Datenverarbeitung

Silko Kruse

(kruse@hs-ulm.de)

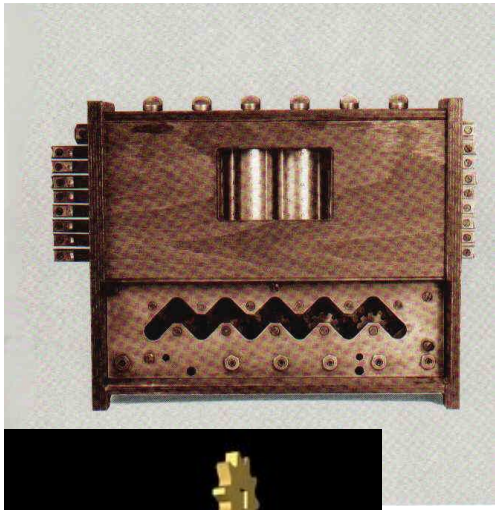
Inhalt

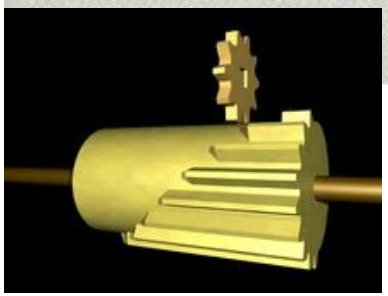
- **Historie, Informationen und Daten, Codes**
- **Zahlendarstellungen, Zahlenkonvertierung, Rechnen mit Dualzahlen, Negative Zahlen, Codesicherung**
- **Aussagenlogik, Schalternetze**
- **Von-Neumann-Rechner, Register und Assemblerbefehle**
- **Assemblerprogrammierung, Sprungbefehle, Unterprogramme, Parametrisierte Prozeduren, Zeiger, Rekursion**
- **Programmierung, Hochsprachenprogrammierung, Datentypen, Primitive Datentypen, Variablen**

Rechenmaschinen

- **Rechenstäbe des Schotten John Napier (1550- 1617)**
- **1622 Rechenschieber** (William Oughtred)
  - Logarithmische Zahlendarstellung

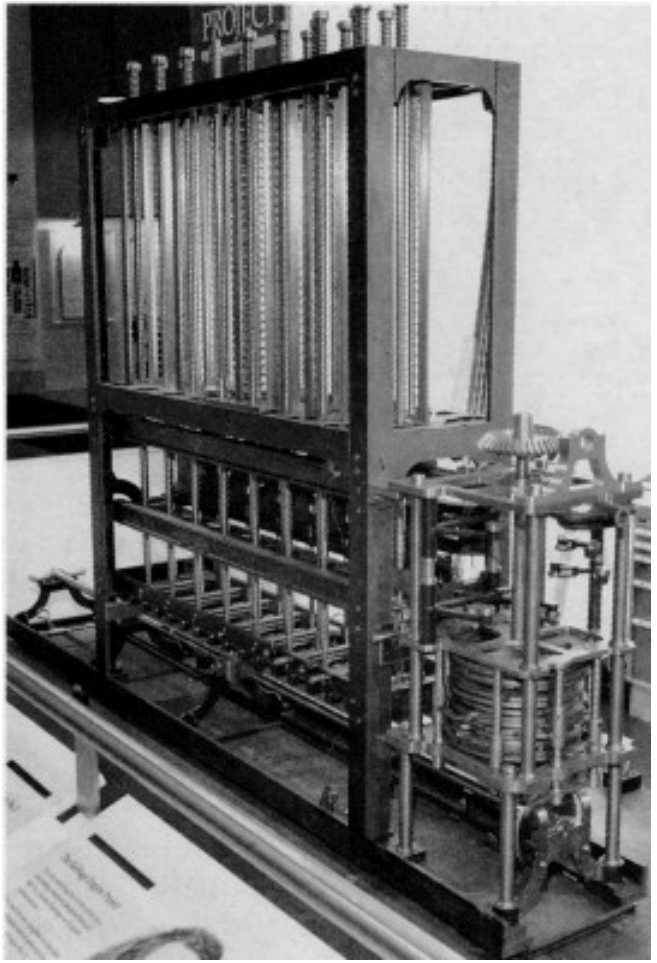
- **1623 Rechenmaschine** (Wilhelm Schickard)
  - Addition und Subtraktion bis zu sechsstelliger Zahlen
- **1643 Pascaline** (Blaise Pascal)
  - Addition und Subtraktion max. siebenstelliger Zahlen
  - Gebaut für seinen Vater, der Steuerbeamter war
- **1673 Rechenmaschine** (Gottfried Wilhelm Leibniz)
  - Staffelwalzenprinzip
  - Multiplikation durch fortgesetztes Addieren





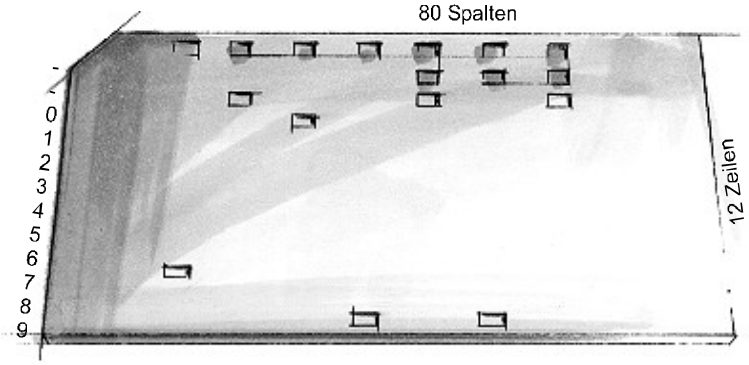
Vorläufer (1)

- **1805 Lochkarte**
  (Joseph-Marie Jacquard)
  - Erstmalige Möglichkeit der Speicherung
- **1833 Analytische Maschine**
  (Charles Babbage)
  - Programmgesteuerte Rechenmaschine
  - Rechenwerk hoher Stellenzahl
  - Paralleler Zehnerübertrag
  - Lochbandsteuerung
  - Speicherwerk für 1000 Zahlen zu 50 Stellen
    - Baugruppen, die auch in modernen Rechnern vorhanden sind
  - Druckwerk für die Ausgabe
  - Bau scheiterte an technischen Unzulänglichkeiten der Zeit



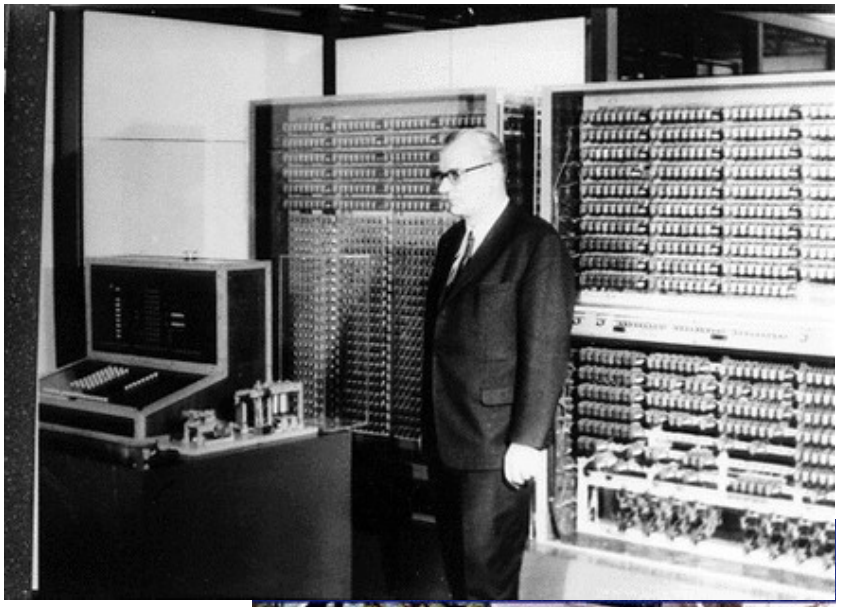
Vorläufer (2)

- **1886 Lochkartenmaschine** (Hermann Hollerith)
  - Elektromagnetische Sortier- und Zählmaschine zur Auswertung von Lochkarten im 20 Dollar Format
  - Anlage bestand aus Kartenlocher, manuellem elektromagnetischem Zähler und Sortiereinrichtung
  - 1890 zur Erleichterung der Volkszählung in den USA eingesetzt
  - Bearbeitungszeit konnte von 7,5 auf 2,5 Jahre reduziert werden
  - Kartenform und Code bis in die 1980er Jahre eingesetzt



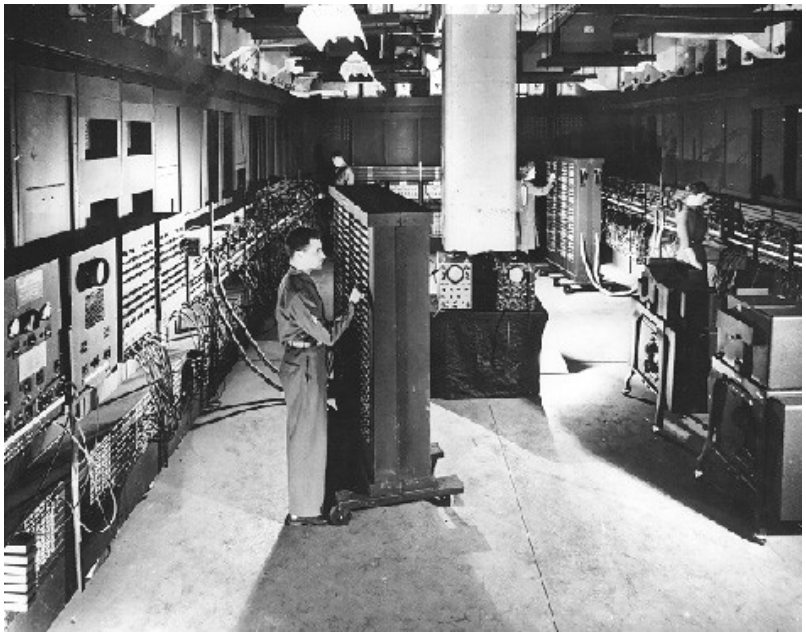
Die Entwicklung des Computers (1)

- **1934 - 1943 Zuse Z3**
  (Konrad Zuse)
  - Erster funktionsfähiger Computer
  - Durch Fortschritte in der Elektromechanik ermöglicht
  - Rechenzeiten
  - Addition: 0,3s, Multiplikation: 5s
  - Interne Zahlendarstellung durch Dualzahlen
    - Bei Eingabe/Ausgabe Dezimal-Dual- bzw. Dual-Dezimal-Konvertierung erforderlich
  - Rechenprogramm von gelochtem 8-spurigem Kinofilm abgetastet



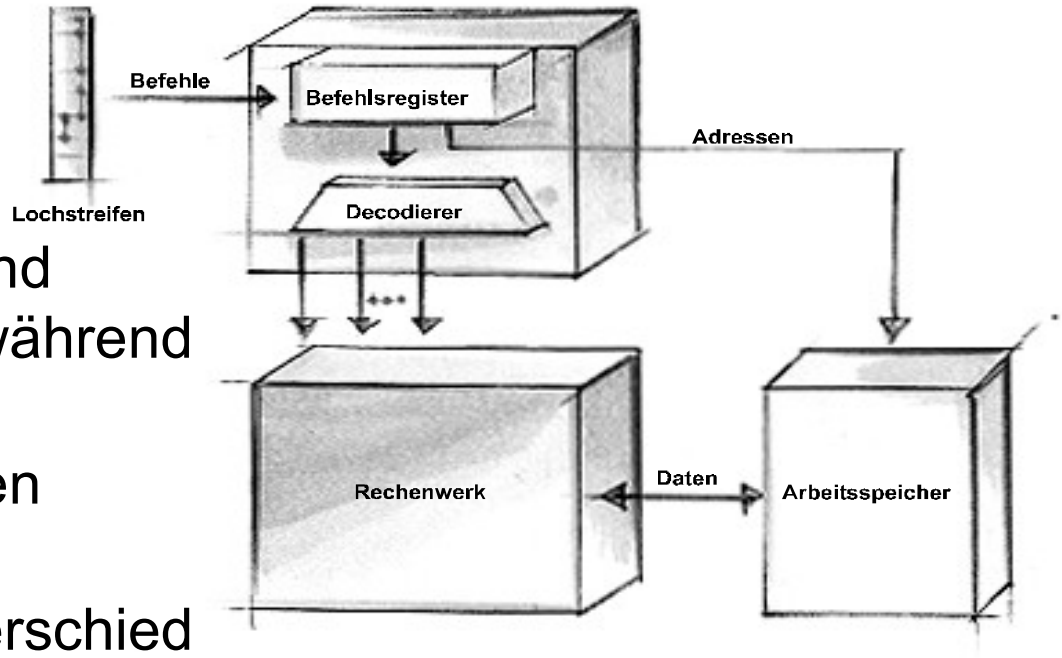
Die Entwicklung des Computers (2)

- **1946 Eniac = 1. Generation**
  (John P. Eckert und John W. Mauchly)
  - **E**lectronic **N**umerical **I**ntegrator **A**nd **C**omputer
  - Röhrentechnik ersetzt Relaistechnik (ca. 18000 Röhren)
  - Schaltzeiten gegenüber der Relaistechnik stark verringert
    - Ca. 1000 Einzelbefehle pro Sekunde
    - Addition: 0,2ms, Multiplikation: 2ms
  - Erster Einsatz 1946 in Los Alamos
  - Berechnung ballistischer Feuertabellen
  - Arithmetische Einheit als Ersatz für die Rechenmaschine
  - Speicher als Notizblock
  - Programm ersetzte Anweisungen eines menschlichen Operators



Die Entwicklung des Computers (3)

- **1946 - 1952 Neumann-Maschine** (Johann (John) von Neumann)
  - **Programm zusammen mit den Daten in einem gemeinsamen Speicher**
  - Erlaubt Programmsprünge und das Verändern der Befehle während eines Rechenvorganges
  - Programme können wie Daten behandelt werden
  - Maschine macht keinen Unterschied zwischen Befehlen und Operanden
  - **Wesentliches Konzept, welches bis heute Grundlage der Rechnertechnik ist!**
  - **"Von-Neumann-Rechner"**



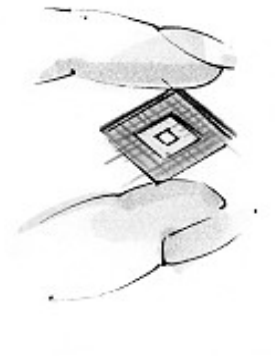
Computergenerationen von 1955 – 1990 (1)

- **1955 - 1960 Tradic = 2. Generation** (J.H. Felker)
  - **TRA**nsitor **DI**gital **C**omputer
  - Transistoren und Dioden (Halbleitertechnik)
  - Ca. 10.000 Einzelbefehle pro Sekunde
  - 700 Transistoren und 10,000 Germanium-dioden

  - Entwickelt als "Leichtgewichtcomputer" für den Flugzeugeinsatz

Computergenerationen von 1955 – 1990 (2)

- **1962 - 1970 Integrierte Schaltkreise: 3. Generation**
  - 100 Transistoren auf drei Quadratmillimetern
  - Ca. 1 Million Einzelbefehle pro Sekunde



