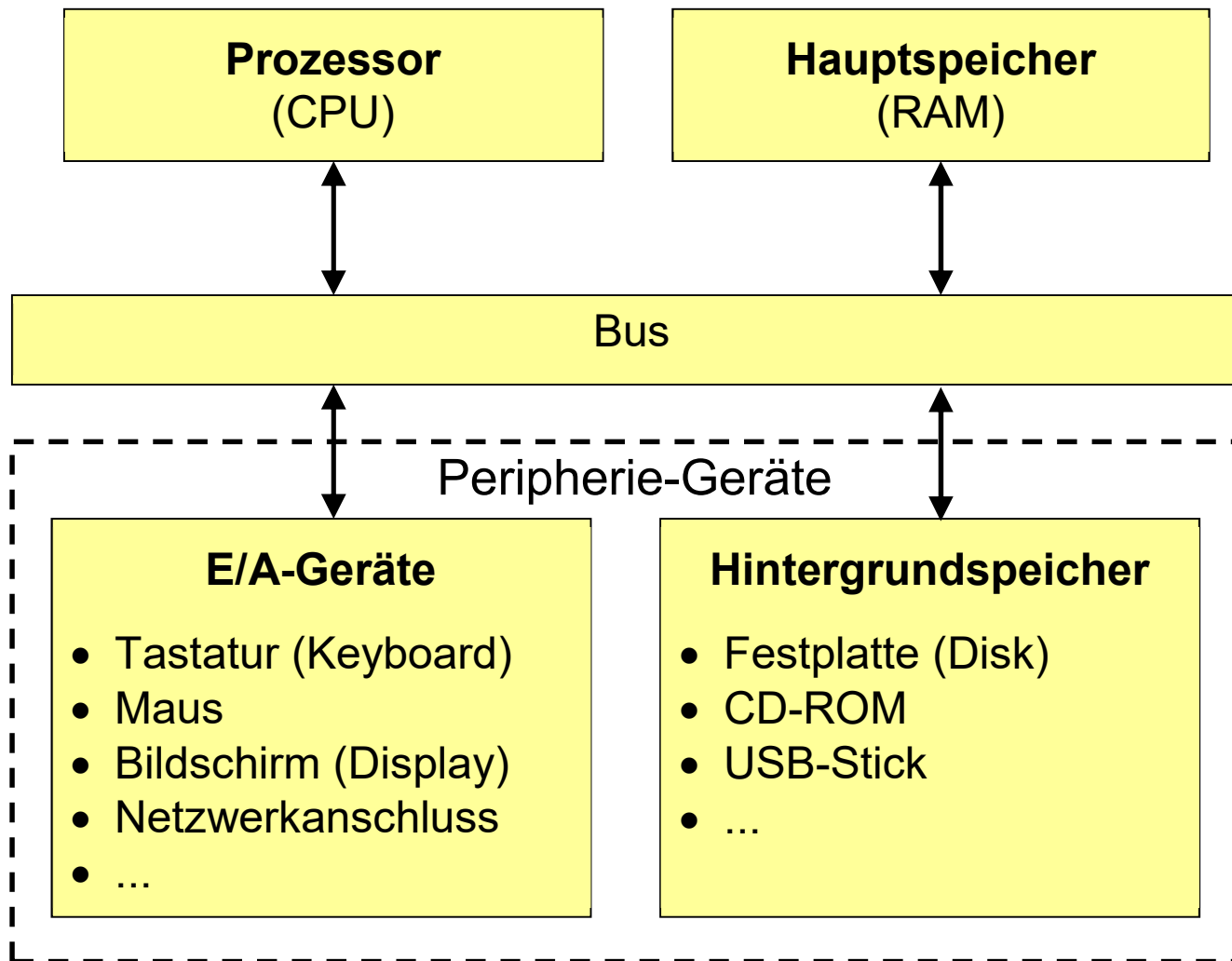


# Programmietechnik 1

## Teil 1: Rechner und Zahlen Hauptspeicher / Programmierumgebung / Stellenwertsysteme / Zeichencodes

# Rechner: Prinzipieller Aufbau



**CPU** = Central  
Processing  
Unit

**RAM** = Random  
Access  
Memory

**E/A** = Ein-/  
Ausgabe  
(*I/O = Input/Output*)

**CD** = Compact  
Disk

**ROM** = Read  
Only  
Memory

**USB** = Universal  
Serial  
Bus

# Rechner: Prinzipielle Arbeitsweise

---

Ein Rechner kann intern nur mit Zahlen umgehen.

- Der **Prozessor** kann Zahlen arithmetisch und logisch verknüpfen, d.h. vergleichen, addieren, usw.

