2A)_{16} = 2 \cdot 16^1 + 10 \cdot 16^0 = 32 + 10 = 42.$$

Direkt zwischen Binär, Oktal, Hex

Gruppieren in 3er- bzw. 4er-Blöcke (von rechts):

$$(110\,010\,111)_2 = (627)_8, \qquad (1010\,1111)_2 = (\mathrm{AF})_{16}.$$

2.6 Beispiele

Dezimal	Binär	Oktal	Hex
10	1010	12	A
26	11010	32	1A
42	101010	52	2A
64	1000000	100	40
100	1100100	144	64

Kleiner Blick über den Tellerrand.
Auch andere Basen sind möglich und wurden erprobt (z. B. Ternär $b = 3$). Für die heutige Praxis gilt: Menschen bevorzugen $b = 10$, Computer arbeiten in $b = 2$; Oktal/Hex dienen als menschenfreundliche Brücke zur Binärwelt, Sexagesimal und Zwanziger-/Zwölfersysteme leben in speziellen Domänen weiter.

2.7 Die Zweierkomplementdarstellung

2.7.1 Worum geht es?

Computer speichern Zahlen als Bitmuster. Für *positive* ganze Zahlen ist das einfach (normale Binärdarstellung). Aber wie speichern wir *negative* Zahlen so, dass Addieren und Subtrahieren trotzdem mit der *gleichen Hardware* funktionieren? Die Antwort ist die **Zweierkomplementdarstellung**.

2.7.2 Warum verwendet man das Zweierkomplement?

- **Ein Addierwerk für alles:** Dieselbe Schaltung addiert sowohl positive als auch negative Zahlen; Subtraktion wird als „Addiere das Zweierkomplement“ ausgeführt.
- **Eindeutige Null:** Es gibt nur *eine* Null (anders als bei Vorzeichen-&-Betrag oder Einerkomplement).
- **Einfache Regeln:** Vorzeichenverlängerung (Sign Extension) ist trivial: führende Einsen bei negativen Zahlen, Nullen bei positiven.

- **Sortier-/Vergleichsfreundlich:** Bei festem Wortbreite-Vergleich funktioniert das wie erwartet.
- **Mathematisch sauber:** Der Wertebereich ist genau $-2^{n-1}, \ldots, 0, \ldots, 2^{n-1}-1$ für n Bit.

2.7.3 So bildet man das Zweierkomplement (aus einer positiven Zahl x)

Für eine feste Wortbreite (z. B. 8 Bit):

1. Schreibe x binär mit führenden Nullen.
2. **Alle Bits invertieren** ($0 \leftrightarrow 1$).
3. **+1 addieren.**

Beispiel: -5 als 8-Bit-Zahl.
$+5 = \texttt{00000101} \Rightarrow$ invertiert $\texttt{11111010} \Rightarrow +1 \Rightarrow \boxed{\texttt{11111011}}$.

So liest man ein Zweierkomplement-Bitmuster

- **MSB (linkestes Bit) = 0** $\Rightarrow$ positive Zahl: normal als Binärzahl lesen.
- **MSB = 1** $\Rightarrow$ negative Zahl: wieder positiv machen durch *invertieren + 1* und ein Minus davor.

Beispiel: $\texttt{11101100}$ (8 Bit) $\Rightarrow$ invertieren $\texttt{00010011}$, $+1 \Rightarrow \texttt{00010100} = 20 \Rightarrow$ Wert ist -20.

Wertebereich

Für n Bit gilt:

$$\boxed{-2^{n-1} \;\leq\; \text{Wert} \;\leq\; 2^{n-1}-1} \quad \text{(z. B. 4 Bit: } -8 \text{ bis } +7\text{)}.$$

Auffällig: Es gibt *kein* $+2^{n-1}$ (bei 4 Bit also kein $+8$); das Muster $\texttt{1000}$ steht für -8.

Rechnen mit Zweierkomplement

Addition/Subtraktion.

- Subtraktion $a - b$ wird als $a +$ (Zweierkomplement von b) gerechnet.
- *Beispiel (4 Bit):* $7 + (-3)$: $\texttt{0111} + \texttt{1101} = \texttt{1 0100} \Rightarrow \texttt{0100} = 4$ (Übertrag links fällt weg).