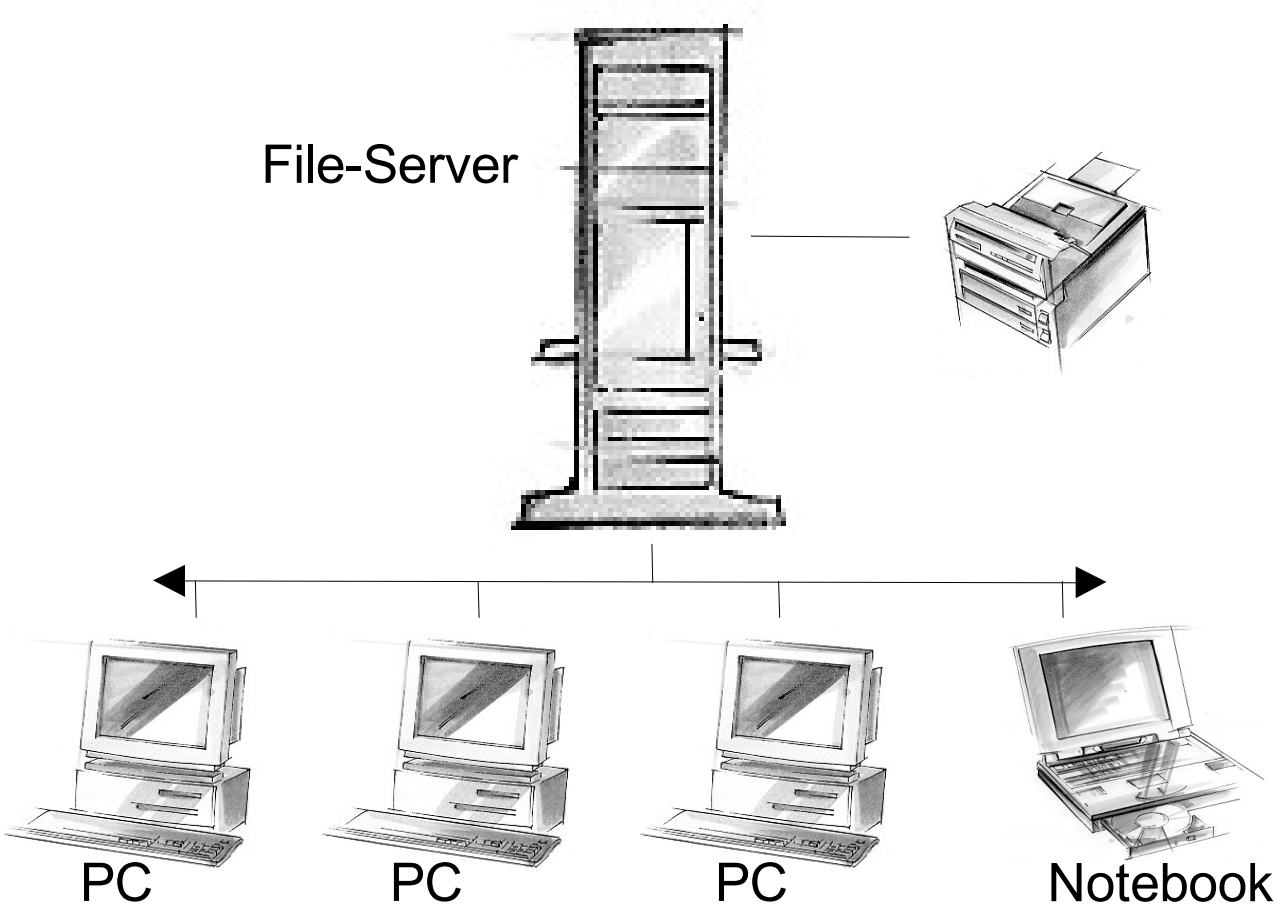
- **1968 Hochintegrierte Schaltkreise: 4. Generation**
  - Beschichtungs-, Ätz- und Aufdampfprozesse auf Siliziumscheiben
  - Ca. 10 Millionen Einzelbefehle pro Sekunde
- **1980 Cray-Computer: 5. Generation**
  - Mehrere Prozessoren werden miteinander verbunden
- **1985 Transputer**
  - "**TRANS**mitter and com**PUTER**"
  - "**TRANS**istor and com**PUTER**"

Personalcomputer

- **1974 Die ersten Homecomputer**
  - ALTAIR-8800 (Bausatz für 395 Dollar)
  - Commodore (PET)
  - Tandy Radio Shack (TRS-80)
  - Alle programmierbar in BASIC
- **1977 Apple-Computer**
- **1981 IBM-Personalcomputer**
  - Grundstein für den heutigen Personalcomputer Standard
  - Prozessor von Intel
  - Betriebssystem MS-DOS von Microsoft
- **1984 Apple Macintosh**
  - Grafische Benutzerführung

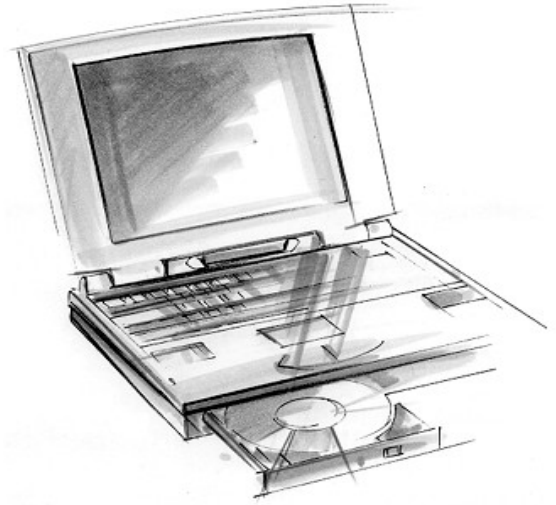
Die Vernetzung von Computern

- **Ab ca. 1985**
  - Rechnernetzwerke
  - Email
  - News
  - Talk



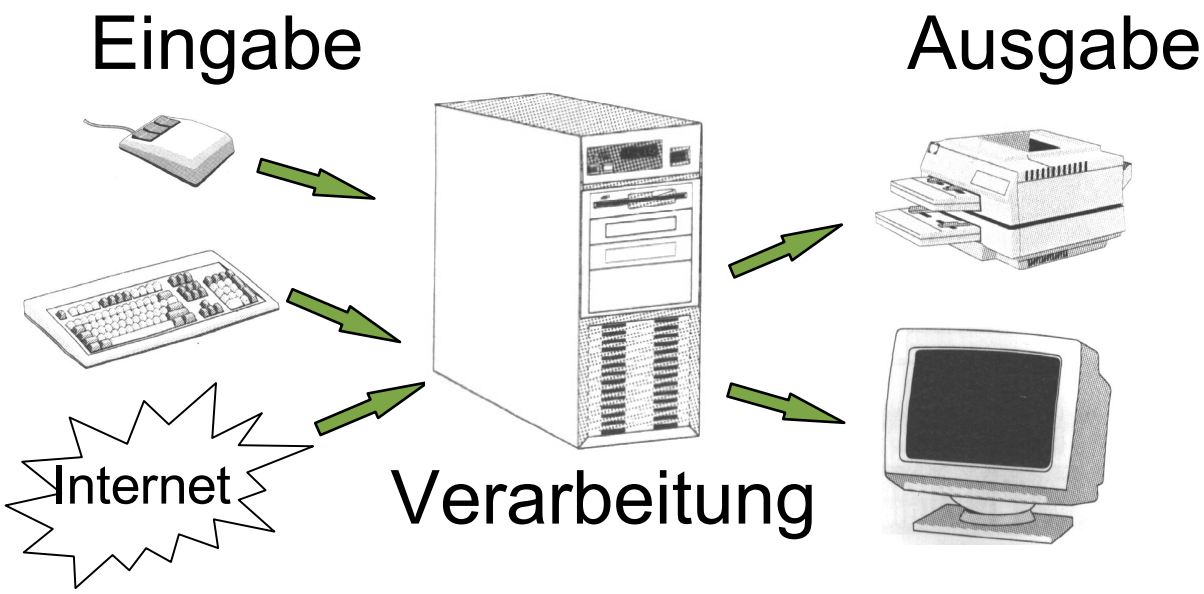
Internet / Multimedia

- **Seit 1993**
  - Internet
    - Medium für weltweite Information und Kommunikation
    - World Wide Web: WWW
  - Multimedia
    - Zusammenwirken von Text, Bild, Video und Ton auf dem Computer
  - Virtual und Augmented Reality
  - Tragbare Computer (Laptops/Notebooks/Netbooks)
  - PDAs, Smartphones
  - Multitouch-Displays



Grundlegende Funktionsweise eines Computers

- Verarbeitung von Daten nach einem fest vorgegebenen und strikt eingehaltenen Verhaltensmuster von geordneten Vorschriften
- Verhaltensmuster wird als Algorithmus bezeichnet



Informationen und Daten

- **Information: Wissen (Kenntnisse) über Sachverhalte oder Vorgänge**
- **Informationen werden üblicherweise durch Aneinanderfügen (Konkatenation) von Zeichen dargestellt**
  - Z. B. Schriftzeichen
    - Buchstaben (A bis Z), Ziffern (0 bis 9)
    - Sonderzeichen (, . - ; : ? $ § “ ! & %)
- **Daten**
  - Zum Zweck der Verarbeitung codierte Informationen
- **Zusammenhang zwischen Daten und Information nicht eindeutig**
  - Dieselbe Information kann in unterschiedlicher Weise ausgedrückt werden
    - Z. B.: Morse-Alphabet, mathematische Formelsprache, Zeichensprache

Zeichenvorrat

- **Definition:** Ein "Zeichen" ist ein Element einer endlichen Menge von unterscheidbaren Elementen, dem "Zeichenvorrat" [2]
- **Beispiele für Zeichenvorräte:**
  - Lateinische Schriftzeichen
  - Arabische Schriftzeichen
  - Blindenschrift (Brailleschrift)
  - Zeichensprache
  - Morsecode
  - Spielkartenfarben
  - Farben der Verkehrsampel

Alphabet

- **Definition: Ein Zeichenvorrat, in dem eine lineare Reihenfolge der Zeichen festgelegt ist, heißt "Alphabet"** [2]
- Begriff des Alphabets in Informatik verallgemeinert zu "endliche Menge unterschiedlicher geordneter Zeichen"
  - Darum Mengenklammern verwendet
- **Beispiele:**
  - {a, b, c, ..., A, B, C, ..., ö, ä, Ü, ...}: Alphabet der Buchstaben
  - {0,1, 2, 3, 4, 5, 6, 7, 8, 9}: Alphabet der Dezimalziffern
  - {rot, gelb, grün}: Alphabet der Verkehrsampelsignale
  - {sehr gut, ..., ungenügend}: Alphabet der Schulnoten
    - Zeichen kann auch Wort oder sogar Wortfolge sein!

Binärer Zeichenvorrat

- **Von großer technischer Bedeutung sind Zeichenvorräte, die nur aus zwei Zeichen bestehen**
  - Kann als Schalter offen/geschlossen interpretiert werden
  - Einfache Implementierung
- **Definition:** Ein Zeichenvorrat mit nur zwei Zeichen heißt "binärer Zeichenvorrat" [2]
- **Definition:** Die Zeichen eines binären Zeichenvorrats heißen "Binärzeichen" oder "Bit" [2]
  - 1 Bit **= 1 Binary Digit**
  - Ein Bit ist die kleinste mögliche Informationseinheit

Beispiele für binäre Zeichenvorräte

- **Farbenpaar** {"rot", "grün"}
- **Intensitätspaar** {"hell", "dunkel"}
- **Ziffernpaar** {0, 1}
- **Zustandspaar** {"gelocht", "ungelocht"},
  {"Vertiefung, "keine Vertiefung"}
- **Wahrheitswerte** {"falsch", "wahr"}
- **Spannungen** {0V, 5V}
- **In diesem Kurs werden wir binäre Zeichen mit dem Zeichenpaar {0, 1} darstellen**

Binärwort

- **Definition:** Ein "Wort" ist eine Folge von Zeichen, die in einem bestimmten Zusammenhang als Einheit betrachtet wird [2]

- **Definition:** Wörter über einem binären Zeichenvorrat heißen "Binärwörter" [6]
  - Ein Binärwort entsteht durch die Aneinanderreihung (Konkatenation) von Bits

- **Definition:** Die Anzahl der Binärzeichen eines Binärwortes heißt "Wortlänge" [2]

Codierung

- **Definition:** Eine "Codierung" ist eine Vorschrift für die eindeutige Zuordnung eines Zeichens eines Zeichenvorrats zu derjenigen eines anderen Zeichenvorrats [2]
  - Der Vorgang des Zuordnens der Zeichen wird als "Codieren" bezeichnet; seine Umkehrung als "Decodierung"
- **Definition:** Ein "Code" ist der bei der Codierung als Bildmenge auftretende Zeichenvorrat [2]
- **Definition:** Ein "n-Bit-Code" ist ein Zeichenvorrat, dessen Zeichen n-Bit-Binärwörter sind [6]
  - Bei technischen Codierungen werden die Zeichen des ursprünglichen Zeichenvorrats üblicherweise durch Binärwörter codiert

Codes

- Ein Bit kann zwei Zeichen eines anderen Alphabets darstellen bzw. codieren
- Mehr Zeichen können durch Zusammenfassen mehrerer Bits zu einem Binärwort dargestellt werden
  - Werden 2 Bits zusammengefasst, so lassen sich bereits 4 Zeichen als Binärwörter darstellen: 00, 01, 10, 11
  - Z. B. Codierung von Ampelzuständen:
    Rot=00, Gelb=01, Grün=10
- **Die Zeichen aller möglichen Alphabete lassen sich durch Binärwörter ausdrücken**

Beispiel

- Das Alphabet der Kleinbuchstaben ausgedrückt durch einen 5-Bit-Code
- a $\rightarrow$ 00000
  b $\rightarrow$ 00001
  c $\rightarrow$ 00010
  d $\rightarrow$ 00011
  e $\rightarrow$ 00100
  usw.
- Anzahl der codierbaren Zeichen: $2^5 = 32$
- Allgemein: Mit *n* Binärzeichen lassen sich 2^n Zeichen eines Alphabets darstellen (codieren)

Codetabelle

- Code dargestellt in Form einer **Codetabelle**
- Z. B. Codierung des Alphabets der 10 Dezimalziffern durch 4-Bit-Binärwörter:
  - Die Binärwörter 1010, 1011, 1100, 1101, 1110, 1111 werden hierbei nicht benötigt
- Die Dezimalzahl 314 würde mit dieser Codetabelle wie folgt codiert:

  0011 0001 0100
- Diese Codetabelle stellt nur eine von vielen Möglichkeiten dar, die 10 Dezimalziffern durch 4-Bit-Binärwörter zu codieren
- Die Codierung irgendeines Alphabets durch Binärwörter wird als **Binärcodierung** bezeichnet

dezimal	binär
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Texte innerhalb des Computers

- **Alphanumerische Codes**
- **Beispiel: 8-Bit-Code ISO/IEC 8859, Teil 1 (sog. Latein 1)**
  - Zeichensatz für Europa, Amerika, Australien

Bit 8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	HEX
Bit 7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
Bit 6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
Bit 5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4 3 2 1	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0 0 0 0	00		SP	0	@	P	`	p			NS	°	À	Đ	à	ð	0
0 0 0 1	01		!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0 0 1 0	02		"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0 0 1 1	03		#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0 1 0 0	04		$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0 1 0 1	05		%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0 1 1 0	06		&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0 1 1 1	07		'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1 0 0 0	08		(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1 0 0 1	09		)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1 0 1 0	10		*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1 0 1 1	11		+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1 1 0 0	12		,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1 1 0 1	13		-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1 1 1 0	14		.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1 1 1 1	15		/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Texte innerhalb des Computers

- **Linke Hälfte:**
  - 7-Bit-Code
- **Internationale Referenzversion (IRV)**
- **ISO/IRC 646: 84+12 Schriftzeichen**
- **Nationale Festlegung der Schriftzeichen der 12 grauen Felder**
- **Z. B. deutsche Referenzversion: DIN 66003**
  - [→ Ä, \ → Ö,] → Ü, {→ ä, | → ö, } → ü, ~ → ß

Bit 8	7	6	5	4	3	2	1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	HEX
0	0	0	0	0	0	0	0	00			SP	0	@	P	`	p			NS	º	À	Ð	à	ð	0
0	0	0	1	0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX									0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Texte innerhalb des Computers

- **Amerikanische Referenzversion**
- **ASCII**
  - "**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange"
- Identisch zu **IRV**

Bit																		
8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		
7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1		
6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1		
5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1		
4	3	2	1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15		
0	0	0	0	00			SP	0	@	P	`	p			NS	º	À	Đ	à	ð	0	
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1	
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2	
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3	
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4	
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5	
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6	
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7	
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8	
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9	
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A	
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B	
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C	
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D	
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E	
1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F	
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F		

Texte innerhalb des Computers

- **Rechte Hälfte: Codierung der Sprachgruppen**
- **Hier: Latein 1**
  - Zeichensatz für Europa, Amerika, Australien
- **Andere z. B.:**
  - Latein 2
    - Für osteuropäische Sprachen
  - Kyrillisch, Griechisch, Arabisch, Hebräisch etc.