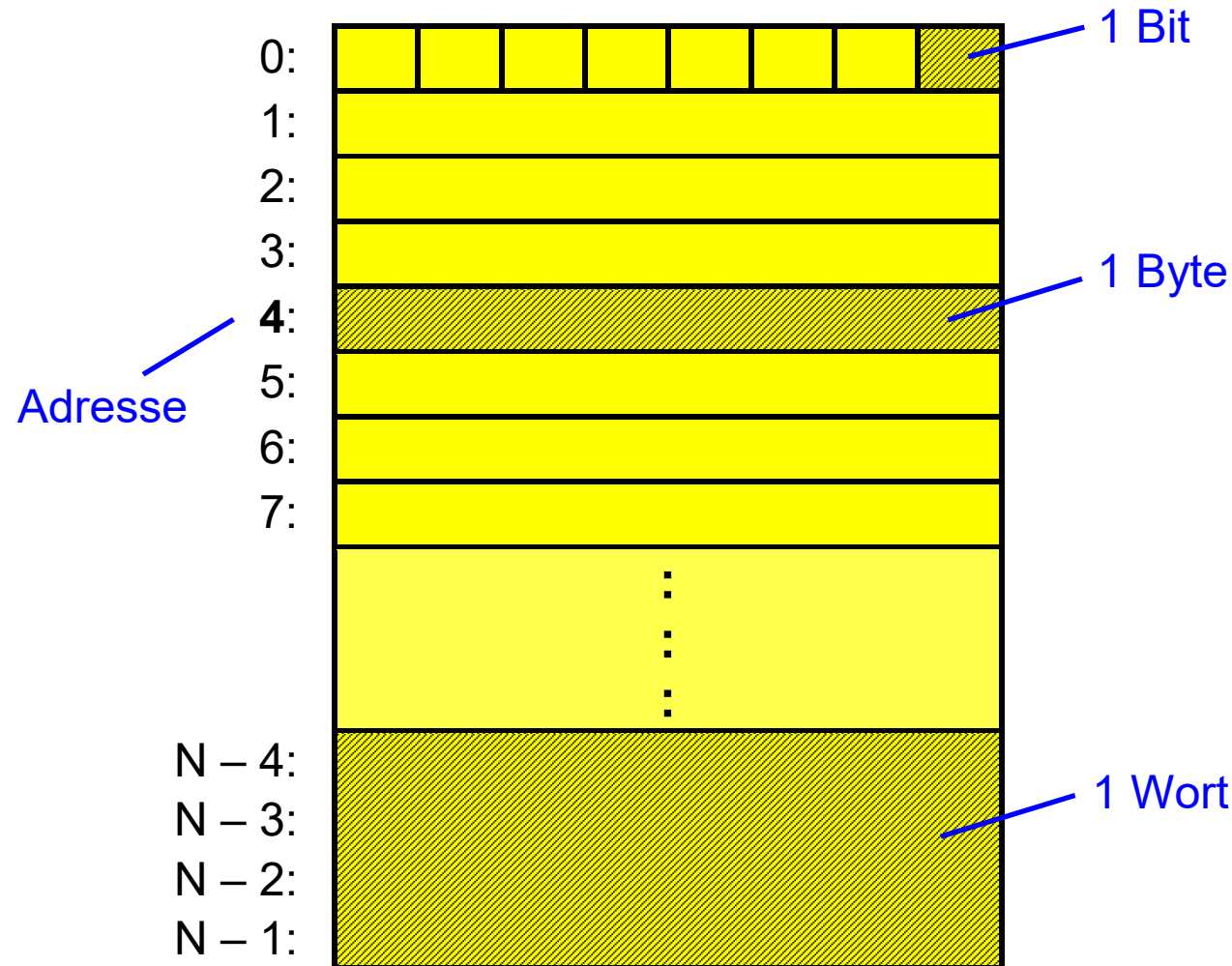
*Die Zahlen müssen dafür im Hauptspeicher liegen.*

Der Prozessor kann Programme ausführen, d.h. Zahlenfolgen als Befehle interpretieren.

*Ein Programm muss dafür im Hauptspeicher liegen.*

- Der **Bus** überträgt Zahlen zwischen Prozessor, Hauptspeicher und Peripheriegeräten.
- Zahlen im **Hauptspeicher** gehen verloren, wenn das zugehörige Programm zu Ende läuft oder der Rechner abgeschaltet wird (*flüchtiger Speicher*).
- Zahlen im **Hintergrundspeicher** bleiben erhalten, wenn ein Programm zu Ende läuft oder der Rechner abgeschaltet wird (*dauerhafter Speicher*).

