Informatik—Skript für die E-Phase

[Jarek Mycan]

10. September 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:

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

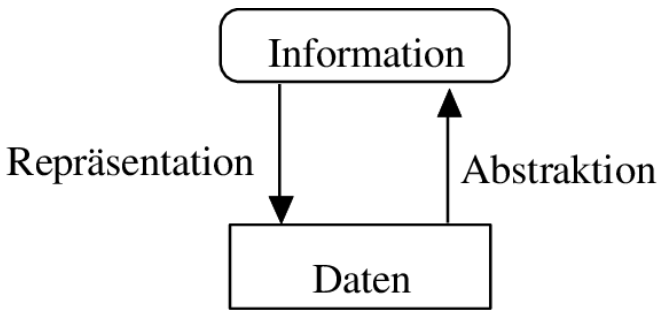


Abbildung 1.1: Wechselspiel zwischen Repräsentation (Information Daten) und Abstraktion (Daten Information).

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

Mensch und Computer

2.1 Worum geht es?

Seit Jahrtausenden nutzt der Mensch Technik, um Arbeit zu erleichtern. Computer sind die logische Fortsetzung: Sie verarbeiten Daten in einer Geschwindigkeit und Zuverlässigkeit, die für Menschen unerreichbar ist. Aber: Computer „verstehen“ nichts – sie führen *Programme* aus. Dieses Kapitel vergleicht verständlich, wie *Menschen* und *Computer* mit Daten umgehen.

2.2 Was ist Datenverarbeitung?

Daten sind Zeichen bzw. Messwerte; *Information* entsteht erst durch *Deutung* im Kontext (vgl. Repräsentation/Abstraktion, siehe Abbildung 1.1 in Kap. Kapitel 1).

- **Eingabe:** Daten aufnehmen (Sinnesorgane, Sensoren, Tastatur …)
- **Verarbeitung:** Ordnen, Vergleichen, Rechnen, Entscheiden
- **Speicherung:** Merken (Gedächtnis) bzw. Speichermedien
- **Ausgabe:** Handeln, Sprechen, Anzeigen, Drucken

2.3 Wie verarbeitet der Mensch Daten?

Menschen nehmen Daten z. B. über *Augen* und *Ohren* auf. Im Gehirn laufen typische Denkoperationen:

- **Ordnen & Prüfen:** Passt das zu Bekanntem? Ist eine Schreibweise „richtig“?
- **Vergleichen & Kontrollieren:** Stimmen Werte überein? Ist ein Ergebnis plausibel?
- **Kombinieren & Schlussfolgern:** Aus Bekanntem Neues ableiten.

Menschen speichern intern (*Gedächtnis*) und extern (*Notizen, Bücher*). Externe Speicherung entlastet, ist teilbar und dauerhaft.

2.4 Wie verarbeitet der Computer Daten?

Computer sind *datenverarbeitende Maschinen*. Ihre Grundteile:

- **Eingabe** (z. B. Tastatur, Maus, Sensoren, Scanner)
- **Zentraleinheit (CPU)** mit *Steuerwerk* und *Rechenwerk (ALU)*; arbeitet streng nach dem *Fetch–Decode–Execute*-Zyklus
- **Speicher** (intern: Register, RAM; extern: Massenspeicher)
- **Ausgabe** (z. B. Bildschirm, Lautsprecher, Drucker)

Historisch sprach man von *EDV* (Elektronische Datenverarbeitung). Heute sind die Prinzipien gleich geblieben – nur viel schneller und mit größerem Speicher.

2.5 Mensch vs. Computer – ein Vergleich

Aspekt	**Mensch**	**Computer**
Aufnahme	Sinne (sehen, hören)	Geräte/Sensoren, Schnittstellen
Tempo/Genauigkeit	langsam, fehleranfällig, aber flexibel	extrem schnell, exakt, wiederholgenau
Deutung	versteht Bedeutung, kann Kontext herstellen	versteht nicht; verarbeitet Bitmuster
Kreativität/Lernen	kreativ, lernt aus Erfahrungen	braucht Programm/Trainingsdaten
Speicherung	Gedächtnis (vergesslich) + Notizen	Speicherhierarchie (Cache–RAM–SSD)
Energie	effizient, benötigt Pausen	dauerhaft, braucht Strom

2.6 Warum braucht ein Computer ein Programm?

Ein Computer „weiß“ nicht, was mit Daten zu tun ist. Erst ein *Programm* (Rezept aus Befehlen) sagt: *Welche Daten? In welcher Reihenfolge? Mit welchen Operationen?* Beispiel Lohnabrechnung:

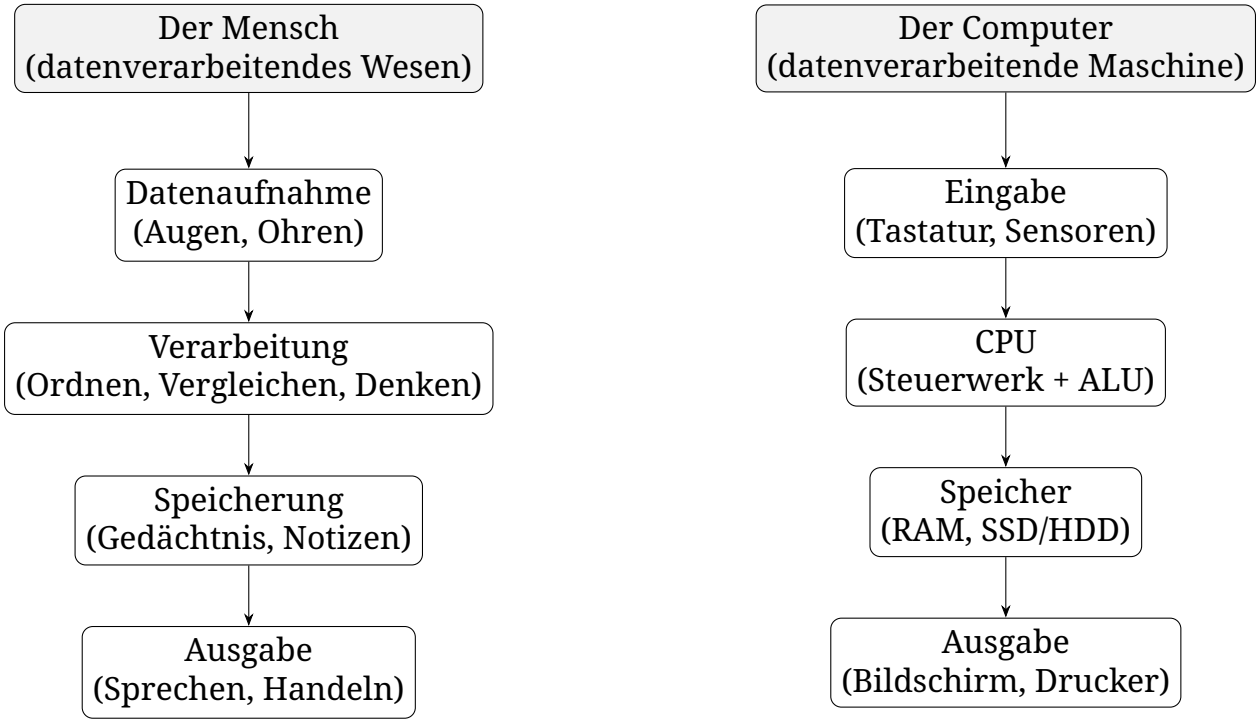
1. **Daten** bereitstellen (Namen, Stunden, Sätze, Abzüge).
2. **Programm** laden (Vorschrift, wie zu rechnen ist).
3. **Ablauf** (vereinfacht): `LOAD` Daten → `RECHNE` → `STORE` Ergebnis.

Ohne Programm passiert nichts – oder das Falsche.

2.7 Wichtige Einsicht: „Garbage in, garbage out“

Fehlen Daten oder sind sie falsch, ist auch das Ergebnis falsch – egal wie schnell der Rechner ist. Beispiel aus dem Alltagstext: Eine Mausefalle „arbeitet nach Programm“, aber wenn *wichtige Information fehlt* (z. B. Ort), bleibt der Erfolg aus.

2.8 Ein einfaches Gesamtbild



2.9 Merksätze

- Menschen *deuten* Daten zu Information; Computer *verarbeiten* Daten nach Programm.
- Ohne Programm keine Verarbeitung; ohne passende Daten kein sinnvolles Ergebnis.
- Externe Speicherung (Notizen/Dateien) erweitert, was intern gemerkt werden kann.

Kapitel 3

Zahlensysteme

3.1 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.

3.2 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:

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

3.3 Andere Zahlensysteme in der Welt

Sexagesimalsystem ($b = 60$)

Das **Babylonische Sexagesimalsystem** prägt uns bis heute: *Zeitmessung* (60 s = 1 min, 60 min = 1 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).

3.4 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).

3.5 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):

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

3.6 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

3.7 Die Zweierkomplementdarstellung

3.7.1 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**.

3.7.2 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.

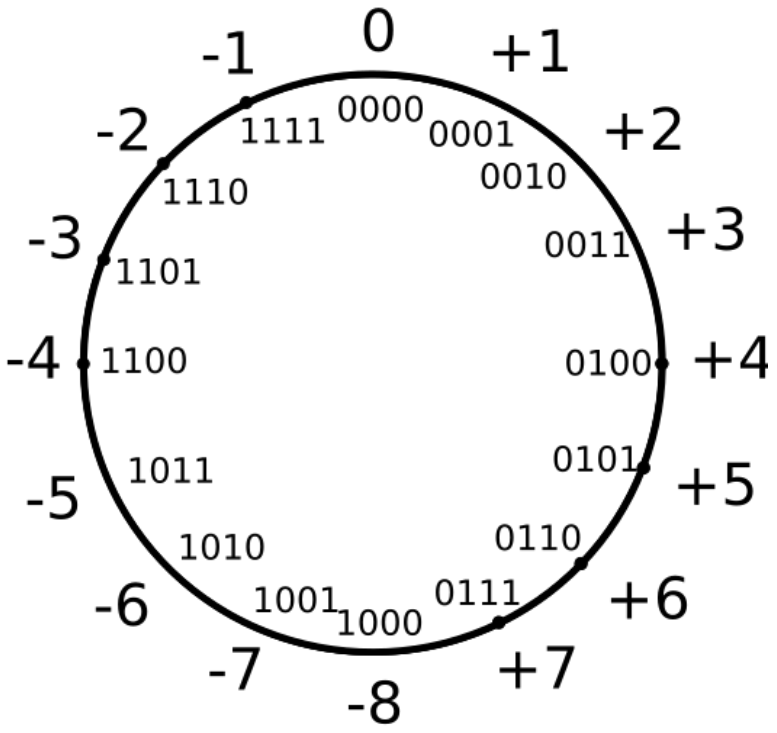


Abbildung 3.1: 4-Bit-Zweierkomplement: Zuordnung der Bitmuster zu Dezimalwerten und Bildung der negativen Werte ($+1$ nach Bitinvertierung).