Überlauf (Overflow) erkennen.

- **Regel:** Addiert man zwei *positive* Zahlen und erhält eine *negative*, oder zwei *negative* und erhält eine *positive*, dann ist Overflow aufgetreten.
- Alternativ technisch: Overflow, wenn *Carry in* das Vorzeichenbit $\neq$ *Carry out* des Vorzeichenbits.

Vorzeichenverlängerung (Sign Extension).

Erweitere eine Zweierkomplementzahl auf mehr Bit, indem du die *linke führende Ziffer* wiederholst:

- positiv: `0`en voran (z. B. `0010` $\rightarrow$ `0000 0010`)
- negativ: `1`en voran (z. B. `1110` $\rightarrow$ `1111 1110`)

Vergleich zu anderen Darstellungen

Vorzeichen & Betrag: Einfach zu verstehen (ein Bit fürs Vorzeichen), aber zwei Nullen ($+0$ und -0) und Subtraktion ist umständlich.

Einerkomplement: Auch zwei Nullen und kompliziertere Addition (End-Around-Carry).

Zweierkomplement: Standard in nahezu allen modernen CPUs – schnell, eindeutig, hardwarefreundlich.

Beispiele (4-Bit)

Bitmuster	Dezimal	Bitmuster	Dezimal
0111	$+7$	1001	-7
0101	$+5$	1011	-5
0000	0	1111	-1
1000	-8	1101	-3

Abbildung

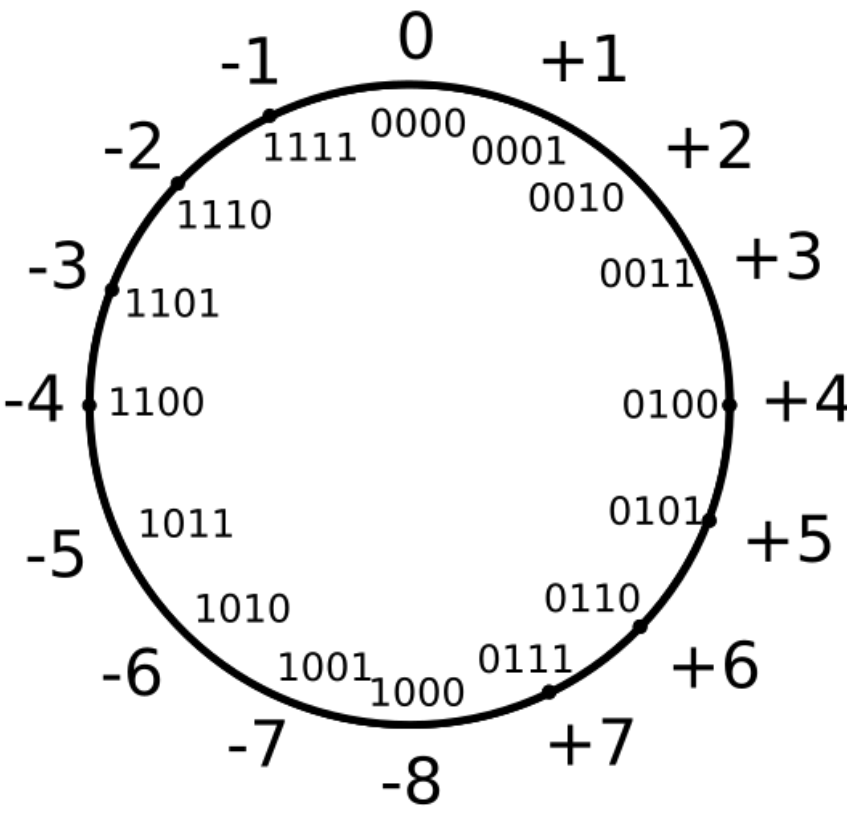

Abbildung 2.1: 4-Bit-Zweierkomplement: Zuordnung der Bitmuster zu Dezimalwerten und Bildung der negativen Werte ($+1$ nach Bitinvertierung).

Kapitel 3

[Thema 1]

3.1 Grundlagen

Hier beginnen wir mit deinem ersten Thema.

Kapitel 4

[Thema 2]

...

Anhang

Quellen, Glossar, etc.