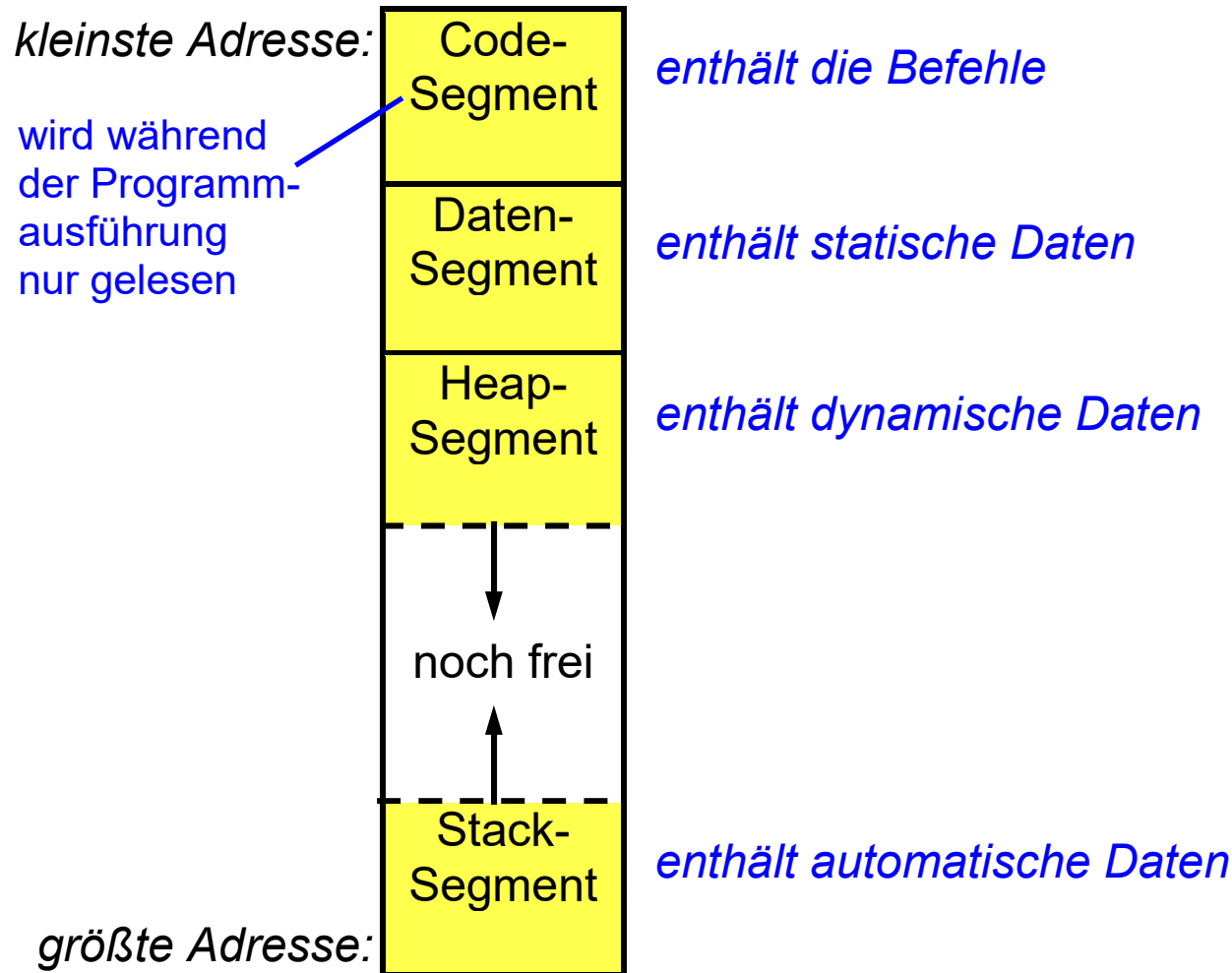
# Rechner: Aufbau des Hauptspeichers (1)



Der **Hauptspeicher** ist aus Programmiersicht eine Folge von durchnummerierten Speicherzellen, die jeweils 1 Byte groß sind.

Die Nummern der Speicherzellen werden als Adressen bezeichnet.

# Rechner: Aufbau des Hauptspeichers (2)



Der **Hauptspeicher** ist logisch aufgeteilt:

- in einen Bereich für die **Befehle** eines Programms
- in der Regel 3 Bereiche für die **Daten** eines Programms



- 1 **Bit**: kleinste Speichereinheit  
*Kann genau eine Binärziffer speichern.*
- 1 **Byte**: Bitkombination, kleinste in Programmen adressierbare Einheit  
*Auf allen "normalen" Rechnern 8 Bit.*
- 1 **Wort**: Bytekombination, Transporteinheit auf dem Bus  
*Größe rechnerabhängig, typisch 8 Byte.*
- 
- 1 **KiB**: Kibibyte =  $2^{10}$  Byte = 1024 Byte  $\approx 10^3$  Byte = 1 **kB**
- 1 **MiB**: Mebibyte =  $2^{20}$  Byte = 1024 · 1024 Byte  $\approx 10^6$  Byte = 1 **MB**
- 1 **GiB**: Gibibyte =  $2^{30}$  Byte = 1024 · 1024 · 1024 Byte  $\approx 10^9$  Byte = 1 **GB**

# Rechner: Programmierung

---

Programme werden in einer **Programmiersprache** geschrieben.

- Programmiersprachen können automatisch in Maschinencode übersetzt werden, d.h. in Zahlenfolgen, die der Prozessor ausführen kann.