3.7.3 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.
$+5 = 00000101 \Rightarrow$ invertiert $11111010 \Rightarrow +1 \Rightarrow \boxed{11111011}$.

So liest man ein Zweierkomplement-Bitmuster

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

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

Wertebereich

Für n Bit gilt:

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

Auffällig: Es gibt *kein* $+2^{n-1}$ (bei 4 Bit also kein $+8$); das Muster `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

Wie rechnet ein Computer wirklich? — Ausblick

Das Umwandeln von Zahlen und das schriftliche Rechnen zeigen *wie* wir mit Binärzahlen umgehen. Ein Computer macht im Kern dasselbe — aber nicht „im Kopf“, sondern mit **logischen Operationen**, die in **elektronischen Schaltungen** realisiert sind. Grundlage ist die Boolesche Algebra: Aus einfachen Verknüpfungen wie NICHT (NOT), UND (AND), ODER (OR) und EXKLUSIV-ODER (XOR) werden Schaltungen aufgebaut, die Bits verarbeiten. Ein Addierer besteht beispielsweise aus *Halb-* und *Volladdierern*: Das *Summenbit* entsteht über XOR (gerade/ungerade Anzahl von Einsen), der *Übertrag* über AND/OR (Mehrheit der Einsen). Darum ist bei $1+1$ das Summenbit 0 und der Übertrag 1 — genau wie in unseren schriftlichen Regeln.

Viele solcher Bausteine zusammen bilden die **ALU** (Arithmetic Logic Unit) einer CPU, gesteuert von einem **Takt**. **Register** und **Speicherzellen** (Flip-Flops) halten Zwischenwerte, und durch die feste Wortbreite rechnet die Hardware effektiv $\bmod 2^n$ — daher kommen Phänomene wie *Übertrag* und *Overflow*. Auf höherer Ebene beschreibt **Software** die Abfolge dieser elementaren Operationen als Befehle und Algorithmen.

Mit diesen inneren Mechanismen — Logikgattern, Schaltnetzen und Schaltwerken, Transistoren und CMOS-Technik, aber auch endlichen Automaten — beschäftigen wir uns später im Kurs. Für jetzt reicht die Idee: Was wir schriftlich üben, setzt der Computer *gleichbedeutend* durch logische Funktionen und Elektronik um.

Kapitel 4

Hardwarearchitektur

4.1 Worum geht es?

Unter **Hardwarearchitektur** versteht man den grundsätzlichen Aufbau eines Rechners: Welche Bausteine gibt es (z. B. Prozessor, Speicher, Ein-/Ausgabe), wie sind sie *organisiert* und *verbunden*, und wie arbeiten sie zusammen, um Programme auszuführen? Die heute dominierende Grundidee ist die **von-Neumann-Architektur**.

4.2 John von Neumann und die Grundidee

In den 1940er Jahren formulierte John von Neumann (gemeinsam mit weiteren Pionieren um ENIAC/EDVAC) eine einfache, aber revolutionäre Idee: **Programm und Daten liegen im gleichen Speicher.** Das heißt, ein Programm ist selbst nur eine Folge von Zahlen (Maschinenbefehlen), die genau wie Daten im Hauptspeicher abgelegt und von der CPU geholt werden. Diese *Stored-Program*-Idee macht Rechner *flexibel* (beliebige Programme ladbar) und *universell*.

Bausteine im von-Neumann-Modell

- **CPU (Prozessor)** mit
  - **Steuerwerk** (kontrolliert den Ablauf, interpretiert Befehle),
  - **Rechenwerk/ALU** (führt Operationen wie Addieren, Vergleichen aus),
  - **Registern** (kleinste, sehr schnelle Speicherplätze, z. B. Akkumulator, Program Counter).
- **Hauptspeicher (RAM)**: enthält *sowohl* Daten *als auch* Befehle.
- **Ein-/Ausgabe (I/O)**: Tastatur, Bildschirm, Netz, Sensoren, Aktoren …
- **Bus-System**: Verbindet die Bausteine (Adress-, Daten- und Steuerbus).

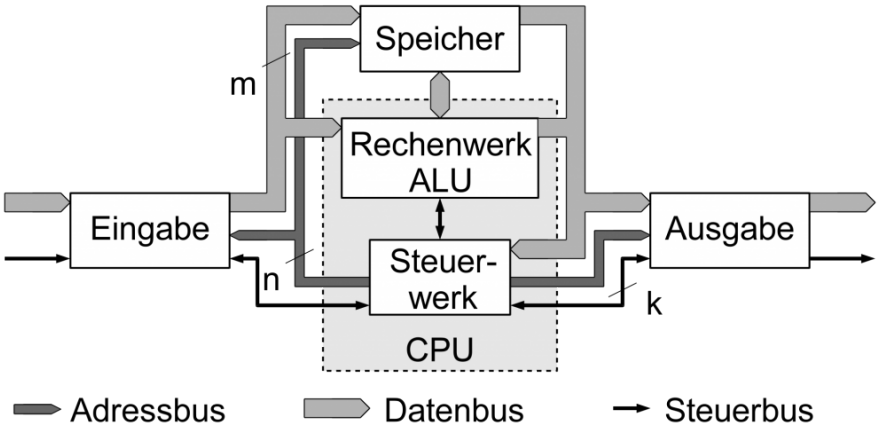


Abbildung 4.1: Architektur von Neumann Rechners.