Bit																		
				8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	HEX
				7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
				6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
				5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4	3	2	1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0	0	0	0	00			SP	0	@	P	`	p			NS	°	À	Đ	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Texte innerhalb des Computers

- Codes ursprünglich für Daten-kommunikation entwickelt
- Austauschcodes
- Daher Platz für Steuerzeichen vorgesehen
- 32-64 Steuerzeichen codierbar

Bit				8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	HEX
				7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
				6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
				5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4	3	2	1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0	0	0	0	00			SP	0	@	P	`	p			NS	°	À	Ð	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Steuerzeichen des 7-Bit ASCII Zeichensatzes

Übertragungssteuerzeichen

SOH	start of heading	Kopfanfang
STX	start of text	Textanfang
ETX	end of text	Textende
EOT	end of trans-mission	Übertragungsende
ENQ	enquiry	Stationsaufforderung
ACK	acknowledge	positive Rückmeldung
DLE	data link escape	Datenübertragungs-umschaltung
NACK	negative acknow-ledge	negative Rückmeldung
SYN	synchronous idle	Synchronisierung
ETB	end of trans-mission block	Datenübertragungs-block Ende

Formatsteuerzeichen

BS	backspace	Rückwärtsschritt
HT	horizontal tabulation	Zeichen-Tabulator
LF	line feed	Zeilenschritt
VT	vertical tabulation	Zeilen-Tabulator
FF	form feed	Formularvorschub
CR	carriage return	Rückführen

Code-Erweiterungszeichen

SO	shift out	Dauerumschaltung
SI	shift in	Rückschaltung
ESC	Escape	Escape

Gerätesteuerzeichen

DC1	device control one	Gerätesteuerzeichen eins
DC2	device control two	Gerätesteuerzeichen zwei
DC3	device control three	Gerätesteuerzeichen drei
DC4	device control four	Gerätesteuerzeichen vier

Informationstrennzeichen

US	unit separator	Teilgruppen-Trenn-zeichen
RS	record separator	Untergruppen-Trenn-zeichen
GS	group separator	Gruppen-Trenn-zeichen
FS	file separator	Hauptgruppen-Trenn-zeichen

Sonstige Steuerzeichen

NUL	null	Null
BEL	bell	Klingel
CAN	cancel	ungültig
EM	end of medium	Aufzeichnungsende
SUB	substitute	Substitution

Texte innerhalb des Computers

- **Heute: Universelle Codes**

- **Unicode**
  - Zeichencodierung mit variabler Bitzahl
  - Prominenter Vertreter: UTF-16LE
    - Zeichen durch 1 bis 2 16 Bit-Wörter codiert
  - Bei 16-Bit Version: 2^{16} Zeichen = 65536 Zeichen darstellbar
  - Die wichtigsten lebenden Sprachen sind abgedeckt
  - Latein 1 entspricht den ersten 256 Plätzen

Texte innerhalb des Computers

- **UCS (32)**
  - Universal Character Set nach ISO/IEC 10646
  - 2^{32} Zeichen = 4 294 967 296 Zeichen darstellbar
  - Unterteilt in 128 Gruppen zu 256 Ebenen zu 256 Reihen zu 256 Zellen
  - *Latein 1* ist in erster Reihe der ersten Ebene
  - Alle Schriftzeichen aus allen lebenden und toten Sprachen darstellbar

Beispiel

- **Die folgende ASCII Zeichenfolge soll codiert werden:**

  Info1

Bit																		
				8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
				7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	HEX
				6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
				5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4	3	2	1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0	0	0	0	00			SP	0	@	P	`	p			NS	º	À	Đ	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Übung

- **Codieren Sie die folgende ASCII Zeichenfolge als binären Datenblock:**

  Computer?

Bit																			
				8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
				7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	HEX
				6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
				5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4	3	2	1		00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0	0	0	0	00			SP	0	@	P	`	p			NS	°	À	Ð	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Übung

- **Decodieren Sie den folgenden binären Datenblock:**

01001000
01101001
00100000
01110100
01101000
01100101
01110010
01100101
00100001

Bit					8	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	HEX
					7	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
					6	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
					5	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4	3	2	1			00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	
0	0	0	0	00			SP	0	@	P	`	p			NS	°	À	Ð	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o	DEL			¯	¿	Ï	ß	ï	ÿ	F
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Zahlendarstellungen

Dualzahlen

- **Computer arbeiten intern im Zahlensystem zur Basis 2, mit den "Dualziffern" 0 und 1**
  - Duales Zahlensystem wurde im 17. Jahrhundert von Gottfried Wilhelm Leibniz entwickelt
- **Definition: "Dualzahlen" sind Wörter aus Dualziffern; damit sind sie ebenfalls Binärwörter**
- **Binärwörter bzw. Dualzahlen dienen zur Darstellung jedweder Information in einem Computer**
  - Hierzu gehören - neben den zu verarbeitenden Daten - ebenfalls die Maschinenbefehle

Bits und Bytes

- **Maßeinheiten für die Kapazität von Speichermedien**
  - 1 Bit = 1 Zeichen (0 oder 1)
  - 1 Byte = 8 Bit (256 Zeichen)
  - 1 kB = 1000 Byte
  - 1 KiB = 1 Kibibyte = 1024 Byte = 2^{10} Byte
  - 1 MB = 1000 kB
  - 1 MiB = 1 Mebibyte = 1048576 Byte = 2^{10} KiB
  - usw.: Gigabyte (GB) und Gibibyte (GiB), Terabyte (TB) und Tebibyte (TiB), Petabyte (PB) und Pebibyte (PiB)...

Interne Verarbeitung mit dem Dualsystem

- **Interne Codierung der zwei Zustände:**

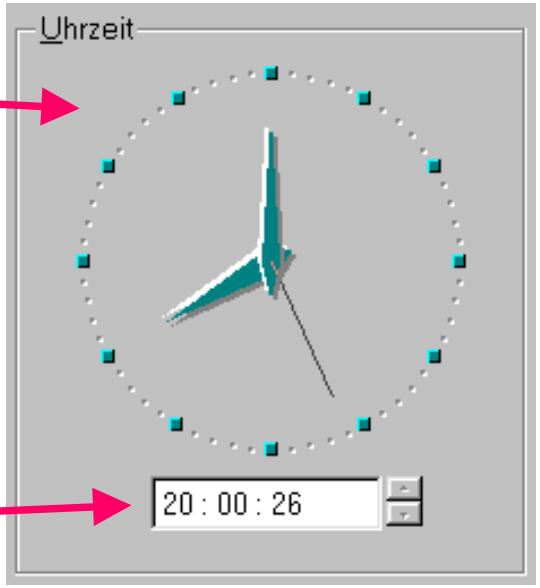
0	**1**
Strom aus	Strom an
Ladung	Keine Ladung
Nicht magnetisch	Magnetisch
Keine Vertiefung	Vertiefung

- Bisher haben wir nur betrachtet, wie Text in Form von Binärwörtern ausgedrückt werden kann

- **Neue Fragestellung:** Wie lassen sich *physikalische Größen* und *Dezimalzahlen* im Dualsystem darstellen und wie lassen sich Berechnungen in diesem System durchführen?

Analoge und digitale Daten

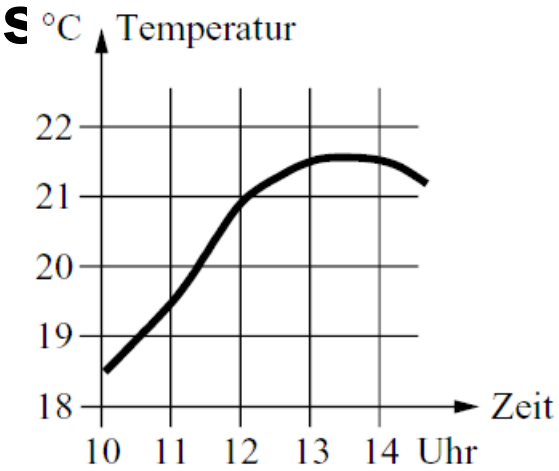
- **Analoge Daten**
  - Analog = entsprechend, vergleichbar
  - Werte ändern sich stufenlos
  - Darstellung durch kontinuierliche Kurve, Skala, Zeigerstellung
  - Beispiel: Zeigerstellung der Uhr

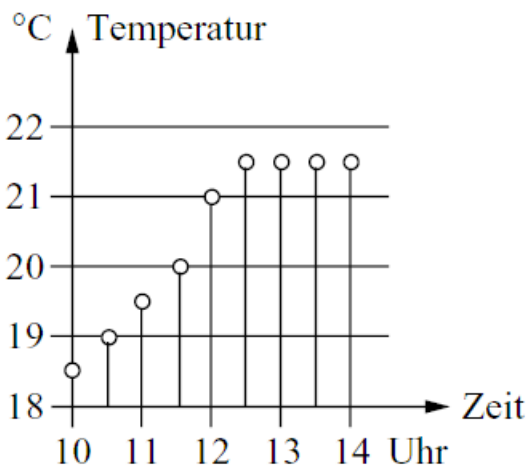


- **Digitale Daten**
  - Digit (engl.) = Zahl
  - Daten werden durch diskrete Zahlenwerte dargestellt
  - Innerhalb des Computers lassen sich Daten nur digital verarbeiten
  - Vor ihrer Verarbeitung müssen analoge Daten **digitalisiert** werden
  - Durch die Digitalisierung geht Information verloren
  - Abstand zwischen Zahlenwerten muss so gering sein, dass Verlust tolerierbar ist

Digitalisierung (1)

- **Zeitlich und betragsmäßig kontinuierliche Werte werden durch zeit- und betragsdiskrete Dualzahlen dargestellt**
  - Abtastung und Quantisierung
- **Beis**






```
10111001
10111110
11000011
11001000
11010010
11010111
11010111
11010111
11010111
```

- Analoge Darstellung:
- Zeit- und wertkontinuierlich

- Diskretisierung:
- Zeit- und wertdiskret
- Zwischenschritt

- Digitalisierung:

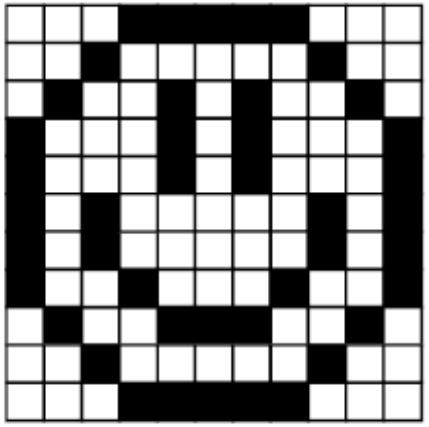
Verschlüsselung im Dualsystem

Digitalisierung (2)

- **Beispiel: Schwarz-Weiß-Graphik**

- Analoge Darstellung:

  Orts- und wertkontinuierlich



- Diskretisierung:

  Orts- und wertdiskret

0	0	0	1	1	1	1	1	0	0	0
0	0	1	0	0	0	0	0	1	0	0
0	1	0	0	1	0	1	0	0	1	0
1	0	0	0	1	0	1	0	0	0	1
1	0	0	0	1	0	1	0	0	0	1
1	0	1	0	0	0	0	0	1	0	1
1	0	1	0	0	0	0	0	1	0	1
1	0	0	1	0	0	0	1	0	0	1
0	1	0	0	1	1	1	0	0	1	0
0	0	1	0	0	0	0	0	1	0	0
0	0	0	1	1	1	1	1	0	0	0

- Digitalisierung:

  Verschlüsselung im Dualsystem

- **Farbzeichnungen:**
  - Farbe lässt sich z. B. als RGB-Farbwerte codieren
  - Z. B. ein Byte pro Farbkanal

Zahlenkonvertierung

Zahlenkonvertierung

- Messwertaufnehmer liefern Ergebnisse als Dualzahlen
- Manuelle Eingabe von Zahlen über Tastatur und Ausgabe von Ergebnissen ebenfalls erforderlich
- Wir benutzen normalerweise das Dezimalsystem für die Darstellung von Zahlenwerten
  - Eingabe/Ausgabe von Zahlenwerten im Dualsystem ungewohnt...
- **Deshalb Zahlenkonvertierung notwendig**
  - Eingabe: Konvertierung von Dezimal- in Dualzahlen
  - Ausgabe: Konvertierung von Dual- in Dezimalzahlen

Dezimalzahlen

- **Wenn wir eine Dezimalzahl $Z = ...a_2a_1a_0,a_{-1}a_{-2}...$ schreiben, so ist dies eine abgekürzte Schreibweise für:**

  $Z = ...a_2 * 10^2 + a_1 * 10^1 + a_0 * 10^0 + a_{-1} * 10^{-1} + a_{-2} * 10^{-2}...$

  - Die Ziffern sind entsprechend ihres Gewichts von links nach rechts angeordnet
  - Einerstelle durch nachgestelltes Komma markiert
- Die Ziffern variieren im Dezimalsystem zwischen 0 und 9
- Gewichte sind Potenzen der Basis 10
- **Wie lassen sich Dezimalzahlen als Dualzahlen darstellen?**