Die primitivsten Programmiersprachen sind die **Assembler-Sprachen**:

- lediglich lesbare Namen für Maschinenbefehle
- automatische Rückübersetzung von Maschinencode in Assemblercode möglich
- rechnerabhängig
- Benutzung mühsam und fehleranfällig

Heute fast ausschließlich **höhere Programmiersprachen**:

- keine automatische Rückübersetzung des Maschinencodes mehr möglich
- rechnerunabhängig
- Beispiele: Fortran, Cobol, C, C++, Java, C#, ...
- in dieser Lehrveranstaltung: **Java**

# Rechner: Java Programmier-Umgebung (1)

---

- mit einem **Editor** wird der **Quellcode** (Source Code) von Java-Programmen erstellt, im Prinzip ist jeder Texteditor geeignet, unter Linux z.B. **vi** oder **pluma**:

**vi** *Beispiel.java*

*Guter Editor für Maus-Feinde*

**pluma** *Beispiel.java*

*Maus-gesteuerter Editor mit Syntaxhervorhebung*

- mit dem **Übersetzer** (Compiler) wird Quellcode auf die korrekte Benutzung der Programmiersprache geprüft und ausführbarer **Bytecode** erzeugt:

**javac** *Beispiel.java*

*Java-Compiler*

*(prüft syntaktische Korrektheit und erzeugt Bytecode **Beispiel.class**)*

- mit der **virtuellen Maschine** wird Bytecode ausgeführt:

**java** *Beispiel*

*Java Virtuelle Maschine*

*(lädt **Beispiel.class** in den Hauptspeicher und führt das Programm aus)*



## Rechner: Java Programmier-Umgebung (2)

---

- aus Quellcode kann **Dokumentation** generiert werden:

`javadoc -d doc Beispiel.java`      *erzeugt im Unterverzeichnis doc  
Dokumentation im HTML-Format*

- mit **statischen Analysewerkzeugen** können Qualitätseigenschaften geprüft werden  
(Aufruf am besten über *ant*, siehe unten):

`checkstyle`      *untersucht Quellcode auf Einhaltung von Java Stilregeln*

`spotbugs`      *untersucht Bytecode auf Fehlerquellen*

- mit einem **Build-Werkzeug** kann die Programmerstellung automatisiert werden,  
z.B mit *ant*:

`ant compile`      *javac aufrufen*

`ant style`      *checkstyle aufrufen*

`ant bugs`      *spotbugs aufrufen*

*Datei build.xml erforderlich, die die Arbeitsschritte für `compile`, `style` und `bugs` enthält*



- das Linux-Dateisystem kann ähnlich wie das von Windows über den File-Manager der grafischen Benutzeroberfläche benutzt werden.
- Geübte bevorzugen oft Konsolen-Kommandos:

Mit Verzeichnissen (Directories) umgehen

<b>mkdir</b> <i>directory</i>	<i>neues Verzeichnis anlegen</i>
<b>rmdir</b> <i>directory</i>	<i>leeres Verzeichnis löschen</i>
<b>cd</b> <i>directory</i>	<i>Arbeitsverzeichnis wechseln</i>
<b>cd</b> <i>..</i>	<i>zum übergeordneten Verzeichnis</i>
<b>cd</b>	<i>zum Heimverzeichnis</i>
<b>ls</b> <i>directory</i>	<i>Verzeichnisinhalt auflisten</i>
<b>ls</b>	<i>aktuelles Arbeitsverzeichnisses auflisten</i>
<b>pwd</b>	<i>aktuelles Arbeitsverzeichnis anzeigen</i>

# Rechner: Linux Umgebung (2)

---

- ... Fortsetzung Konsolen-Kommandos:

Mit Dateien (*Files*) umgehen

**rm** *file*                      *Datei löschen*

**mv** *source target*            *Datei umbenennen oder in ein anderes Verzeichnis verschieben*

**cp** *source target*            *Datei kopieren*

Hilfe zu einem Kommando anfordern

**man** *command*                *zeigt die Beschreibung des Kommandos im Handbuch (Manual) an*

und vieles mehr ...

# Zahlen: 1, 2, 3, 4, viele ...

---

II

IIII

III

IIIIII

IIIIII

Motto der Programmiertechnik:

"Alles ist eine Zahl"

- auf einen Blick kann ein Mensch höchstens die Anzahl 4 sicher erkennen
- bei größerer Anzahl hilft nur abzählen  
dazu braucht man ein Zahlensystem

# Zahlen: Dezimalsystem (10er-System)

---

Das Dezimalsystem ist ein **Stellenwertsystem**,  
d.h. der Wert einer Ziffer hängt davon ab, an welcher Stelle sie in einer Zahl steht.

$$\begin{aligned} 5582 &= 5 \cdot 1000 + 5 \cdot 100 + 8 \cdot 10 + 2 \cdot 1 \\ &= 5 \cdot 10^3 + 5 \cdot 10^2 + 8 \cdot 10^1 + 2 \cdot 10^0 \end{aligned}$$

**Ziffern**            0, 1, 2, 3, 4, 5, 6, 7, 8, 9

**Basis**            10 (grösste Ziffer plus 1)

**Stellenwerte**    1, 10, 100, 1000, ... ( $10^{i-1}$  für die i-te Stelle von hinten)

Vorteile von Stellenwertsystemen:

- Mit kleinem Zeichenvorrat (Ziffern) beliebig grosse Zahlen darstellbar
- Einfaches Rechnen