4.3 Der Befehlszyklus (Fetch–Decode–Execute)

Jeder Maschinenbefehl läuft in drei Schritten durch die CPU:

1. **Fetch** (Holen): Der *Program Counter (PC)* zeigt auf die nächste Befehlsadresse. Der Befehl wird aus dem Speicher gelesen und in das *Befehlsregister (IR)* gelegt.

2. **Decode** (Dekodieren): Das Steuerwerk „versteht“, welcher Operationstyp gemeint ist (z. B. ADD, LOAD, STORE) und welche Operanden/Adressen beteiligt sind.

3. **Execute** (Ausführen): Die ALU rechnet bzw. I/O/Speicherzugriffe passieren. Der PC wird auf den nächsten Befehl gesetzt (oder bei Sprüngen angepasst).

Mini-Beispiel (gedankliches Maschinenprogramm).

```
1         LOAD R0, [x]    ; lade x in Register R0
2         LOAD R1, [y]    ; lade y in Register R1
3         ADD  R0, R1     ; R0 := R0 + R1
4         STORE R0, [z]   ; speichere Ergebnis in z
```

Listing 4.1: Addition zweier Speicherstellen und Ablage des Ergebnisses

Hier holt die CPU nacheinander die Befehle (Fetch), dekodiert sie (Decode) und führt sie aus (Execute).

4.4 Speicher, Wortbreite und Adressierung

- **Wortbreite** (z. B. 32 Bit, 64 Bit) gibt an, wie viele Bits die CPU in einem Schritt besonders effizient verarbeitet (Register- und ALU-Breite). Sie beeinflusst u. a. den darstellbaren Adressraum und Zahlenbereich.

- **Adressbus/Datenbus**: Mit n Adressleitungen kann man 2^n Speicheradressen ansprechen.

- **Speicherhierarchie**: Register → Caches (L1/L2/L3) → RAM → SSD/HDD. Je näher an der CPU, desto schneller (aber kleiner/teurer).

4.5 Warum ist das so erfolgreich?

- **Einfachheit**: Ein einheitlicher Speicher für Programme und Daten macht die Hardware und das Laden von Programmen einfach.

- **Flexibilität**: Beliebige Programme können nachgeladen werden; Selbstmodifizierender Code ist (theoretisch) möglich.

- **Universalität**: Mit genug Speicher und Zeit kann ein solcher Rechner jede berechenbare Aufgabe lösen (Church–Turing-Idee).

4.6 Grenzen: der Von-Neumann-Flaschenhals

Weil *Programm* und *Daten* über *denselben* Speicher-/Busweg kommen, konkurrieren sie um Bandbreite. Das bremst: Die CPU könnte schneller rechnen, als Daten/Befehle nachgeliefert werden. Was hilft dagegen?

- **Caches (Zwischenspeicher):** Kleine, sehr schnelle Speicher *in der CPU*, die oft gebrauchte Daten/Befehle bereithalten.

- **Vorabruf (Prefetch):** Die CPU *holt schon vorher*, was sie gleich brauchen wird.

- **Pipelining (Fließband):** Eine Instruktion wird in Schritte zerlegt; mehrere Instruktionen sind gleichzeitig in *verschiedenen* Schritten.

- **Superskalar (mehrere Einheiten):** Die CPU kann *mehr als einen* Befehl pro Takt starten.

- **Mehr Kerne (Multi-Core):** Mehrere Rechenkerne arbeiten *parallel* an unterschiedlichen Aufgaben.

- **Vektoreinheiten (SIMD):** *Eine* Rechenanweisung auf *viele* Daten gleichzeitig ausführen.

- **Breitere/ schnellere Verbindungen:** Mehr Bits pro Takt (*breiter Bus*) und schnellerer Speicher (z. B. DDR, HBM) liefern Daten zügiger.

Merksatz: Wenn die CPU warten muss, helfen *nähere Vorräte* (Cache), *rechtzeitig holen* (Prefetch), *gleichzeitig arbeiten* (Pipeline/parallel), und *breitere/-schnellere Wege* (Bus/Speicher).

4.7 Harvard vs. von Neumann (und die Praxis heute)

Die **Harvard-Architektur** trennt *Befehls-* und *Datenspeicher* (je eigener Bus). Vorteil: Gleichzeitige Zugriffe, kein Flaschenhals an dieser Stelle. Viele *Mikrocontroller/DSPs* und auch *moderne CPUs intern* nutzen eine **modifizierte Harvard-Architektur**: z. B. getrennte *Instruktions-* und *Datencaches*, obwohl der *Hauptspeicher* gemeinsam ist. Damit kombiniert man die Programmierfreundlichkeit des von-Neumann-Modells mit Leistungsgewinnen.

Kapitel 5

Grundlagen der Computernetze

5.1 Warum vernetzen wir Computer?

Ein einzelner Computer ist nützlich — richtig spannend wird es erst, wenn Geräte *miteinander* Daten austauschen: Wir verschicken Nachrichten, teilen Dateien, streamen Videos, rufen Webseiten ab. Ein **Computernetz** verbindet Geräte (Hosts), damit sie Informationen austauschen können: im **LAN** (z. B. Schule, Zuhause), im **WAN** (Verbindung über große Distanzen) und letztlich im **Internet** — dem Netz der Netze.