Allgemeine Darstellung

- **Darstellung natürlicher Zahlen in B-adischen Systemen:**

- $Z = a_n \cdot B^n + a_{n-1} \cdot B^{n-1} + \ldots + a_1 \cdot B^1 + a_0 \cdot B^0 = \sum_{i=0}^{n} a_i \cdot B^i$

- Z = Zahl im B-adischen Zahlensystem
- B = Basis des Zahlensystems,
  z. B. 10 (dezimal), 2 (dual), 8 (oktal),
  16 (hexadezimal)
- B^i = Wertigkeit der i-ten Stelle
- a_i = Ziffer (Koeffizient) der i-ten Stelle aus der Menge der Ziffern {0, 1, 2, ..., B-1}

Konvertierung von Dezimalzahlen in Dualzahlen

- **Satz: Die ziffernmäßige Darstellung einer ganzen nicht-negativen Zahl z im Zahlensystem zur Basis B>=2,**

  $Z = a_n a_{n-1} \ldots a_1 a_0$

  ergibt sich aus folgender Vorschrift (Horner-Schema):

$$
\begin{aligned}
Z &= q_0 \cdot B + a_0, \\
q_0 &= q_1 \cdot B + a_1, \\
q_1 &= q_2 \cdot B + a_2, \\
&\vdots \\
q_k &= q_{k+1} \cdot B + a_{k+1}, \\
&\vdots \\
q_{n-1} &= q_n \cdot B + a_n = 0 \cdot B + a_n = a_n
\end{aligned}
$$

- Für Dualsystem: $B = 2$!
- Die Ziffern $a_0,\ldots,a_n$ werden berechnet, indem die Zahl Z durch B dividiert und der Divisionsrest notiert wird
- Der letzte Divisionsrest liefert a_n

- **Die Divisionsreste a_i (i=0,...,n) sind die gesuchten Ziffern**

Beispiel

- **743 soll im Dualsystem dargestellt werden**
- $743 \div 2 = 371$ Rest 1: a_0
  $371 \div 2 = 185$ Rest 1: a_1
  $185 \div 2 = 92$ Rest 1: a_2
  $92 \div 2 = 46$ Rest 0: a_3
  $46 \div 2 = 23$ Rest 0: a_4
  $23 \div 2 = 11$ Rest 1: a_5
  $11 \div 2 = 5$ Rest 1: a_6
  $5 \div 2 = 2$ Rest 1: a_7
  $2 \div 2 = 1$ Rest 0: a_8
  $1 \div 2 = 0$ Rest 1: a_9

$$
\begin{aligned}
Z &= q_0 \cdot B + a_0, \\
q_0 &= q_1 \cdot B + a_1, \\
q_1 &= q_2 \cdot B + a_2, \\
&\vdots \\
q_k &= q_{k+1} \cdot B + a_{k+1}, \\
&\vdots \\
q_{n-1} &= q_n \cdot B + a_n = \\
&\quad 0 \cdot B + a_n = a_n
\end{aligned}
$$

- $743_{10} = 1011100111_2$

Beispiel

- **743 soll im Dualsystem dargestellt werden**

- $743 \div 2 = 371$ Rest 1: a_0 ← Least Significant Bit: LSB
  $371 \div 2 = 185$ Rest 1: a_1
  $185 \div 2 = 92$ Rest 1: a_2
  $92 \div 2 = 46$ Rest 0: a_3
  $46 \div 2 = 23$ Rest 0: a_4
  $23 \div 2 = 11$ Rest 1: a_5
  $11 \div 2 = 5$ Rest 1: a_6
  $5 \div 2 = 2$ Rest 1: a_7
  $2 \div 2 = 1$ Rest 0: a_8
  $1 \div 2 = 0$ Rest 1: a_9 ← Most Significant Bit: MSB

- $743_{10} = 1011100111_2$

Konvertierung von Dualzahlen in Dezimalzahlen

- **Satz:** Ist in einem Zahlensystem zur ganzzahligen Basis $B \geq 2$ mit den Ziffern $0 \leq a_i < B$ eine nicht-negative ganze Zahl Z dargestellt durch

  $$Z = a_n a_{n-1} \ldots a_1 a_0$$

  dann ist

  $$Z = \sum_{i=0}^{n} a_i \cdot B^i = a_n \cdot B^n + a_{n-1} \cdot B^{n-1} + \ldots + a_1 \cdot B^1 + a_0 \cdot B^0$$

  Diese Darstellung ist eindeutig

Beispiele

- **Zehnersystem**

210 (Stelle)

743

3 Einer $= 3 * 10^0 = 3$

4 Zehner $= 4 * 10^1 = 40$

7 Hunderter $= 7 * 10^2 = 700$

bzw.

$Z = 743 = 7*10^2 + 4*10^1 + 3*10^0$

- **Dualsystem**

98 7654 3210

10 1110 0111

1 * 1		$= 1 * 2^0$
1 * 2		$= 1 * 2^1$
1 * 4		$= 1 * 2^2$
0 * 8		$= 0 * 2^3$
0 * 16		$= 0 * 2^4$
1 * 32		$= 1 * 2^5$
1 * 64		$= 1 * 2^6$
1 * 128		$= 1 * 2^7$
0 * 256		$= 0 * 2^8$
1 * 512		$= 1 * 2^9$
= 743		

Übungen

- **Konvertieren Sie die Dezimalzahl 813 in das Dualsystem**
- **Berechnen Sie zur Dualzahl 11010010011100 die zugehörende Dezimalzahl**

Darstellung gebrochener Zahlen

- **Darstellung gebrochener Zahlen in B-adischen Systemen:**
- $$Z = a_n \cdot B^n + a_{n-1} \cdot B^{n-1} + \ldots + a_1 \cdot B^1 + a_0 \cdot B^0$$
  $$+ a_{-1} \cdot B^{-1} + a_{-2} \cdot B^{-2} + \ldots + a_{-(m-1)} \cdot B^{-(m-1)} + a_{-m} \cdot B^{-m}$$
  $$= \sum_{i=-m}^{n} a_i \cdot B^i = \sum_{i=0}^{n} a_i \cdot B^i + \sum_{i=-m}^{-1} a_i \cdot B^i$$
  $$= Z_v + Z_n$$
  - Z_v: Vorkommastellen, Z_n: Nachkommastellen
- **Beide Teile werden unabhängig voneinander konvertiert**

Konvertierung der Nachkommastellen (1)

- **Erfolgt wieder nach dem Horner-Schema:**
- $$\sum_{i=-m}^{-1} a_i \cdot B^i = (\ldots(a_{-m} \div B + a_{-(m-1)}) \div B + \ldots + a_{-1}) \div B$$
- Die Basis *B* tritt jetzt allerdings im Nenner auf
- Zur Bestimmung der Koeffizienten a_i wird daher nicht mehr durch die Basis *B* geteilt, sondern mit ihr multipliziert
- Die hierbei entstehenden Produkte bestehen aus einem ganzzahligen Anteil und Nachkommastellen

Konvertierung der Nachkommastellen (2)

- **Der ganzzahlige Anteil ist jeweils der Koeffizient**
- Mit den Nachkommastellen wird die Rechnung wiederholt bis diese verschwinden oder eine gegebene Genauigkeit erreicht ist

- **Beispiel: 0,6875 soll im Dualsystem dargestellt werden**

← Most Significant Bit: MSB

0,6875	* 2	=	0,375	+	1:	a_{-1}
0,375	* 2	=	0,75	+	0:	a_{-2}
0,75	* 2	=	0,5	+	1:	a_{-3}
0,5	* 2	=	0.0	+	1:	a_{-4}

← Least Significant Bit: LSB

$0{,}6875_{10} = 0{,}1011_2$

Beispiel

- **0,24 soll im Dualsystem dargestellt werden**

$$
\begin{aligned}
0{,}24 \;*\; 2 &= 0{,}48 + 0: \; a_{-1} \\
0{,}48 \;*\; 2 &= 0{,}96 + 0: \; a_{-2} \\
0{,}96 \;*\; 2 &= 0{,}92 + 1: \; a_{-3} \\
0{,}92 \;*\; 2 &= 0{,}84 + 1: \; a_{-4} \\
0{,}84 \;*\; 2 &= 0{,}68 + 1: \; a_{-5} \\
0{,}68 \;*\; 2 &= 0{,}36 + 1: \; a_{-6} \\
0{,}36 \;*\; 2 &= 0{,}72 + 0: \; a_{-7} \\
0{,}72 \;*\; 2 &= 0{,}44 + 1: \; a_{-8}
\end{aligned}
$$

Abbruch nach 8 Stellen

$$0{,}24_{10} \approx 0{,}00111101_{2}$$

Beispiel

- **0,1 soll im Dualsystem dargestellt werden**

0,1	* 2	=	0,2	+	0:	a_{-1}
0,2	* 2	=	0,4	+	0:	a_{-2}
0,4	* 2	=	0,8	+	0:	a_{-3}
0,8	* 2	=	0,6	+	1:	a_{-4}
0,6	* 2	=	0,2	+	1:	a_{-5}

Periodische Zahl!

$$0,1_{10} = 0,0\overline{0011}_2$$

Übung

- **Konvertieren Sie die Dezimalzahl 4,8125 in das Dualsystem**

Rückkonvertierung der Nachkommastellen

- **Einsetzen der Koeffizienten in die allgemeine Gleichung zur Darstellung der Nachkommastellen:**

$$Z_n = a_{-1} \cdot B^{-1} + a_{-2} \cdot B^{-2} + \ldots + a_{-(m-1)} \cdot B^{-(m-1)} + a_{-m} \cdot B^{-m}$$
$$= \sum_{i=-m}^{-1} a_i \cdot B^i$$

- **Beispiel:**

$$0{,}1011_2 = 1*2^{-1} + 0*2^{-2} + 1*2^{-3} + 1*2^{-4}$$
$$= 0{,}5 + 0{,}0 + 0{,}125 + 0{,}0625$$
$$= 0{,}6875$$

Übung

- **Konvertieren Sie die Dualzahl $10110{,}0111_2$ in das Dezimalsystem**

Andere Zahlensysteme

- **Bisher nur zwei Zahlenbasen betrachtet:**
  - 10 (Dezimalsystem)
  - 2 (Dualsystem)
- **Konvertierungsalgorithmen allgemein anwendbar**
- **Beispiele: Oktalsystem, Hexadezimalsystem**

Oktalsystem

- **Beispiel:**
- **743 soll im Oktalsystem dargestellt werden**
- $743 \div 8 = 92$ Rest 7: a_0
  $92 \div 8 = 11$ Rest 4: a_1
  $11 \div 8 = 1$ Rest 3: a_2
  $1 \div 8 = 0$ Rest 1: a_3

  $743_{10} = 1347_8$
- **Rückkonvertierung oktal nach dezimal:**

  $1*8^3 + 3*8^2 + 4*8^1 + 7*8^0$

  $= 1*512 + 3*64 + 4*8 + 7*1 = \underline{743}$

Hexadezimalsystem (1)

- **System zur Basis 16**

Dezimal-zahl	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hexa-dezimal-ziffer	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Hexadezimalsystem (2)

- **Beispiel:**
- **743 soll im Hexadezimalsystem dargestellt werden**
- 743 ÷ 16 = 46 Rest 7: a_0

  46 ÷ 16 = 2 Rest E: a_1

  2 ÷ 16 = 0 Rest 2: a_2

  $743_{10} = 2E7_{16}$
- **Rückkonvertierung hexadezimal nach dezimal:**

  $2*16^2 + E*16^1 + 7*16^0$

  $= 2*256 + E*16 + 7*1 = 2*256 + 14*16 + 7*1 =$

  743

0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Hexadezimalsystem (3)

- **Hauptsächlich zur kompakten Darstellung von Binärdaten eingesetzt**
  - Z. B. für logische Operationen, Register- und Speicherinhalte
- **Beispiel: 32-Bit-Maske für logische UND-Operation:**
- **$0000\,1111\,0000\,1001\,0000\,0000\,0000\,0000_2$ =**

$0\quad F\quad 0\quad 9\quad 0\quad 0\quad 0\quad 0_{16}$

Kennzeichnung von Zahlensystemen

- **Binär: B**
  - Bsp.: **B**1011100111 oder **%**1011100111
- **Hexadezimal: 0x**
  - Bsp.: **0x**2E7 oder **0X**2E7 oder **$**2E7 oder **0x**2e7 oder **0X**2e7 oder **$**2e7
- **Oktal: 0** (Null)
  - Bsp.: **0**1347
- **Dezimal:**
  - Bsp.: 743 (keine spezielle Markierung vorangestellt)

Dual ↔ Oktal ↔ Hexadezimal

- **Alle drei Zahlensysteme haben als Basis eine Zweierpotenz**
  - dual: $B=2^1$, oktal: $B=2^3$, hexadezimal: $B=2^4$
- Dadurch einfache Konvertierung zwischen den Systemen möglich
- Beispiele:
- dual ↔ hexadezimal

B 0010 1110 0111 (Aufteilen in Vierergruppen)

0x 2 E 7

- dual ↔ oktal

B 001 011 100 111 (Aufteilen in Dreiergruppen)

0 1 3 4 7

Beliebige Basen

- 743 als Zahl zur Basis 7:
- $743 \div 7 = 106$ Rest 1: a_0

  $106 \div 7 = 15$ Rest 1: a_1

  $15 \div 7 = 2$ Rest 1: a_2

  $2 \div 7 = 0$ Rest 2: a_3

  $743_{10} = 2111_7$
- Rückkonvertierung nach dezimal:

  $2*7^3 + 1*7^2 + 1*7^1 + 1*7^0$

  $= 2*343 + 1*49 + 1*7 + 1 = \underline{743}$

Übung

- **Berechnen Sie die Lösung von**

  $435_7 + 44_5$

  und geben Sie das Ergebnis im Dualsystem und Hexadezimalsystem an

Rechnen mit Dualzahlen und Codesicherung

Arithmetik im Dualsystem: Addition

- **Funktioniert im Prinzip wie im Dezimalsystem**
  - 0+0 = 0, 0+1 = 1, 1+0 = 1, 1+1 = 10
  - Bei 1+1 entsteht ein Übertrag (carry bit) von 1
    - Übertrag erfordert zusätzliche Stelle
- Beispiele
- Dezimalsystem:
  5 + 6 = 11

  Dualsystem:

```
   0 1 0 1
+  0 1 1 0
     1          Übertrag
-------------
   1 0 1 1
```

- Dezimalsystem:
  5 + 12 = 17

  Dualsystem:

```
   0 1 0 1
+  1 1 0 0
   1 1          Übertrag
-------------
1  0 0 0 1
```

Übung

- **Führen Sie die folgende Addition im Dualsystem durch:**

$$47_{10} + 23_{10} = ?_2 + ?_2 = ?_2$$

Zahlenkonvertierung: Fortsetzung

- Bisher nur positive Zahlen betrachtet
- Verarbeitung negativer Zahlen ebenfalls erforderlich
- Wie lässt sich -743_{10} als Dualzahl darstellen?
- Darstellung einer Dualzahl mit Vorzeichen nicht möglich, da das interne Alphabet nur aus der Menge B = {0,1} besteht
  - D. h. folgendes ist NICHT möglich: Z = -1011100111_2
- Codierung des Vorzeichens mittels des Zeichenvorrates B = {0,1} benötigt!