# Zahlen: Stellenwertsysteme

---

Stellenwertsysteme sind mit beliebiger Basis möglich:

$$z_3 z_2 z_1 z_0 = z_3 \cdot b^3 + z_2 \cdot b^2 + z_1 \cdot b^1 + z_0 \cdot b^0$$

**Basis**  $b \geq 2$

**Ziffern**  $z_i \in \{ 0, 1, \dots, b - 1 \}$

**Stellenwerte**  $1, b^1, b^2, b^3, \dots$

$$\text{Wert}(z_{n-1} \dots z_2 z_1 z_0) = \sum_{i=0}^{n-1} \text{Wert}(z_i) \cdot b^i$$

Beim Programmieren übliche Stellenwertsysteme:

System	Basis	Ziffern	Stellenwerte
Binär (Dual)	2	0,1	1,2,4, <u>8</u> , <u>16</u> ,...
Oktal	8	0,1,2,3,4,5,6,7	1, <u>8</u> ,64,512,...
Dezimal	10	0,1,2,3,4,5,6,7,8,9	1,10,100,1000,...
Hexadezimal (Sedezimal)	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F	1, <u>16</u> ,256,4096,...

# Zahlen: Binärsystem (2er-System)

---

- Rechner stellen Zahlen im Binärsystem dar, weil sich die beiden Ziffern gut kodieren lassen:  
Ziffer **1**: Strom fließt / Kondensator geladen  
Ziffer **0**: kein Strom / Kondensator nicht geladen

- Binärzahlen sind für Menschen eher unübersichtlich:

$$5582_{10} = 1010111001110_2$$

- Oktal- oder Hexadezimalsystem als Kurzschreibweise:

Jede Oktalziffer fasst 3 Binärziffern zusammen.

$$5582_{10} = \underline{001} \underline{010} \underline{111} \underline{001} \underline{110}_2 = 1 \ 2 \ 7 \ 1 \ 6_8$$

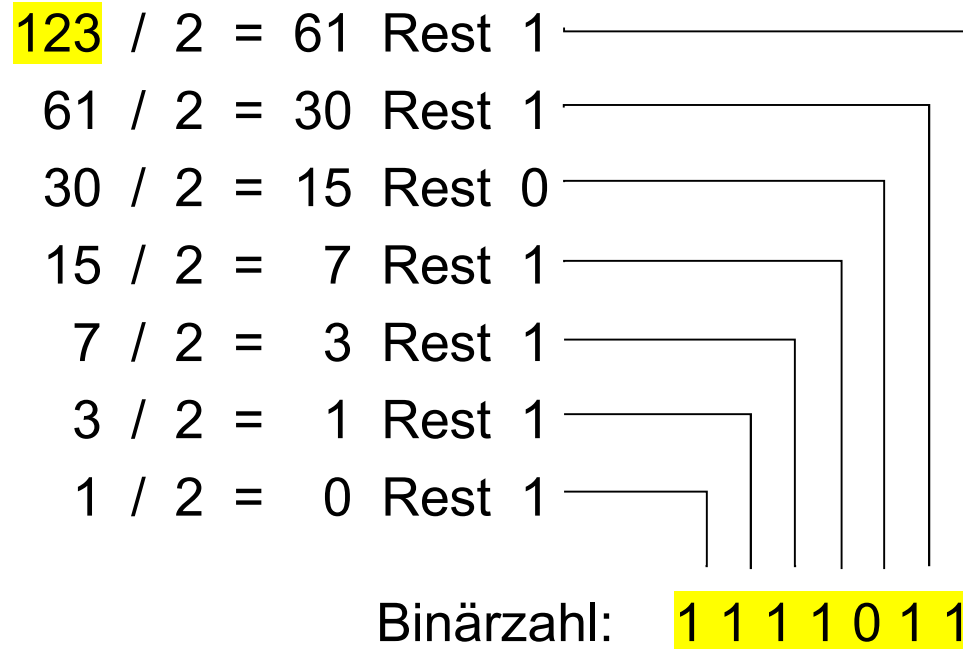
Jede Hexadezimalziffer fasst 4 Binärziffern zusammen.

$$5582_{10} = \underline{0001} \underline{0101} \underline{1100} \underline{1110}_2 = 1 \ 5 \ C \ E_{16}$$

# Zahlen: Umrechnung vom Dezimal- zum Binärsystem



- wiederholte Division durch 2 liefert die Binärstellen von rechts nach links:





# Zahlen: Vorzeichen-Darstellung



- Feste Anzahl Stellen für alle Zahlen verwenden.
- Die **vorderste Stelle als Vorzeichen** interpretieren:
  - 0** bedeutet positives Vorzeichen
  - 1** bedeutet negatives Vorzeichen
- Die restlichen Stellen geben den Betrag an, bei negativen Zahlen in **2er-Komplement-Darstellung**.