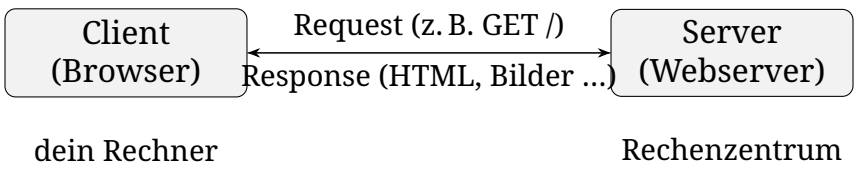
5.2 Client und Server

Viele Anwendungen folgen dem **Client–Server**-Modell:

- Der **Client** stellt eine Anfrage (Request) — z. B. dein Browser.
- Der **Server** antwortet (Response) — z. B. der Webserver mit der HTML-Seite.

Es gibt häufig *viele* Clients, aber *wenige* zentrale Server. Beispiele:

- **Web:** Browser (Client) ↔ Webserver (HTTP/HTTPS).
- **E-Mail:** Mail-App (Client) ↔ Mailserver (SMTP/IMAP/POP3).
- **Dateien in der Schule:** PCs (Clients) ↔ Schul-Fileserver.



Merksatz.
Client fragt, Server antwortet. Das kann sehr schnell gehen — oft in Millisekunden —, weil viele Anfragen parallel bearbeitet werden.

5.3 Datenpakete (allgemein)

Große Dateien oder Webseiten werden im Netz **in viele kleine Stücke** zerlegt — die **Pakete**. Jedes Paket enthält:

- einen **Kopf** (*Header*) mit Steuerinformationen (z. B. Absender-Adresse, Empfänger-Adresse, Paket-Nummer) und
- die **Nutzdaten** (*Payload*), also ein Stück der eigentlichen Information.

Pakete gehen nicht immer denselben Weg; Router entscheiden unterwegs *Paket für Paket*, welcher Pfad gerade passt. Am Ziel setzt die Anwendung die Teile wieder korrekt zusammen.

Ein Datenpaket: Steuerinfos + Nutzdaten

Warum Pakete?
Kleine Einheiten lassen sich

- effizient weiterleiten (Router können schnell entscheiden),
- bei Fehlern *gezielt* neu senden (nicht die ganze Datei),
- auf mehreren Wegen gleichzeitig schicken (Last verteilen).

5.4 Adressierung von Computern

Damit Pakete ihr Ziel finden, brauchen sie Adressen — ähnlich wie Briefe.

Drei „Adressen“ auf einen Blick

- **MAC-Adresse** (Hardware-Adresse): Kennzeichnet *die Netzwerkkarte*. Wird im lokalen Netz (z. B. im LAN/WLAN) benutzt, um Frames an das richtige Gerät zu schicken.
- **IP-Adresse** (Internet-Adresse): Kennzeichnet *das Gerät im Netz*. Beispiele: IPv4 wie `192.168.1.20`, IPv6 wie `2001:db8::1`. Router benutzen IP-Adressen, um Pakete *durch viele Netze* zum Ziel zu leiten.
- **Portnummer** (Anwendungs-Adresse): Kennzeichnet *den Dienst* auf dem Gerät. Beispiel: Webserver hört oft auf `80` (HTTP) oder `443` (HTTPS); ein Spiel oder Chat kann andere Ports nutzen.

Alltagsbild.

- *IP-Adresse* ≈ Straßenadresse eines Hauses (wohin der Brief soll).
- *Port* ≈ Briefkasten/Abteilung im Haus (wer den Brief entgegennimmt).
- *MAC* ≈ Namensschild an der Netzwerkkarte (auf der letzten Zustellstrecke im LAN).

Namen statt Zahlen: DNS

Menschen merken sich lieber Namen als Zahlen. Das **Domain Name System (DNS)** übersetzt z. B. `www.schule.de` in die zugehörige IP-Adresse. Dein Rechner fragt dazu einen DNS-Server an — ähnlich wie ein Telefonbuch für das Internet.

Anhang

Übersicht der Anhänge

1. **Arbeitsblatt 1: Einführung**
   Bezug: Kapitel 1 — Einführung
   Datei: `aufgabenblatt-1.pdf`

2. **Arbeitsblatt 3.0: Zahlendarstellungen & Zweierkomplement**
   Bezug: Kapitel 3 — Zahlensysteme
   Datei: `aufgabenblatt-3.0.pdf`

3. **Arbeitsblatt 3.1: Zahlendarstellungen & Zweierkomplement**
   Bezug: Kapitel 3 — Zahlensysteme
   Datei: `aufgabenblatt-4.1.pdf`

4. **Arbeitsblatt 4: Hardwarearchitektur**
   Bezug: Punkt — Inhaltsverzeichnis
   Datei: `aufgabenblatt-4.pdf`

Heinrich-von-Kleist-Schule
Eschborn

Fach: Informatik
Kurs: E1
Datum: _____
Name: _____

Arbeitsblatt 1 — Einführung

Thema: Information, Repräsentation, Abstraktion, Bits/Bytes, Textkodierung

---

Bearbeitungshinweise

- Antworte präzise in ganzen Sätzen, wo sinnvoll mit Skizzen/Beispielen.
- Kennzeichne Ergebnisse klar. Rechenschritte und Begründungen angeben.
- Nutze bei Bedarf Quellenangaben (URL, Zugriffstag) für Rechercheaufgaben.