Negative Zahlen

Darstellung negativer Zahlen

- Naheliegendste Lösung:

  Darstellung des Vorzeichens mittels zusätzlichem Bit am Anfang der Dualzahl

- **Vorzeichen-Betrags-Darstellung**
- Beispiel für dreistellige Dualzahl:

  $4_{10} = 100_2$

- Einführen des zusätzlichen Bits mit

  0: positive Zahl und

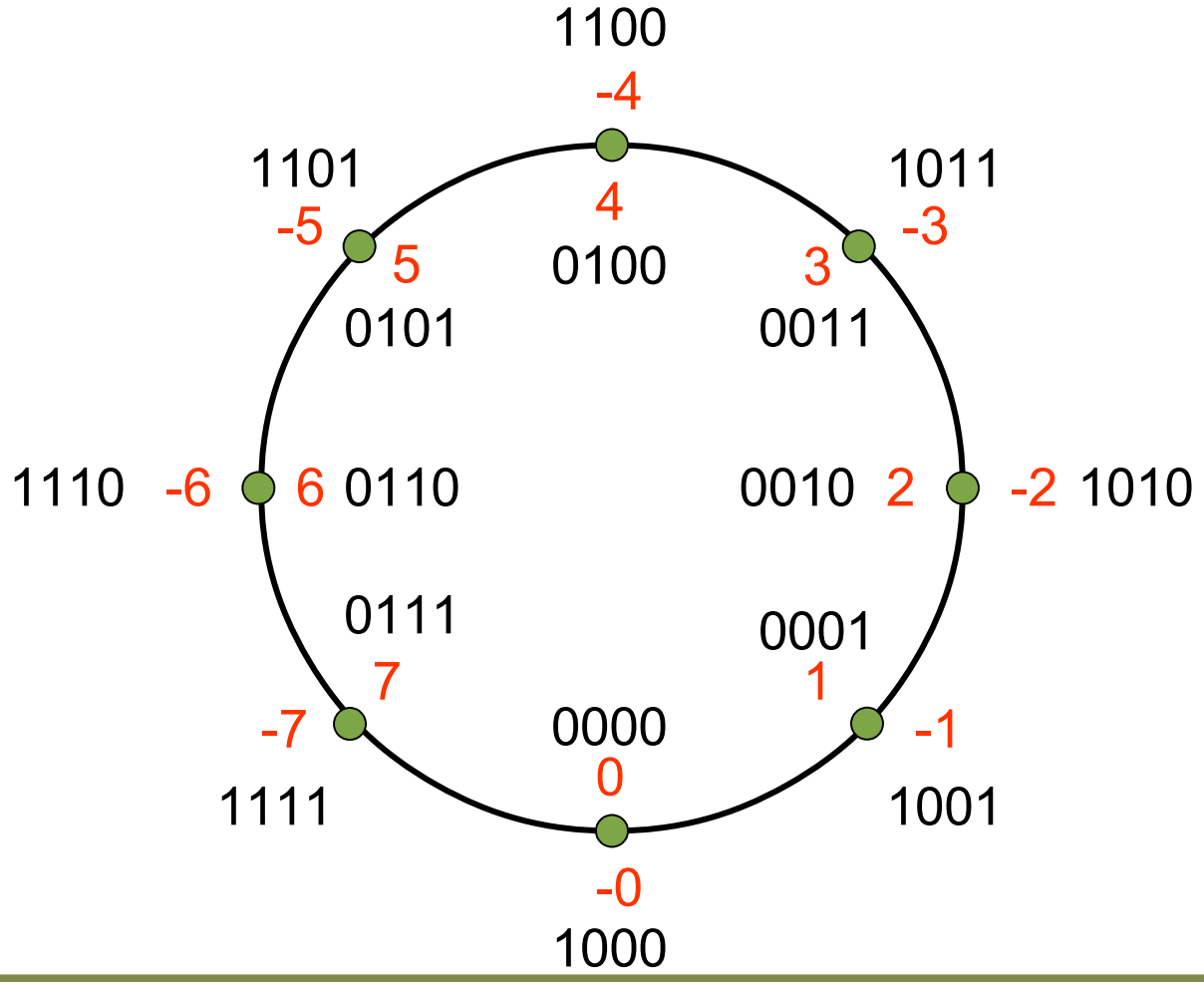
  1: negative Zahl

  ermöglicht:

  $4_{10} = 0100_2$, $-4_{10} = 1100_2$

Zahlenkreis

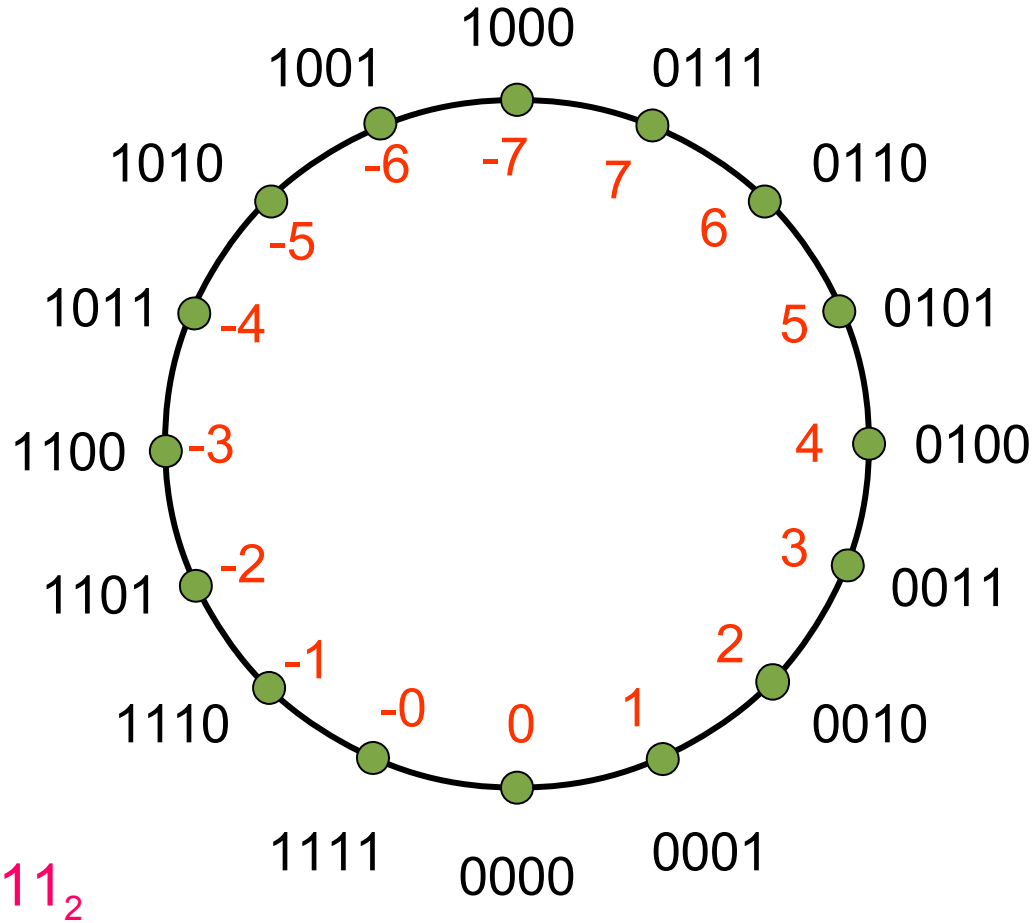
- Ermöglicht Visualisierung einer Zahlendarstellung
- Zahlenkreis für die **Vorzeichen-Betrags-Darstellung**:



- Darstellung nicht homogen
  - Vorzeichenstelle erfordert gesonderte Behandlung
- Gut für Multiplikation/ Division geeignet, weniger für Addition/Subtraktion
- -0=1000 ausgeschlossen

1-Komplement-Darstellung

- Erzielt durch Bit-weises Invertieren der positiven Dualzahl
- Höchstwertiges Bit steht auch hier wieder für das Vorzeichen:
  - 0: positive Zahl,
    1: negative Zahl
- Es existiert eine redundante Darstellung der Null!
- Beispiel:
  $4_{10}+(-4_{10}) = 0100_2+1011_2 = 1111_2$



2-Komplement-Darstellung

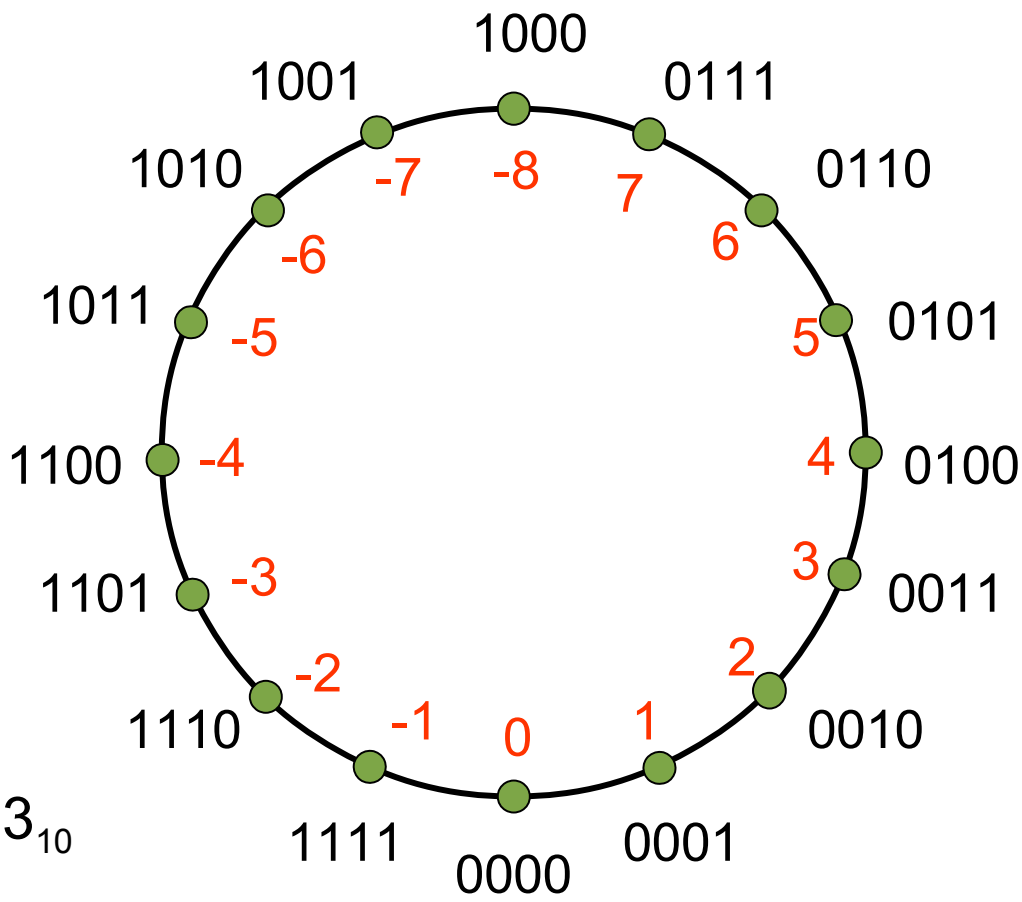
- Erzielt durch Bit-weises Invertieren der positiven Dualzahl mit anschließender Addition von 1
- Einzig gebräuchliche Darstellung negativer Zahlen
- (Two's complement numbers)
- Beispiele:

$4_{10}+(-4_{10}) = 0100_2+1100_2 = 0000_2$

$2_{10}-5_{10} = 0010_2+1011_2 = 1101_2 = -3_{10}$

$7_{10}-6_{10} = 0111_2+1010_2 = 0001_2 = 1_{10}$

$2_{10}+4_{10} = 0010_2+0100_2 = 0110_2 = 6_{10}$



2-Komplement-Bildung

- Invertieren eines jeden Bits und Addieren von Eins
- Beispiel: -743?
- 743_{10} = 0$1011100111_2$

1. Invertierung: `10100011000`

2. Addition von Eins: `+ 1`

2-Komplement: `10100011001 = -743`

- Rückkonvertierung:

1. Invertierung: `01011100110`

2. Addition von Eins: `+ 1`

2-Komplement: `01011100111 = +743`

Übung

- Führen Sie die folgende Subtraktion mittels 2-Komplement-Zahlen durch

  47_{10}-23_{10} = ?$_2$

- Geben Sie das Ergebnis als **Dualzahl** an

Arithmetik im Dualsystem: Subtraktion

- In früheren Prozessoren Subtraktion durch Addition des Subtrahenden in 2-Komplement-Darstellung
- In heutigen Prozessoren echte Subtraktion implementiert!
- Beispiel:

	47_{10}	$1\ 0\ 1\ 1\ 1\ 1_2$
-	28_{10}	$-0\ 1\ 1\ 1\ 0\ 0_2$
	1	1
=	19_{10}	$0\ 1\ 0\ 0\ 1\ 1_2$

Arithmetik im Dualsystem: Überlauf (1)

- Prozessoren verarbeiten Daten in festen Wortlängen
  - Z. B. 8, 16, 32, 64 Bits
- Zahlenbereich daher begrenzt
- Überschreitung des Zahlenbereichs muss signalisiert werden
- Bereichsüberschreitung wird als Bit in einem **Prozessorstatusregister** gespeichert und ist per Befehl abfragbar
  - Übertragsbit c (*carry bit*)
- Prozessorstatusregister speichert ebenfalls, ob Ergebnis gleich Null ist
  - Nullbit z (*zero bit*)

Arithmetik im Dualsystem: Überlauf (2)

- **Signalisierung eines Überlaufs:**
  - Überlauf tritt auf bei Übertrag in das höchstwertige Bit oder in die nicht verfügbare Stelle oberhalb des höchstwertigen Bit
  - Kein Überlauf, wenn Übertrag in beide Stellen
- **Beispiele für 4-Bit-Dualzahlen:**
- $7_{10} + 5_{10} =$

