Kapitel 1

Zahlensysteme

1.1 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.

1.2 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:

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

1.3 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).

1.4 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).

1.5 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):

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

1.6 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.

1.7 Die Zweierkomplementdarstellung

1.7.1 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**.

1.7.2 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.

1.7.3 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

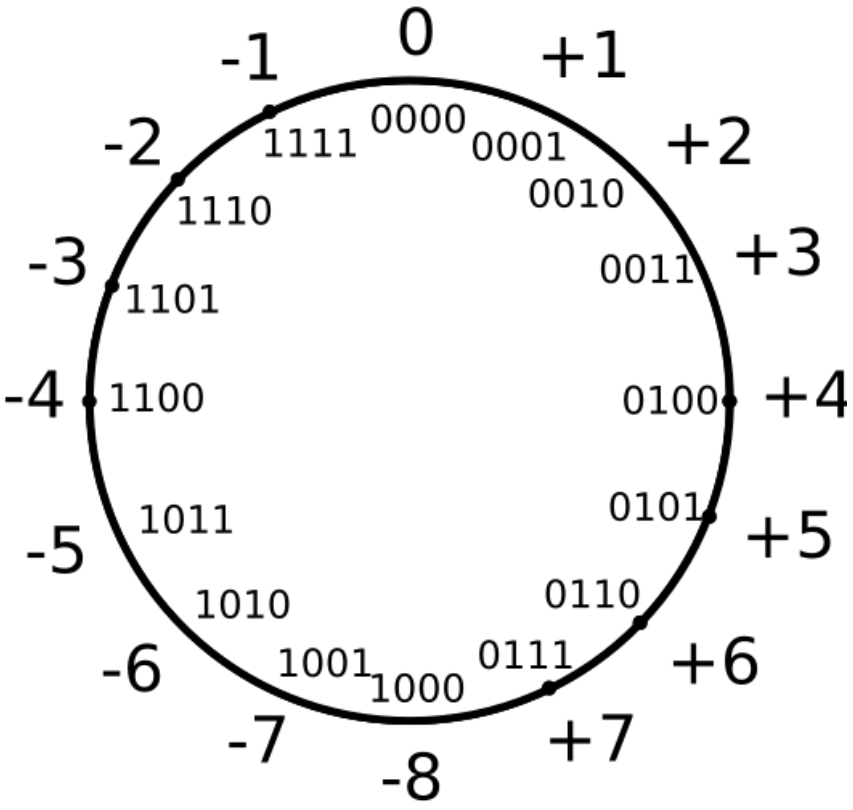

Abbildung 1.1: 4-Bit-Zweierkomplement: Zuordnung der Bitmuster zu Dezimalwerten und Bildung der negativen Werte ($+1$ nach Bitinvertierung).