Präsenzaufgaben

Aufgabe 1: Begriff klären: Information vs. Daten. [*6 BE*]
Erkläre mit eigenen Worten den Unterschied zwischen *Daten* und *Information*. Gib **zwei** Beispiele, in denen dieselben Daten je nach Kontext *unterschiedliche* Information bedeuten.

Aufgabe 2: Repräsentation oder Abstraktion? [*8 BE*]
Ordne die folgenden Tätigkeiten zu und begründe jeweils kurz (*Repräsentation* = Information → Daten, *Abstraktion* = Daten → Information):

Aufgabe 2:a) Ein Sensor wandelt Temperatur in eine Zahl in Grad Celsius um.

Aufgabe 2:b) Ein Bildbetrachter zeigt aus einer PNG-Datei ein Foto an.

Aufgabe 2:c) Ein MP3-Encoder erzeugt aus einer WAV-Datei eine komprimierte Datei.

Aufgabe 2:d) Ein Statistiktool erkennt in Messwerten einen Trend.

Aufgabe 3: Bits, Bytes, Wortbreite. [*8 BE*]

Aufgabe 3:a) Warum liest/schreibt die Hardware Daten *gruppenweise*? Nenne zwei Gründe.

Aufgabe 3:b) Erkläre „Wortbreite“ und gib typische Werte an. Was ändert sich beim Übergang von 32-Bit zu 64-Bit?

Aufgabe 3:c) Ein System nutzt 64-Bit-Register, aber der Speicher ist *byteweise* adressierbar. Ist das ein Widerspruch? Begründe.

Aufgabe 4: „Pipeline“ vom Phänomen zur Information. [*8 BE*]
Beschreibe für das Beispiel „Foto mit dem Smartphone“ die Schritte *Messung* → *Repräsentation* → *Verarbeitung* → *Abstraktion* stichpunktartig (Sensor, A/D-Wandlung, Dateiformat, Anzeige/Erkennung …).

Aufgabe 5: Textkodierung – ASCII vs. Unicode. [*10 BE*]

Aufgabe 5:a) Nenne **drei** Zeichen, die in ASCII fehlen, und erkläre, warum verschiedene 8-Bit-Codepages (ISO-8859-1, Windows-1252 …) zu Problemen führten.

Aufgabe 5:b) Was unterscheidet *Codepunkt* und *Kodierung*? Erkläre an einem Beispiel (z. B. Buchstabe „ä“).

Aufgabe 5:c) Worin liegt der Vorteil von UTF-8 gegenüber einer festen 8-Bit-Kodierung?

Hausaufgaben

Aufgabe 1: Recherche: Mojibake in freier Wildbahn. [*8 BE*]
Finde **zwei** reale Beispiele (Screenshot, Link oder kurze Beschreibung), in denen Text *falsch* dargestellt wurde (z. B. „Ã¤“ statt „ä“). Erkläre die Ursache in 2–3 Sätzen (*welche* Kodierung wurde vermutlich geschrieben, *welche* beim Lesen angenommen?).

Aufgabe 2: UTF-8 zum Anfassen. [*10 BE*]
Bestimme die UTF-8-Bytefolgen (hexadezimal) für die Zeichen: `A`, `ä`, `€`. Beschreibe jeweils in 1–2 Sätzen, warum die Länge 1, 2 bzw. 3 Bytes beträgt.

Aufgabe 3: Datenmenge einschätzen. [*8 BE*]
Ein Graustufenbild hat 800×600 Pixel, 8 Bit pro Pixel.

Aufgabe 3:a) Wie groß ist die *unkomprimierte* Datei in Byte/KiB?

Aufgabe 3:b) Wie groß wäre dasselbe Bild als RGB (24 Bit pro Pixel)?

Aufgabe 3:c) Warum kann eine PNG-Datei trotzdem deutlich kleiner sein?

Aufgabe 4: Transferaufgabe: Abstraktion bewusst wählen. [*10 BE*]
Du entwickelst eine App, die Schritte zählt und „Aktivitätslevel“ anzeigt.

Aufgabe 4:a) Welche *Rohdaten* könnten erfasst werden? (mind. 3)

Aufgabe 4:b) Wie würdest du daraus ein *Modell* bauen (welche Features, welche Stufen)?

Aufgabe 4:c) Wo liegen Risiken falscher Abstraktionen?

Bezug: Kapitel 1 „Einführung“. Dieses Blatt vertieft die Inhalte zu Information Daten, Repräsentation/Abstraktion, Bits/Bytes und Textkodierung.

Heinrich-von-Kleist-Schule
Eschborn

Fach: Informatik
Kurs: E1
Datum: _____
Name: _____

Aufgabenblatt 3.0

Thema: Zahlendarstellungen & Zweierkomplement

Bearbeitungshinweise

- Ergebnisse klar kennzeichnen. Rechenschritte nachvollziehbar darstellen.
- Verwende bei Binärzahlen den Index $_2$, bei Hexzahlen $_{16}$, bei Dezimalzahlen $_{10}$.