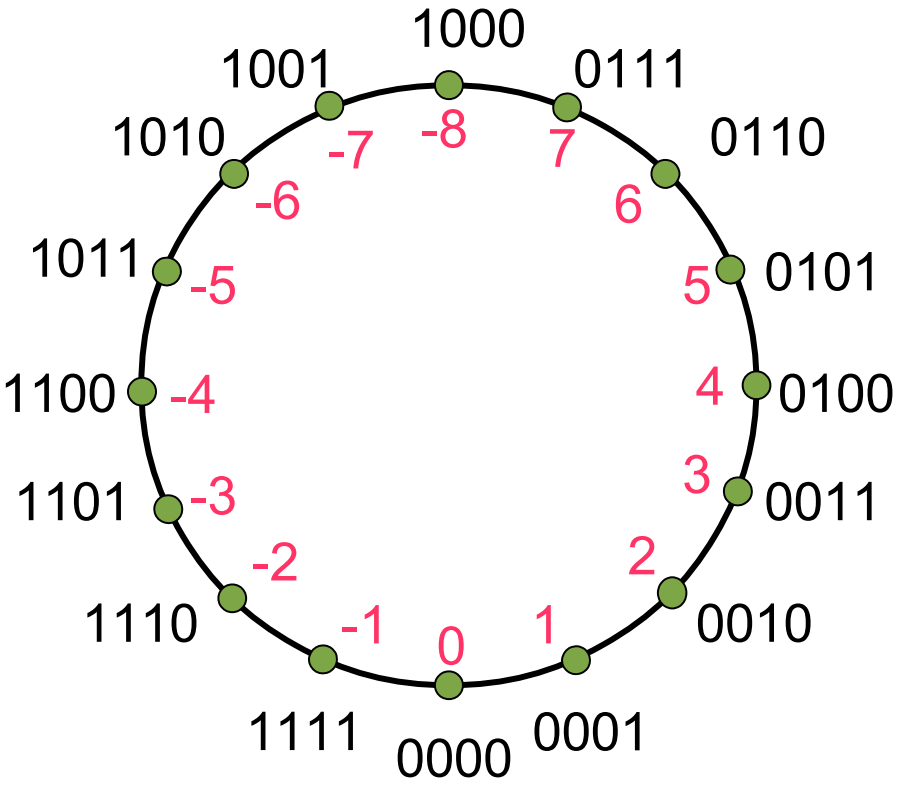
$$\begin{array}{r} 0\ 1\ 1\ 1_2 \\ +\,0\ 1\ 0\ 1_2 \\ \hline 1\ 1\ 0\ 0 = -4_{10} \end{array}$$

Überlauf: 1 1 1

- $5_{10} + (-4)_{10} =$

$$\begin{array}{r} 0\ 1\ 0\ 1_2 \\ +\,1\ 1\ 0\ 0_2 \\ \hline 0\ 0\ 0\ 1 = 1_{10} \end{array}$$

Kein Überlauf: 1 1



Multiplikation mit Dualzahlen

- "Kleines 1-mal-1" für Dualzahlen: 0*0 = 0, 0*1 = 0, 1*0=0, 1*1=1
  - Bei mehrstelligen Zahlen Behandlung entsprechend Dezimalsystem
- Beispiel: $86_{10}*21_{10}$

  $1010110_2*10101_2$:

```
                1 0 1 0 1 1 0                  10101
+               0 0 0 0 0 0 0                   0101
+                 1 0 1 0 1 1 0                  101
+                   0 0 0 0 0 0 0                 01
+                     1 0 1 0 1 1 0               1
-------------------------------------------------------
              1 1 1 0 0 0 0 1 1 1 0₂ = 1806₁₀
```

- Grundoperationen: Kopieren und Verschieben

Multiplikation mit Dualzahlen: Andere Richtung

- $1010110_2 * 10101_2$:

$$
\begin{array}{rcccccccccc}
 & & & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 10101 \\
+ & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 10100 \\
+ & & 1 & 0 & 1 & 0 & 1 & 1 & 0 & & 10001 \\
+ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & 10000 \\
+ & 1 & 0 & 1 & 0 & 1 & 1 & 0 & & & 10001 \\
\hline
 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1\,0_2 = 1806_{10}
\end{array}
$$

Übung

- Multiplizieren Sie

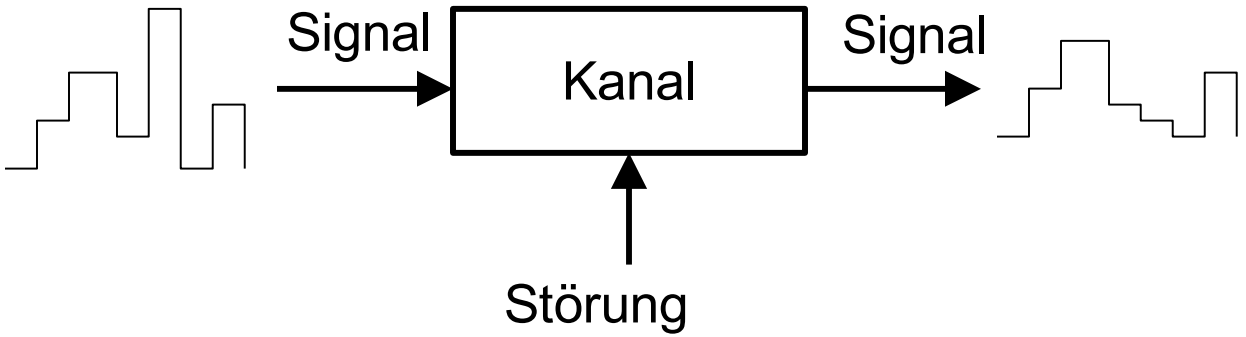
  $13_{10} * 11_{10}$

  im **Dualsystem**

Codesicherung

Codesicherung (1)

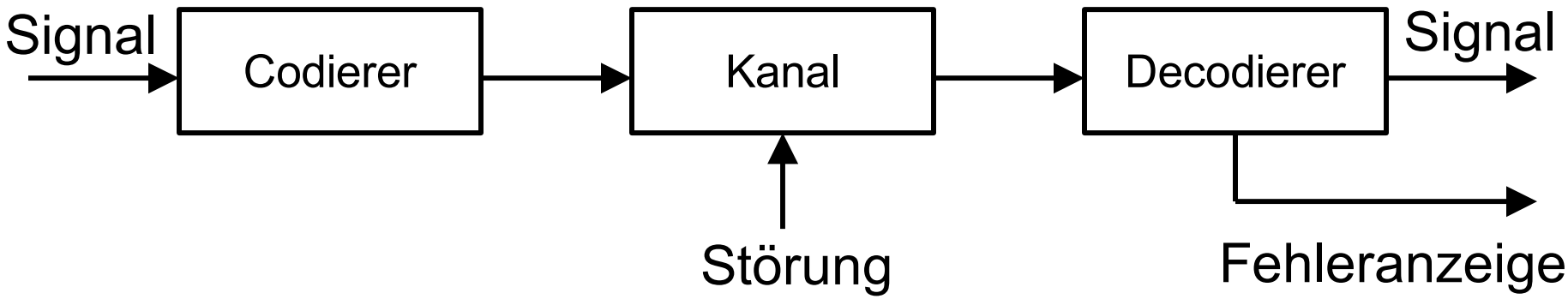
- **Daten bzw. Codewörter können bei der Übertragung und Speicherung verändert werden**
  - Z. B. Digitalfernsehen, Surfen mittels Modem
  - Speichern auf Festplatte, USB-Stick



- **Schutz vor Störungen erforderlich**

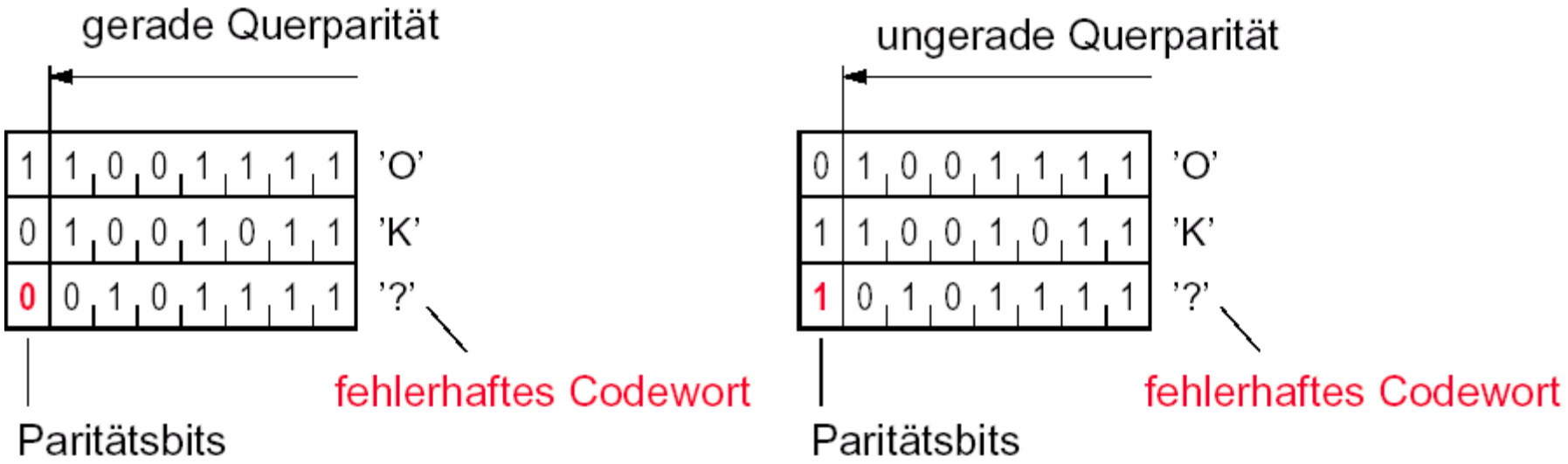
Codesicherung (2)

- Fehlererkennung und Fehlerkorrektur durch redundante Information
- Nutzinformation (*n* Bits) wird vor der Übertragung um Prüfinformation (*p* Bits) ergänzt
- Prüfinformation wird vom Empfänger ausgewertet
- Codewörter der Länge *n* werden auf Codewörter der Länge $m = n + p$ abgebildet



Fehlererkennende Codes (1)

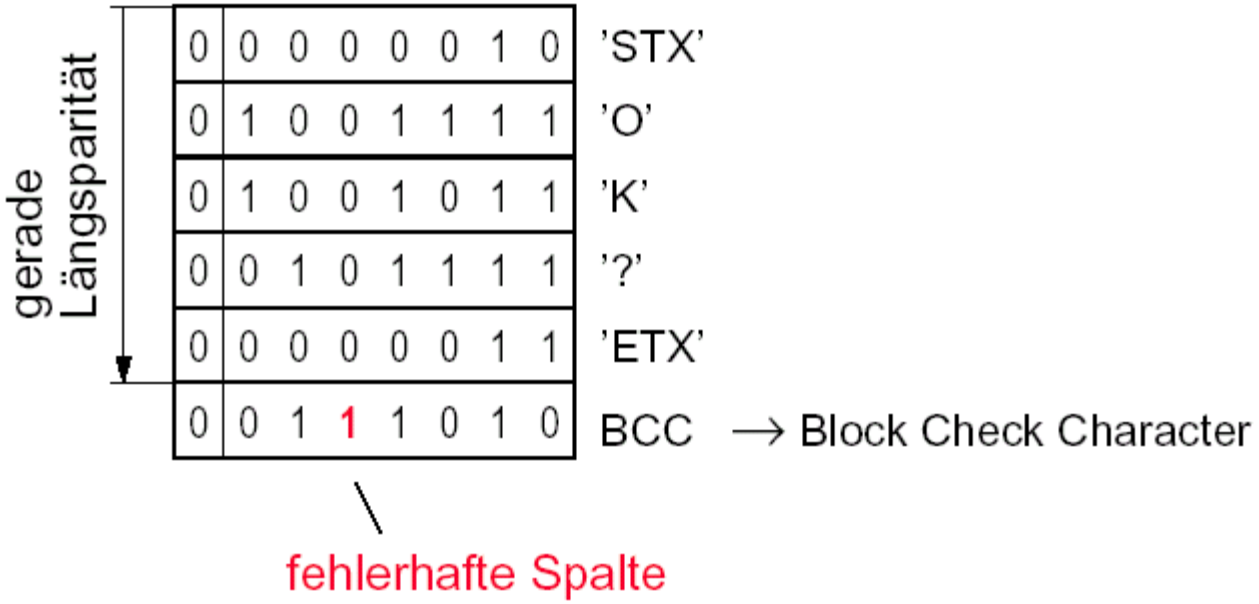
- **Quersicherung** **durch wortweise** **Paritätsbildung**
- **Gerade Parität** (*even parity*) **Ungerade Parität** (*odd parity*)



- Bei Fehlererkennung: Wiederholung der Übertragung!

Fehlererkennende Codes (2)

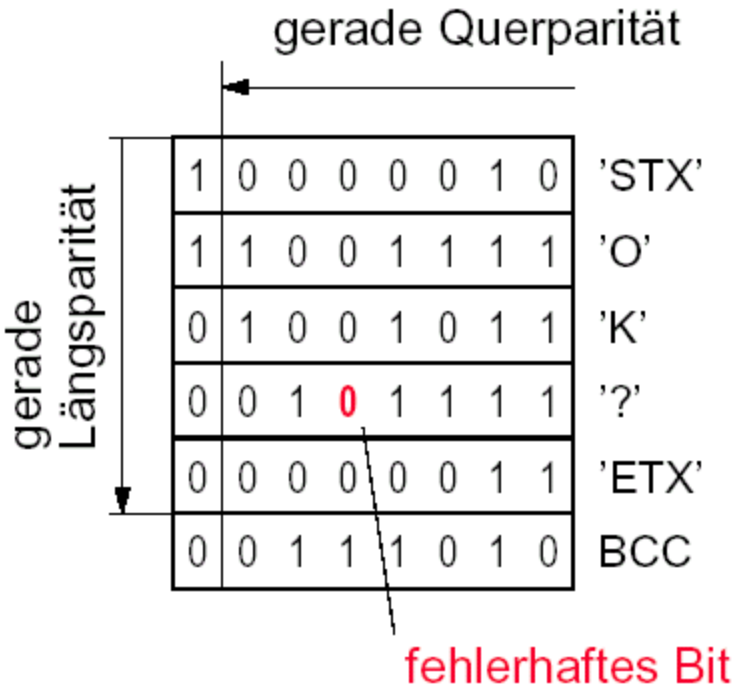
- **Längssicherung durch blockweise Paritätsbildung**



- **Bei Quer- und Längssicherung mit Paritätsbildung**
  - 1-Bit-Fehler und Mehr-Bit-Fehler in ungerader Anzahl erkennbar

Fehlerkorrigierende Codes

- **Beispiel:**
  Blockweise Kreuzsicherung
  (Rechtecksicherung)
- Kombination aus Quer- und Längssicherung
- 1-Bit-Fehler sind lokalisierbar und damit korrigierbar
- Mehr-Bit-Fehler sind in gewissem Umfang erkennbar
  - 4-Bit-Rechteckfehler beispielsweise NICHT erkennbar



Übung

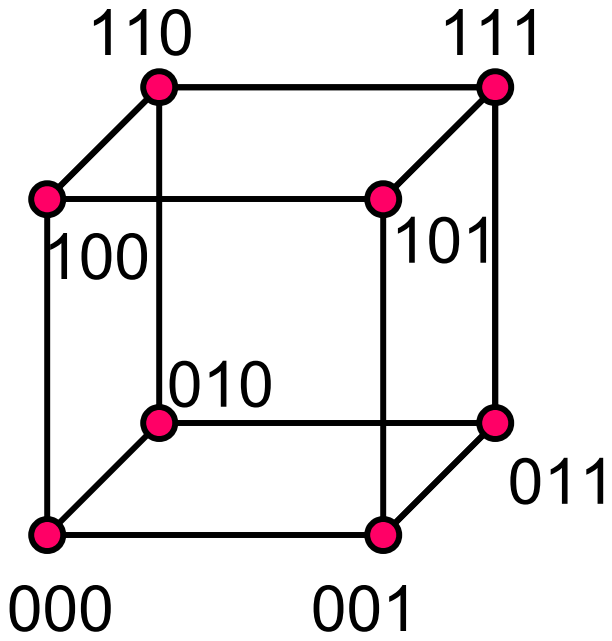
- Gegeben ist ein mit Kreuzsicherung und gerader Parität gesicherter Datenblock. Markieren Sie fehlerhafte Bits und interpretieren Sie den korrigierten Datenblock als ASCII Zeichenfolge.