*Das 2er-Komplement einer Zahl erhält man durch bit-weise Invertierung und anschließende Addition von 1.*

Beispiele mit 4-stelliger Zahlendarstellung:

$$+0_{10} = 0000_2$$

$$+1_{10} = 0001_2$$

$$+7_{10} = 0111_2$$

$$+8_{10} \text{ nicht darstellbar}$$

$$-0_{10} \text{ gibt es nicht}$$

$$-1_{10} = 1111_2$$

$$-7_{10} = 1001_2$$

$$-8_{10} = 1000_2$$

# Zahlen: Vorzeichenwechsel durch 2er-Komplement

---

- erster Schritt: bit-weise invertieren

$$\begin{array}{rcl} +5582_{10} & = & 0001010111001110_2 \\ & & 1110101000110001_2 \end{array} \quad \text{1er-Komplement}$$

- zweiter Schritt: 1 addieren

$$\begin{array}{rcl} & & 1110101000110001_2 \\ & + & 0000000000000001_2 \\ \hline -5582_{10} & = & 1110101000110010_2 \end{array} \quad \text{2er-Komplement}$$

- erneute Anwendung liefert ursprüngliches Vorzeichen:

$$\begin{array}{rcl} -5582_{10} & = & 1110101000110010_2 \\ & & 0001010111001101_2 \quad \text{1er-Komplement} \\ & + & 0000000000000001_2 \\ \hline +5582_{10} & = & 0001010111001110_2 \quad \text{2er-Komplement} \end{array}$$

# Zahlen: Festkomma-Darstellung

---

Stellenwertsysteme mit Nachkommateil:

$$z_1 z_0, z_{-1} z_{-2} = z_1 \cdot b^1 + z_0 \cdot b^0 + z_{-1} \cdot b^{-1} + z_{-2} \cdot b^{-2}$$

**Basis**  $b \geq 2$

**Ziffern**  $z_i \in \{0, 1, \dots, b-1\}$

**Stellenwerte**  $\dots, b^{-2}, b^{-1}, 1, b^1, b^2, \dots$

**Komma** **fest** hinter der Ziffer mit Stellenwert 1

$$\text{Wert}(z_{n-1} \dots z_0, z_{-1} \dots z_{-m}) = \sum_{i=-m}^{n-1} \text{Wert}(z_i) \cdot b^i$$

Beispiele:

$$11,11_2 = 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 2 + 1 + 0,5 + 0,25 = 3,75_{10}$$

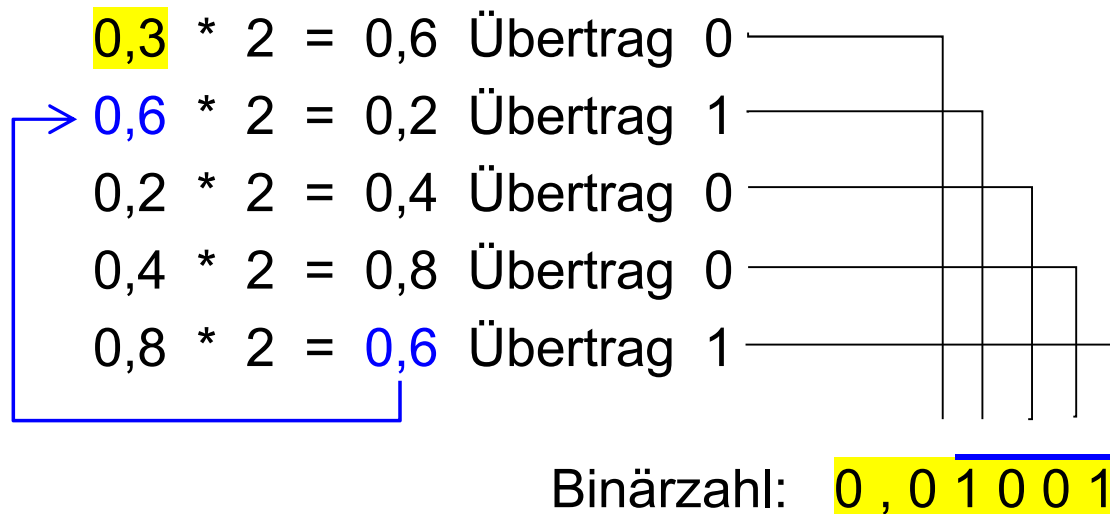
$$2,3_{10} = 10,01001_2$$

*Rechner verwenden keine Festkomma-Darstellung,  
weil sie für sehr große und sehr kleine Zahlen zu viel Speicherplatz braucht.*

# Zahlen: Umrechnung von Nachkommastellen



- wiederholte Multiplikation mit 2 liefert die Binärstellen von links nach rechts:



*Binärzahl wird  
in diesem Beispiel  
ab der zweiten  
Nachkommastelle  
periodisch*

# Zahlen: Gleitkomma-Darstellung



**Mantisse** · 2<sup>**Exponent**</sup>    Rechner stellen gebrochene und sehr große Zahlen als Binärzahlen-Paar mit Gleitkomma dar.

- Die **Mantisse** ist eine Binärzahl mit Vorzeichen und Komma  
*Das Komma steht meist hinter der ersten 1 von links.*
- Der **Exponent** ist eine ganze Binärzahl mit Vorzeichen.