Präsenzaufgaben

Aufgabe 1: Zahlendarstellung I (Binär). Wandle in die Binärdarstellung um:
a) 55_{10} b) 42_{10} c) 127_{10} d) 73951_{10}. [*8 BE*]

Aufgabe 2: Zahlendarstellung II (Hex). Wandle in die Hexadezimaldarstellung um:
a) 224_{10} b) 69_{10} c) 171_{10} d) 57005_{10}. [*8 BE*]

Aufgabe 3: Zahlenbereiche. Beantworte kurz und begründe:

Aufgabe 3:a) Größte darstellbare Zahl mit 5 Bit (*vorzeichenlos*).

Aufgabe 3:b) Wie viele verschiedene Werte lassen sich mit 32 Bit darstellen?

Aufgabe 3:c) Größte darstellbare Zahl mit 5 Bit in 2er-Komplement.

Aufgabe 3:d) Kleinste darstellbare Zahl mit 5 Bit in 2er-Komplement.

Aufgabe 3:e) In UNIX-Systemen wird die Zeit als Sekunden seit dem 1. 1. 1970 gezählt. Bei vorzeichenloser 32-Bit-Speicherung: In welchem Jahr tritt ein Überlaufproblem auf?

[*10 BE*]

Aufgabe 4: 2er-Komplement (8 Bit). Gib die 8-Bit-2er-Komplement-Darstellung an:
a) 9_{10} b) -42_{10} c) 127_{10} d) -128_{10}. [*8 BE*]

Aufgabe 5: BCD. Stelle die Dezimalzahlen als BCD dar (je Dezimalziffer 4 Bit):
a) 9 b) 42 c) 524. [*6 BE*]

Hausaufgaben

Aufgabe 1: Zahlendarstellungen – Tabelle vervollständigen. Trage die jeweils fehlenden Darstellungen ein.

Dezimal	Binär	Hex
12_{10}		
85_{10}		
3529_{10}		

[*6 BE*]

Aufgabe 2: Addition (vorzeichenlos, Binär). Addiere und gib die Dezimalwerte der Summanden und des Ergebnisses an. Tritt ein Overflow auf?

Aufgabe 2:a) $1011_2 + 0001_2$ Overflow? ☐ ja ☐ nein

Aufgabe 2:b) $10011_2 + 10100_2$ Overflow? ☐ ja ☐ nein

[*8 BE*]

Aufgabe 3: Addition (2er-Komplement, 8 Bit). Addiere die folgenden 8-Bit-2er-Komplement-Zahlen. Gib die Dezimalwerte der Summanden und des Ergebnisses an. Tritt ein Overflow auf?

Aufgabe 3:a) $00101010_2 + 10000000_2$ Overflow? ☐ ja ☐ nein

Aufgabe 3:b) $01000011_2 + 01000100_2$ Overflow? ☐ ja ☐ nein

[*10 BE*]

Aufgabe 4: Subtraktion (2er-Komplement, 8 Bit). Wandle zunächst in 8-Bit-2er-Komplement und berechne:

Aufgabe 4:a) $10 - 63$ Ergebnis mit 8 Bit korrekt darstellbar? ☐ ja ☐ nein

Aufgabe 4:b) $-50 - 80$ Ergebnis mit 8 Bit korrekt darstellbar? ☐ ja ☐ nein

[*8 BE*]

Aufgabe 5: Größer oder kleiner? Welche Zahl ist größer? Begründe durch Umrechnung ins Dezimalsystem (vorzeichenlos).

Aufgabe 5:a) 1111_2 oder F_{16}

Aufgabe 5:b) 10101_2 oder AC_{16}

Aufgabe 5:c) 10010101_2 oder $8C_{16}$

[*6 BE*]

Heinrich-von-Kleist-Schule
Eschborn

Fach: Informatik
Kurs: E1
Datum: _____
Name: _____

Arbeitsblatt 3.1

Thema: Zahlensysteme – Umwandeln & schriftlich Rechnen (Binär/Hex)

Bearbeitungshinweise

- Ergebnisse klar kennzeichnen; Rechenschritte (schriftlich) nachvollziehbar darstellen.
- Verwende bei Binärzahlen den Index $_2$, bei Hexzahlen $_{16}$, bei Dezimalzahlen $_{10}$.
- Bei schriftlicher Division/Multiplikation bitte wie im Tafelanschrieb zeigen (Zwischenzeilen).

Präsenzaufgaben

Aufgabe 1: Dual → Dezimal. Berechne die Dezimalwerte. [*8 BE*]
a) 1101111010_2 b) 1010110_2 c) 1111111001_2 d) 1100110011_2.

Aufgabe 2: Hex → Dezimal. Berechne die Dezimalwerte. [*8 BE*]
a) $14F5B_{16}$ b) $AB3D_{16}$ c) $5EA3_{16}$ d) $9C23_{16}$.

Aufgabe 3: Dezimal → Dual und Hex. Wandle jeweils in beide Systeme um (ohne Taschenrechner). [*8 BE*]
a) 3786_{10} b) 14876_{10} c) 2243_{10} d) 1024_{10}.

Aufgabe 4: Dual ↔ Hex. [*8 BE*]
a) $1101111010_2 \to {}_{16}$ b) $1010110_2 \to {}_{16}$ c) $1111111001_2 \to {}_{16}$ d) $1100110011_2 \to {}_{16}$
e) $14F5B_{16} \to {}_2$ f) $AB3D_{16} \to {}_2$ g) $5EA3_{16} \to {}_2$ h) $9C23_{16} \to {}_2$.