1	1	0	0	1	0	0	1
1	1	1	0	1	1	1	0
0	1	1	0	0	1	1	0
0	1	1	0	1	1	1	1
1	0	1	0	0	0	0	1
1	0	0	1	1	1	1	1

Bit					00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	HEX
8					0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	
7					0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	
6					0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	
5					0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	
4	3	2	1																		
0	0	0	0	00			SP	0	@	P	`	p			NS	º	À	Đ	à	ð	0
0	0	0	1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0	0	1	0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0	0	1	1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0	1	0	0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0	1	0	1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0	1	1	0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0	1	1	1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1	0	0	0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1	0	0	1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1	0	1	0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1	0	1	1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1	1	0	0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1	1	0	1	13			-	=	M	]	m	}			SH	½	Í	Ý	í	ý	D
1	1	1	0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1	1	1	1	15			/	?	O	_	o	DE			¯	¿	Ï	ß	ï	ÿ	F
HEX					0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

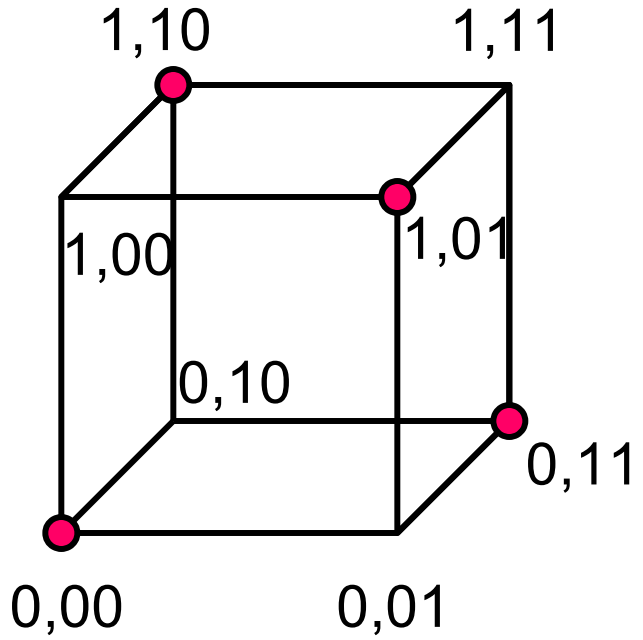
Fehlerkorrigierende Codes (2)

- **Hamming-Distanz**
  - Definiert als die minimale Anzahl unterschiedlicher Bits zweier benachbarter Codewörter
    - Mindestdistanz zweier Codewörter
- **Beispiel: 3-Bit-Code**
  - D. h. Codewörter haben die Länge 3 Bit
  - Mit Nutzinformation $n = 3$ Bits und Prüfinformation $p = 0$ Bits sind Codewörter als Menge der Eckpunkte eines Würfels darstellbar
  - Hamming-Distanz h=1
  - Erhöhung der Mindestdistanz ermöglicht durch Einführen von Redundanz Fehlererkennung oder sogar Fehlerkorrektur



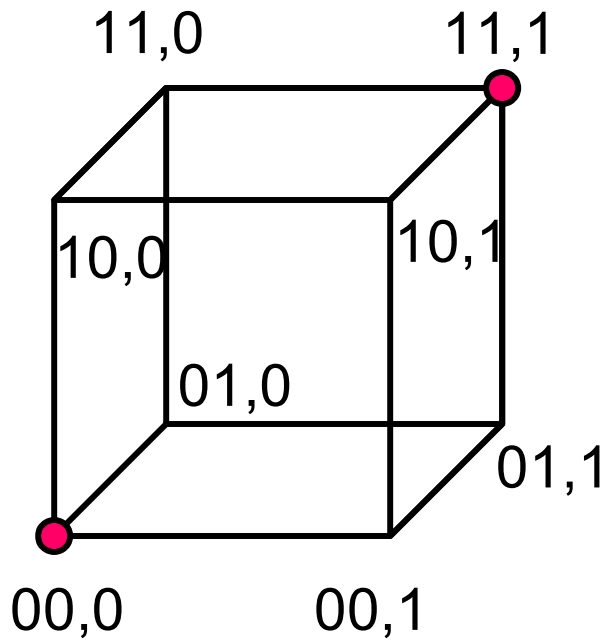
Fehlerkorrigierende Codes (3)

- **Hamming-Distanz h=2**
  - Nutzinformation *n* = *2* Bits und Prüfinformation *p* = 1 Bit
  - Zwischen je zwei Codewörtern liegt 1 ungültiges Codewort
  - Ermöglicht 1-Bit-Fehlererkennung
    - Gültige Codewörter unterscheiden sich in mindestens 2 Bits
      - Abweichung von einem Bit ('Bit-Kipper') automatisch Fehler
  - Darstellung entspricht gerader Parität



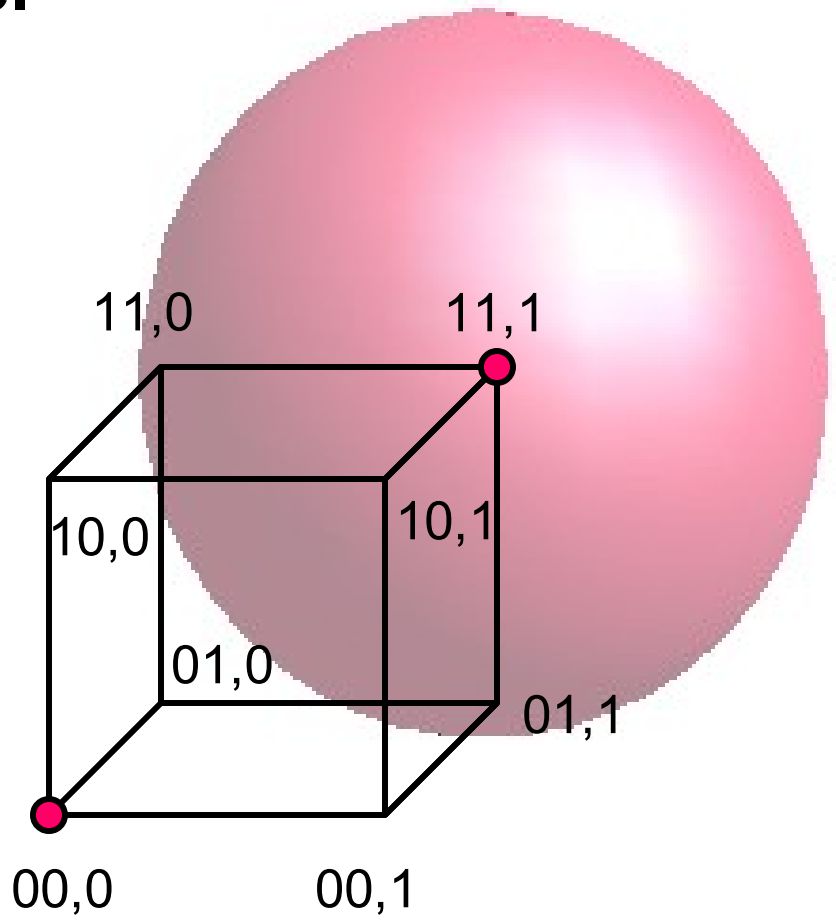
Fehlerkorrigierende Codes (4)

- **Hamming-Distanz h=3**
  - Nutzinformation *n* = 1 Bit und Prüfinformation *p* = 2 Bits
  - Zwischen je zwei Codewörtern liegen 2 ungültige Codewörter
  - Ermöglicht 2-Bit-Fehlererkennung
    - Gültige Codewörter unterscheiden sich in mindestens 3 Bits
      - Abweichungen bis max. 2 Bit automatisch Fehler
    - Ermöglicht 1-Bit-Fehlerkorrektur
      - 1-Bit-Fehler liegen innerhalb der 'Korrekturkugel' des jeweiligen Codeworts



Fehlerkorrigierende Codes (5)

- **Hamming-Distanz h=3**
- **Korrekturkugel**



Aussagenlogik

Aussagenlogik: Wahrheitswerte

- Viele Sätze natürlicher Sprache lassen eine Ja-Nein-Bewertung zu
- Diese Sätze stellen Aussagen dar, die entweder *wahr* oder *falsch* sind
  - Keine syntaktische, sondern eine *semantische* Eigenschaft von Sätzen
- Die beiden möglichen Werte, die einer Aussage zugeordnet werden können, heißen *Wahrheitswerte*

Aussagensätze (1)

- **Beispiele:**

  S1: 0 und 1 sind gleich

  - Diese Aussage ist falsch

  S2: Zwei Ziffern *x* und *y* sind gleich

  - Wahrheitswert abhängig von der Belegung von *x* und *y*

  S3: Zwei Zahlen *X* und *Y* sind gleich

  - Wahrheitswert von der Belegung der jeweiligen Ziffern der Zahlen *X* und *Y* abhängig

  S4: Zwei Zahlen *X* und *Y* sind dann und nur dann gleich, wenn ihre korrespondierenden Ziffern gleich sind

  - Diese Aussage ist wahr, ungeachtet des Wertes der einzelnen Ziffern

Aussagensätze (2)

- **Beispiele:**

  S1: 0 und 1 sind gleich

  S2: Zwei Ziffern *x* und *y* sind gleich

  S3: Zwei Zahlen *X* und *Y* sind gleich

  S4: Zwei Zahlen *X* und *Y* sind dann und nur dann gleich, wenn ihre korrespondierenden Ziffern gleich sind

  - Die Wahrheitswerte der Aussagen S1 und S4 sind unveränderlich und bekannt
  - Die Wahrheitswerte der Sätze S2 und S3 sind *variabel* und folglich unbekannt, solange keine *Variablenwerte* festgelegt worden sind
    - Die Sätze S2 und S3 sind erst dann Aussagen, nachdem die Werte ihrer Variablen festgelegt worden sind

Aussagensätze (3)

- **Analyse von S4:**

  Zwei Zahlen *X* und *Y* sind dann und nur dann gleich, wenn ihre korrespondierenden Ziffern gleich sind

  - S4 kann in zwei einfachere Aussagen, die durch *dann und nur dann* verbunden sind, zerlegt werden
  - Die erste Aussage ist identisch mit S3: Zwei Zahlen *X* und *Y* sind gleich
  - Die zweite Aussage lautet:
  - S5: Die Ziffern in allen Ziffernpaaren gleichen Gewichts sind gleich

Verknüpfung von Aussagen (1)

- S5: Die Ziffern in allen Ziffernpaaren gleichen Gewichts sind gleich
  - Unter der Annahme, dass *X* und *Y* zwei Zahlen zu jeweils *n* Ziffern sind, können wir S5 in weitere *n* Aussagen zerlegen, die durch *und* verknüpft sind
- S5 lautet dann:
  - Die zwei Ziffern in Position 0 sind gleich *und* die zwei Ziffern in Position 1 sind gleich *und*...*und* die zwei Ziffern in Position *n*-1 sind gleich
  - Die gefundenen Aussagen können nicht weiter in einfachere Aussagen zerlegt werden
  - Sie werden deshalb als *atomische* Aussagen bezeichnet

Verknüpfung von Aussagen (2)

- Wir schließen:
  - Aussagen können zur Bildung komplexerer Aussagen verknüpft werden
  - Komplexe Aussagen können in eine Anzahl *atomischer* Aussagen zerlegt werden
  - Aussagen sind entweder wahr oder falsch
- *Boolsche Algebra*: Formalismus zur zweifelsfreien Feststellung der Wahrheit oder Falschheit von Aussagen

Boolsche Algebra (1)

- Entwickelt vom Engländer George Boole (1815-1864)
  - *"An investigation into the laws of thought"*
- **Ziel**: Formale Begründung der Logik
- Aussagenwerte sollten berechenbar werden wie Zahlenwerte z. B. der Addition oder der Multiplikation
  - Mittels der Boolschen Algebra lassen sich unter Zuhilfenahme logischer Verknüpfungen komplexe Aussagen aus atomischen Aussagen bilden

Boolsche Algebra (2)

- Die Wahrheit oder Falschheit zusammengesetzter Aussagen kann aus dem *Wahrheitswert* ihrer atomischen Aussagen und den *verknüpfenden Operationen* abgeleitet werden
- Wahrheitswerte sollen hier durch die Dualziffern 0 und 1 ausgedrückt werden
  - 1: wahr
  - 0: falsch
- Logische Operationen, wie z. B. das "Und", sind in der Umgangssprache zu ungenau definiert
  - "Kompensatorisches Und"
- Das Ergebnis logischer Operationen wird daher mittels *Wahrheitstabellen* exakt beschrieben

Logische Operationen

Negation	$\overline{a}$ bzw. $\neg a$	nicht a
Konjunktion	$a \wedge b$	a und b
Disjunktion	$a \vee b$	a oder b
Antivalenz	$a \oplus b$	entweder a oder b
Äquivalenz	$a \Leftrightarrow b$	a genau dann, wenn b
Implikation	$a \Rightarrow b$	wenn a, dann b
Peirce-Funktion (NOR)	$a * b$	weder a noch b
Sheffer-Strich (NAND)	$a \mid b$	a und b, nicht beide

- In einer Implikation wird *a* die *Hypothese* und *b* die *Folgerung* genannt

Wahrheitstabellen

- Zur Erstellung werden alle Kombinationen der Wahrheitswerte der atomischen Aussagen gebildet
- Jeder Kombination wird der Wahrheitswert der logischen Operation zugeordnet

a	b	$a \wedge b$	$a \vee b$	$a \oplus b$	$a \Leftrightarrow b$	$a \Rightarrow b$	a * b	a | b
0	0	0	0	0	1	1	1	1
0	1	0	1	1	0	1	0	1
1	0	0	1	1	0	0	0	1
1	1	1	1	0	1	1	0	0

- Implikation: Eine falsche Hypothese hat keine Bedeutung für die korrekte Folgerung
  - Darum wird in diesem Fall die Implikation als wahr definiert

Ersetzung komplexer Operatoren

- Komplexe Operatoren lassen sich auf die *Konjunktion*, *Disjunktion* und *Negation* zurückführen

a	b	$\overline{a}$	$\overline{b}$	$a \Rightarrow b$	$\overline{a} \vee b$	a * b	$\overline{a \vee b}$	a | b	$\overline{a \wedge b}$
0	0	1	1	1	1	1	1	1	1
0	1	1	0	1	1	0	0	1	1
1	0	0	1	0	0	0	0	1	1
1	1	0	0	1	1	0	0	0	0

- Ausdrücke, in denen nur die Operationen *Konjunktion*, *Disjunktion* und *Negation* vorkommen, werden als *Boolsche Ausdrücke* bezeichnet