Die Details sind rechnerabhängig, aber weit verbreitet ist der **Standard IEEE 754**:

Vorzeichen <b>v</b> (1 Stelle)	Exponent <b>e</b> (8 Stellen)	Mantisse <b>m</b> (23 Stellen)
--------------------------------	-------------------------------	--------------------------------

$$\text{Falls } e = 0: \quad \text{Wert}(\text{vem}) = (-1)^v \cdot 0.m \cdot 2^{-126}$$

$$\text{Falls } 1 \leq e \leq 254: \quad \text{Wert}(\text{vem}) = (-1)^v \cdot 1.m \cdot 2^{e-127}$$

*Der Exponent 255 ist für die speziellen Werte Unendlich und NaN (Not a Number) reserviert.*

Beispiel:

$$11,11_2 = (-1)^0 \cdot 1.111 \cdot 2^{128-127}$$

0	1 0000000	111 0000000000000000000000
---	-----------	----------------------------

# Zahlen: Wertebereiche

---

32-stellige ganze Binärzahl mit Vorzeichen:

- grösster Wert:  $2\,147\,483\,647 = 2^{31} - 1 \approx 2 \cdot 10^9$
- kleinster Wert:  $-2\,147\,483\,648 = -2^{31} \approx -2 \cdot 10^9$

32-stellige Gleitkomma-Zahl nach IEEE 754:

- grösster Wert:  $(2 - 2^{-23}) \cdot 2^{127} \approx 3,4 \cdot 10^{38}$
- kleinster Wert:  $-(2 - 2^{-23}) \cdot 2^{127} \approx -3,4 \cdot 10^{38}$
- kleinster positiver Wert:  $2^{-149} \approx 1,4 \cdot 10^{-45}$

**Achtung:** Gleitkommazahlen sind ungenau !

*Der größte Wert z.B. hat zwar stolze 128 Binärstellen (entspricht ungefähr 39 Dezimalstellen), von denen werden aber nur die ersten 24 dargestellt (entspricht ungefähr 7 Dezimalstellen).*

# Zahlen: Codierung nicht-numerischer Daten

---

## Codierung:

Alle vorkommenden nicht-numerischen Werte auflisten und durchnummerieren.

*Der Rechner verwendet die Nummern statt der Werte.*

Beispiele:

- durchnummerierte Liste von Buchstaben, Ziffern, Sonderzeichen (z.B. Satzzeichen) und Steuerzeichen (z.B. Zeilenvorschub).
- durchnummerierte Liste von Befehlen, die ein Prozessor ausführen kann.
- durchnummerierte Liste der am Bildschirm darstellbaren Farben
- usw.

# Zahlen: Zeichencodierung

---

- **ASCII** (American Standard Code for Information Interchange)

7-Bit Code mit den "wichtigsten" internationalen Zeichen

*Normen: ISO/IEC 646, DIN 6603*

**Latin-1** ist Erweiterung auf 8-Bit mit z.B. westeuropäischen Umlauten

*Normen: ISO/IEC 8859, DIN 6603*

- **EBCDIC** (Extended Binary Coded Decimal Interchange Code)

8-Bit Code, z.B. auf IBM Großrechnern üblich

- **Unicode**

Industriestandard mit dem Anspruch sämtliche Sinn tragenden Schriftzeichen aller bekannten Schriftkulturen zu erfassen (*auch Keilschrift !*)

anfangs 16-Bit-, inzwischen 21-Bit-Code

enthält ASCII mit Latin-1 als Untermenge

*Normen: ISO/IEC 10646 (Universal Character Set, mit Unicode kompatibler 32-Bit-Code)*



# Zahlen: ASCII – Zeichentabelle

	0	1	2	3	4	5	6	7
00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
01	BS	HT	NL	VT	NP	CR	SO	SI
02	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
03	CAN	EM	SUB	ESC	FS	GS	RS	US
04	SP	!	"	#	\$	%	&	'
05	(	)	*	+	,	-	.	/
06	0	1	2	3	4	5	6	7
07	8	9	:	;	<	=	>	?
10	@	A	B	C	D	E	F	G
11	H	I	J	K	L	M	N	O
12	P	Q	R	S	T	U	V	W
13	X	Y	Z	[	\	]	^	_
14	`	a	b	c	d	e	f	g
15	h	i	j	k	l	m	n	o
16	p	q	r	s	t	u	v	w
17	x	y	z	{		}	~	DEL

Beispiele:

Code	Zeichen	
	'3'	'A'
<i>oktal</i>	63	101
<i>hex</i>	33	41
<i>dezimal</i>	51	65
<i>binär</i>	110011	1000001