Hausaufgaben

Aufgabe 1: Addition (schriftlich, Binär). Ergebnis zusätzlich in Dezimal angeben. [*9 BE*]
a) $1110_2 + 1001_2$ b) $110111_2 + 101110_2$ c) $1010110_2 + 1100111_2$.

Aufgabe 2: Subtraktion (schriftlich, Binär). Ergebnis zusätzlich in Dezimal angeben. [*9 BE*]
a) $110111_2 - 11010_2$ b) $1100110_2 - 111001_2$ c) $10101010_2 - 1111101_2$.

Aufgabe 3: Multiplikation (schriftlich, Binär). Ergebnis zusätzlich in Dezimal angeben. [*9 BE*]
a) $111_2 \cdot 1011_2$ b) $1010_2 \cdot 110011_2$ c) $111_2 \cdot 1101_2$.

Aufgabe 4: Division (schriftlich, Binär). Ergebnis (Quotient) und Rest angeben; zusätzlich Dezimalwerte. [*9 BE*]
a) $10010001_2 : 101_2$ b) $1101100110_2 : 1010_2$ c) $1111111001_2 : 1110001_2$.

Bezug: Kapitel 2. Dieses Blatt vertieft die Inhalte zum Umwandeln von Zahlen verschiedener Basen.

Heinrich-von-Kleist-Schule
Eschborn

Fach: Informatik
Kurs: E1
Datum: _____
Name(n): _____

Arbeitsblatt 4

Thema: Hardwarearchitektur — Von-Neumann-Architektur (Kapitel 4)

Bearbeitungshinweise

- **Arbeitsform:** Gruppenarbeit (2–3 Personen).
- **Abgabe:** 1–2 Seiten Handout (Stichpunkte, Skizzen/Diagramme & Quellen).
- **Präsentation:** 7–10 Minuten pro Gruppe.
- **Quellen:** Internet/Lehrvideos/Bücher; Quellen am Ende angeben.
- **Bezug:** Inhalte zu Kapitel 3 *Hardwarearchitektur*.

Ziel

Ihr versteht Aufbau, Komponenten und Arbeitsweise der **Von-Neumann-Architektur** und könnt Vorteile, Nachteile und Abgrenzung zur Harvard-Architektur erläutern.

Gruppenauftrag

Aufgabe 1: Hintergrund. [*6 BE*]
Wer war *John von Neumann*? In welchem historischen Kontext (1940er) entstand die Architektur? Nenne wichtige Projekte/Computer der Zeit.

Aufgabe 2: Grundidee der Von-Neumann-Architektur. [*8 BE*]
Erklärt den Begriff *„Speicherprogrammiertechnik“*. Warum ist ein gemeinsamer Speicher für Programm *und* Daten so bedeutsam? Skizziert das Grundschema (Blockdiagramm).

Aufgabe 3: Hauptkomponenten (präzise beschreiben). [*12 BE*]

Aufgabe 3:a) **ALU (Rechenwerk):** Aufgaben, typische Operationen, Rolle des Übertrags/Flags.

Aufgabe 3:b) **Steuerwerk (Control Unit):** Befehlsholung, Dekodierung, Steuersignale.

Aufgabe 3:c) **Speicher:** Welche Arten von Informationen liegen dort? (Programm, Daten, Stack …)

Aufgabe 3:d) **Ein-/Ausgabe (I/O):** Beispiele (Tastatur, Display, Netz), wie angebunden?

Aufgabe 3:e) **Bus-System:** Adress-, Daten- und Steuerbus – Zweck und Zusammenspiel.

Aufgabe 4: Arbeitsweise: Fetch–Decode–Execute. [*10 BE*]
Beschreibt den Von-Neumann-Zyklus (Befehl holen → decodieren → ausführen). Veranschaulicht das an *einem* einfachen Maschinenbefehl (z. B. `LOAD`, `ADD`, `STORE`) mit einem Mini-Beispiel.

Aufgabe 5: Vor- und Nachteile. [*8 BE*]
Warum war das Modell revolutionär? Welche Grenzen gibt es (z. B. *Von-Neumann-Flaschenhals*) und wodurch entstehen sie?

Aufgabe 6: Vergleich (optional): Harvard vs. Von Neumann. [*6 BE*]
Was unterscheidet die Harvard-Architektur? Wo wird sie eingesetzt? Nenne ein konkretes Beispiel (z. B. Mikrocontroller/DSP) und begründe, warum Harvard dort sinnvoll ist.

Hausaufgaben / Vertiefung

Aufgabe 1: Skizze mit Legende. [*6 BE*]
Zeichne ein eigenes Blockdiagramm einer Von-Neumann-CPU (ALU, Steuerwerk, Speicher, I/O, Busse). Beschrifte alle Pfeile kurz (welche Signale/Informationen fließen?).

Aufgabe 2: Beispielablauf. [*6 BE*]
Simuliere auf einer halben Seite den Ablauf von zwei Befehlen (z. B. `LOAD A`, `ADD B`, `STORE A`) im *Fetch–Decode–Execute*-Zyklus. Was passiert in welchem Takt? Welche Register/Busse sind beteiligt?

Aufgabe 3: Kurzvergleich. [*4 BE*]
Erkläre in 5–6 Sätzen, wie „getrennte Instruktions-/Daten-Caches“ (I-Cache/D-Cache) das Von-Neumann-Prinzip *ergänzen* und wo trotzdem der Flaschenhals bleibt.

Arbeitsauftrag „Von-Neumann-Architektur“.