# Zahlen: Unicode

---

- **Code Point** = mögliche Zeichennummer im Zahlenbereich 000000 – 10ffff  
*Der Zahlenbereich ist in derzeit 17 Planes (Ebenen) mit je 65536 Code Points unterteilt (nicht alle dieser Code Points sind mit Zeichen belegt).*  
*Die Plane 0 (Zahlenbereich 0 – ffff) heißt Basic Multilingual Plane (BMP) (viele Anwendungen brauchen nur die Zeichen der BMP).*  
*Die ersten 128 Zeichen (Zahlenbereich 0 – 7f) entsprechen dem ASCII-Code.*
- **Codierungsform** = Darstellung von Code Points als Folge von **Code Units**  
*Beim Unicode Transformation Format (UTF) ist der Speicherplatzbedarf zeichenabhängig:*
  - UTF-8**      1 bis 4 Code Units zu je 8 Bit stellen einen Code Point dar  
*(verbreitet bei Internetdiensten wie z.B. E-Mail)*
  - UTF-16**    1 bis 2 Code Units zu je 16 Bit stellen einen Code Point dar  
*(verwendet z.B. in Windows und Java)*
- **Codierungsschema** = Darstellung von Code Units als Bytesequenzen  
*Legt die Reihenfolge der Bytes einer Code Unit fest.*

# Zahlen: UTF-8

---

Die erste Code-Unit eines Zeichens zeigt am Anfang die Länge der Bytesequenz an, eventuelle Folgebytes beginnen immer mit 10:

- 1-Byte-Zeichen: **0**xxxxxxx *7 Bit*
- 2-Byte-Sequenz: **110**xxxxx **10**xxxxxx *11 Bit*
- 3-Byte-Sequenz: **1110**xxxx **10**xxxxxx **10**xxxxxx *16 Bit*
- 4-Byte-Sequenz: **11110**xxx **10**xxxxxx **10**xxxxxx **10**xxxxxx *21 Bit*

Beispiele:

<u>Zeichen</u>	<u>UTF-16</u>	<u>UTF-16 binär</u>	<u>UTF-8 binär</u>
A	U+0041	00000000 01000001	<b>0</b> 1000001
Ä	U+00C4	00000000 11000100	<b>110</b> 00011 <b>10</b> 000100
€	U+20AC	00100000 10101100	<b>1110</b> 0010 <b>10</b> 000010 <b>10</b> 101100

# Zahlen: UTF-16

Zeichen außerhalb der Basic Multilingual Plane (U+10000 bis U+10FFFF) werden in ein **High-Surrogate** und ein **Low-Surrogate** mit je 16 Bit aufgeteilt:


- 4-Byte-Sequenz: **110110**xx xxxxxxxx **110111**xx xxxxxxxx  
*High-Surrogate* *Low-Surrogate*

*In der BMP sind die Bereiche der Surrogates nicht mit Zeichen belegt!*

Berechnungsvorschrift:

- vom gegebenen Code-Point  $10000_{16}$  subtrahieren  
*es entsteht eine 20-Bit-Zahl zwischen  $00000_{16}$  und  $FFFFF_{16}$*
- die höherwertigen 10 Bit zu  $D800_{16}$  addieren  
*es entsteht das High-Surrogate mit Werten zwischen  $D800_{16}$  und  $DBFF_{16}$*
- die niederwertigen 10 Bit zu  $DC00_{16}$  addieren  
*es entsteht das Low-Surrogate mit Werten zwischen  $DC00_{16}$  und  $DEFF_{16}$*

Beispiel:

<u>Zeichen</u>	<u>Unicode</u>	<u>UTF-16 hex</u>	<u>UTF-16 binär</u>
	U+1D11E	D834 DD1E	<b>110110</b> 00 00110100 <b>110111</b> 01 00011110

---

ant 1-8	EBCDIC 1-23	Latin-1 1-23
ASCII 1-23,1-24	Editor 1-7	Low-Surrogate 1-27
Assemblersprache 1-6	Einerkomplement 1-17	MB 1-5
Binärsystem 1-14,1-15	Festkommazahl 1-18	MiB 1-5
Bit 1-5	GB 1-5	Objectcode 1-7
BMP 1-25	GiB 1-5	Oktalsystem 1-13,1-14
Bus 1-1,1-2	Gleitkommazahl 1-20,1-21	Peripheriegerät 1-1
Byte 1-5	Hauptspeicher 1-1,1-2,1-3,1-4	Prozessor 1-1,1-2
Bytecode 1-7	Heapsegment 1-4	Quellcode 1-7
checkstyle 1-8	Hexadezimalsystem 1-13,1-14	spotbugs 1-8
Code Point 1-25	High-Surrogate 1-27	Stacksegment 1-4
Code Unit 1-25	höhere Programmiersprache 1-6	Stellenwertsystem 1-12,1-13
Codesegment 1-4	IEEE-754 1-20,1-21	Übersetzer 1-7
Codierung 1-22	ISO 8859 1-23	Unicode 1-23,1-25
Codierungsform 1-25	java 1-7	UTF-16 1-25,1-27
Codierungsschema 1-25	javac 1-7	UTF-8 1-25,1-26
Compiler 1-7	javadoc 1-8	Wort 1-5
CPU 1-1	kB 1-5	Zeichencodierung 1-25,1-26
Datensegment 1-4	KiB 1-5	Zweierkomplement 1-16,1-17
Dezimalsystem 1-12		