Übung

- Zeigen Sie, dass zwei Ausdrücke dann und nur dann äquivalent sind, wenn sie sich gegenseitig implizieren
- **D. h.** $(a \Leftrightarrow b) \equiv (a \Rightarrow b) \wedge (b \Rightarrow a)$

a	b	a ∧ b	a ∨ b	a ⊕ b	a ⇔ b	a ⇒ b	a * b	a | b
0	0	0	0	0	1	1	1	1
0	1	0	1	1	0	1	0	1
1	0	0	1	1	0	0	0	1
1	1	1	1	0	1	1	0	0

Sätze zum Umformen in Boolsche Ausdrücke

$$a \Leftrightarrow b \equiv (\bar{a} \wedge \bar{b}) \vee (a \wedge b) \equiv (\bar{a} \vee b) \wedge (a \vee \bar{b})$$

$$a \oplus b \equiv (\bar{a} \wedge b) \vee (a \wedge \bar{b}) \equiv (\bar{a} \vee \bar{b}) \wedge (a \vee b)$$

$$a \Rightarrow b \equiv \overline{\bar{a} \vee b}$$

$$a * b \equiv \overline{a \vee b}$$

$$a \mid b \equiv \overline{a \wedge b}$$

Logische Ausdrücke

- Definition: Ein logischer Ausdruck entsteht durch endlich oft durchgeführte Verknüpfungen mit Konstanten, Variablen und Ausdrücken
- Die Reihenfolge in der die Verknüpfungen anzuwenden sind, wird durch die Art der jeweiligen Verknüpfung, d. h. des Operators, oder durch Klammern festgelegt
- Das Definieren von Operatorrangfolgen ermöglicht es, viele Klammern in Ausdrücken einzusparen
- Üblich ist folgende Rangfolge: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$
  - Von links nach rechts, d. h. die Negation $\neg$ bindet am stärksten

Auswertung von Ausdrücken

- 1. Schritt: Niederschreiben aller 0/1-Kombinationen für die Variablen des Ausdrucks

- 2. Schritt: Für jede Kombination wird der Wert entsprechend der Klammerstruktur und dem Rang einer Verknüpfung schrittweise mit Hilfe der Definitionen der einzelnen Verknüpfungen ermittelt

Beispiel

- Zu Prüfen ist die Äquivalenz der beiden Booleschen Ausdrücke $(a \wedge b) \vee c$ und $(a \vee c) \wedge (b \vee c)$

a	b	c	a ∧ b	(a ∧ b) ∨ c	(a ∨ c)	(b ∨ c)	(a ∨ c) ∧ (b ∨ c)
0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	1
0	1	0	0	0	0	1	0
0	1	1	0	1	1	1	1
1	0	0	0	0	1	0	0
1	0	1	0	1	1	1	1
1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1

Axiome

- Die folgenden 10 Gleichungen definieren die Boolsche Algebra

1. $a \wedge b \equiv b \wedge a$
2. $a \vee b \equiv b \vee a$
3. $(a \wedge b) \wedge c \equiv a \wedge (b \wedge c)$
4. $(a \vee b) \vee c \equiv a \vee (b \vee c)$
5. $(a \vee b) \wedge c \equiv (a \wedge c) \vee (b \wedge c)$
6. $(a \wedge b) \vee c \equiv (a \vee c) \wedge (b \vee c)$
7. $a \wedge 1 \equiv a$
8. $a \vee 0 \equiv a$
9. $a \wedge \bar{a} \equiv 0$
10. $a \vee \bar{a} \equiv 1$

(1), (2): Kommutativgesetze

(3), (4): Assoziativgesetze
(Besagen, dass die Reihenfolge der Auswertung bei mehrgliedrigen Ausdrücken unwichtig ist)

(5), (6): Distributivgesetze
(Gemischte Klammerausdrücke können „ausmultipliziert“ und „ausaddiert“ werden)

(7), (8): Neutralitätsgesetze

(9), (10): Komplementärgesetze

Sätze (1)

- Die folgenden Gleichungen zur Umformung Boolscher Ausdrücke sind aus den Axiomen ableitbar oder können mittels Wahrheitstabellen gezeigt werden
- Vereinfachung Boolscher Ausdrücke:

11. $a \wedge a \equiv a$
12. $a \vee a \equiv a$
13. $a \wedge 0 \equiv 0$
14. $a \vee 1 \equiv 1$
15. $a \vee (a \wedge b) \equiv a$
16. $a \wedge (a \vee b) \equiv a$
17. $a \vee (\bar{a} \wedge b) \equiv a \vee b$
18. $a \wedge (\bar{a} \vee b) \equiv a \wedge b$

(11), (12): Idempotenzgesetze

(13), (14): Extremalgesetze

(15), (16): Absorptionsgesetze

Beispiel

- Wir zeigen die Gültigkeit von Satz 15: $a \vee (a \wedge b) \equiv a$

a	b	a ∧ b	a ∨ (a ∧ b)
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

Übung

- Zeigen Sie die Gültigkeit von Satz 18: $a \wedge (\bar{a} \vee b) \equiv a \wedge b$

a	b	a ∧ b	a ∨ b	a ⊕ b	a ⇔ b	a ⇒ b	a * b	a | b
0	0	0	0	0	1	1	1	1
0	1	0	1	1	0	1	0	1
1	0	0	1	1	0	0	0	1
1	1	1	1	0	1	1	0	0

Sätze (2)

- Sätze für die Negation eines Ausdrucks:

  19. $\overline{\overline{a}} \equiv a$

  20. $\overline{a} \wedge \overline{b} \equiv \overline{(a \vee b)}$

  21. $\overline{a} \vee \overline{b} \equiv \overline{(a \wedge b)}$

- Die Sätze Nr. 20 und 21 werden als *de Morgansche Gesetze* bezeichnet
- Verallgemeinerung von 20 und 21 auf *n* Variablen:

  22. $\overline{x_1 \wedge x_2 \wedge \ldots \wedge x_n} \equiv \overline{x_1} \vee \overline{x_2} \vee \ldots \vee \overline{x_n}$

  23. $\overline{x_1 \vee x_2 \vee \ldots \vee x_n} \equiv \overline{x_1} \wedge \overline{x_2} \wedge \ldots \wedge \overline{x_n}$

Beispiel: Vereinfachung eines Ausdrucks

$$x_1 \vee (\overline{x_1} \wedge x_2) \vee (\overline{x_1} \wedge \overline{x_2} \wedge x_3) \equiv \text{(mit 6.)}$$

$$(x_1 \vee \overline{x_1}) \wedge (x_1 \vee x_2) \vee (\overline{x_1} \wedge \overline{x_2} \wedge x_3) \equiv \text{(mit 10.)}$$

$$1 \wedge (x_1 \vee x_2) \vee (\overline{x_1} \wedge \overline{x_2} \wedge x_3) \equiv \text{(mit 7.)}$$

$$(x_1 \vee x_2) \vee (\overline{x_1} \wedge \overline{x_2} \wedge x_3) \equiv \text{(mit 20.)}$$

$$(x_1 \vee x_2) \vee (\overline{(x_1 \vee x_2)} \wedge x_3) \equiv \text{(mit 6.)}$$

$$(x_1 \vee x_2) \vee \overline{(x_1 \vee x_2)} \wedge ((x_1 \vee x_2) \vee x_3) \equiv \text{(mit 10. und 4.)}$$

$$1 \wedge (x_1 \vee x_2 \vee x_3) \equiv \text{(mit 7.)}$$

$$x_1 \vee x_2 \vee x_3$$

4. $(a \vee b) \vee c \equiv a \vee (b \vee c)$
6. $(a \wedge b) \vee c \equiv (a \vee c) \wedge (b \vee c)$
7. $a \wedge 1 \equiv a$
10. $a \vee \overline{a} \equiv 1$
20. $\overline{a} \wedge \overline{b} \equiv \overline{(a \vee b)}$

Übung

- Vereinfachen Sie den folgenden Ausdruck zu 1:

$$x_1 \vee (\overline{x_1} \wedge x_2) \vee (\overline{x_1} \wedge \overline{x_2})$$

4. $(a \vee b) \vee c \equiv a \vee (b \vee c)$
6. $(a \wedge b) \vee c \equiv (a \vee c) \wedge (b \vee c)$
7. $a \wedge 1 \equiv a$
10. $a \vee \bar{a} \equiv 1$
17. $a \vee (\bar{a} \wedge b) \equiv a \vee b$
20. $\bar{a} \wedge \bar{b} \equiv \overline{(a \vee b)}$

Übung

- Zeigen Sie die Gültigkeit von Satz 18 erneut, diesmal ohne Wahrheitstabellen:

$$a \wedge (\bar{a} \vee b) \equiv a \wedge b$$

Tautologie und Kontradiktion

- **Definition**: Ausdrücke, die für alle Kombinationen der Wahrheitswerte ihrer Variablen den Wahrheitswert 1 annehmen, heißen allgemein gültig oder **Tautologie**

- **Definition**: Ausdrücke, die für alle Kombinationen der Wahrheitswerte ihrer Variablen den Wahrheitswert 0 annehmen, heißen **Kontradiktion**

Übung

- Überprüfen Sie mittels Wahrheitstabelle, dass der Ausdruck

  $x_1 \vee (\overline{x_1} \wedge x_2) \vee (\overline{x_1} \wedge \overline{x_2})$

  aus der vorletzten Übung eine **Tautologie** ist

Dualität (1)

- **Definition**: Zwei Verknüpfungen heißen **dual**, wenn ihre Wahrheitstabellen derart zusammenhängen, dass die Wahrheitstabelle der einen Operation entsteht, wenn in der Wahrheitstabelle der dualen Operation 0 und 1 vertauscht werden
- Beispiel: Konjunktion und Disjunktion sind dual zueinander

a	b	$a \wedge b$
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$a \vee b$
1	1	1
1	0	1
0	1	1
0	0	0

Dualität (2)

- **Definition**: Aus einem Ausdruck entsteht der duale Ausdruck, wenn die Verknüpfungszeichen durch ihre dualen ersetzt sowie 0 und 1 vertauscht werden
- Duale Operationen sind:
  - $\wedge$ und $\vee$
  - $\neg$ und $\neg$
  - * und | (Peirce-Funktion (NOR) und Sheffer-Strich (NAND))
  - $\oplus$ und $\Leftrightarrow$ (Antivalenz und Äquivalenz)
- **Beispiele:**

$x_1 \vee x_2 \vee \ldots \vee x_n$ *ist dual zu* $x_1 \wedge x_2 \wedge \ldots \wedge x_n$

$(a \vee \overline{b} \wedge c) \wedge \overline{(c \wedge 1)}$ *ist dual zu* $(a \wedge \overline{b} \vee c) \vee \overline{(c \vee 0)}$

Negation von Ausdrücken

- **Satz**: Ein Ausdruck wird negiert, indem man zum dualen Ausdruck übergeht und jede Variable einzeln negiert.
- Beispiele:

$$\overline{x_1 \vee x_2 \vee \ldots \vee x_n} \equiv \overline{x_1} \wedge \overline{x_2} \wedge \ldots \wedge \overline{x_n}$$

$$\overline{(a \vee \overline{b} \wedge c) \wedge \overline{(c \wedge 1)}} \equiv (\overline{a} \wedge b \vee \overline{c}) \vee \overline{(\overline{c} \vee 0)} \equiv (\overline{a} \wedge b \vee \overline{c}) \vee (c \wedge 1)$$

Übung

- Negieren Sie den folgenden Ausdruck:

$$\bar{a} \vee b \wedge \overline{c \wedge 1}$$

Halbaddierer

- Beispiel für die Umsetzung von Rechenoperationen mittels logischer Ausdrücke
- Halbaddierfunktion
- Wir betrachten die Addition von zwei Zahlen

$$X = x_{n-1} \ldots x_1 x_0$$
$$Y = y_{n-1} \ldots y_1 y_0$$

- Ein Übertrag $u_{i+1} = 1$ entsteht dann und nur dann, wenn $x_i=1$ *und* $y_i=1$: $u_{i+1} = x_i \wedge y_i$
- Eine Summenziffer $s_i = 1$ entsteht dann und nur dann, wenn *entweder* $x_i=1$ *oder* $y_i=1$: $s_i = x_i \oplus y_i$

Volladdierer

- Wir betrachten wieder die Addition von zwei Zahlen

  $X = x_{n-1} \ldots x_1 x_0$
  $Y = y_{n-1} \ldots y_1 y_0$

- Ein Übertrag $u_{i+1} = 1$ entsteht dann und nur dann, wenn $x_i = 1$ *und* $y_i = 1$ *oder* $x_i = 1$ *oder* $y_i = 1$ *und* $u_i = 1$:

  $u_{i+1} = (x_i \wedge y_i) \vee (x_i \vee y_i) \wedge u_i$

- Eine Summenziffer $s_i = 1$ entsteht dann und nur dann, wenn *entweder* $x_i = 1$ *oder* $y_i = 1$ *oder* $u_i = 1$:

  $s_i = x_i \oplus y_i \oplus u_i$

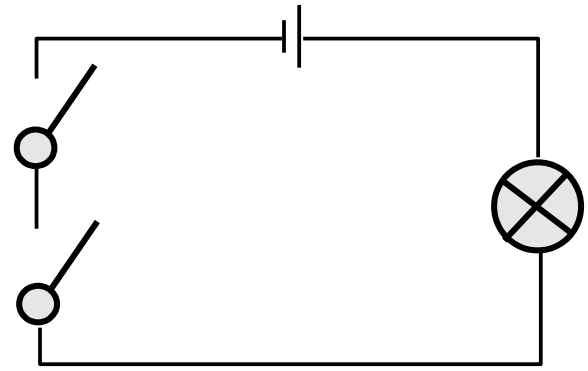
Schalternetze

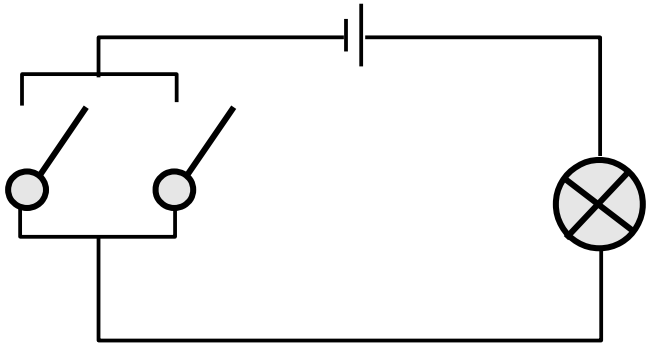
Schalternetze

- Boolesche Funktionen wie der Addierer lassen sich mittels Schalternetzen realisieren
- Hierfür werden atomische Aussagen durch "wahr" und "falsch" Signale und ihre logischen Verknüpfungen durch Schalter repräsentiert
- Eine formalisierte Beschreibung von Schalternetzen führt zur **Schaltalgebra** von Claude E. Shannon (1938)
  - Es lässt sich zeigen, dass die Shannonsche Schaltalgebra und die Boolsche Algebra äquivalent sind

Serien-parallele Schaltungen

- Wird eine Lampe nicht durch einen einzelnen Schalter, sondern durch zwei Schalter eingeschaltet, so ergeben sich zwei mögliche Kombinationen:
- Die Serienschaltung und die Parallelschaltung
- Die **Serienschaltung** steht für das logische **und**
  - **Und-Schaltung**
- Die **Parallelschaltung** steht für das logische **oder**
  - **Oder-Schaltung**



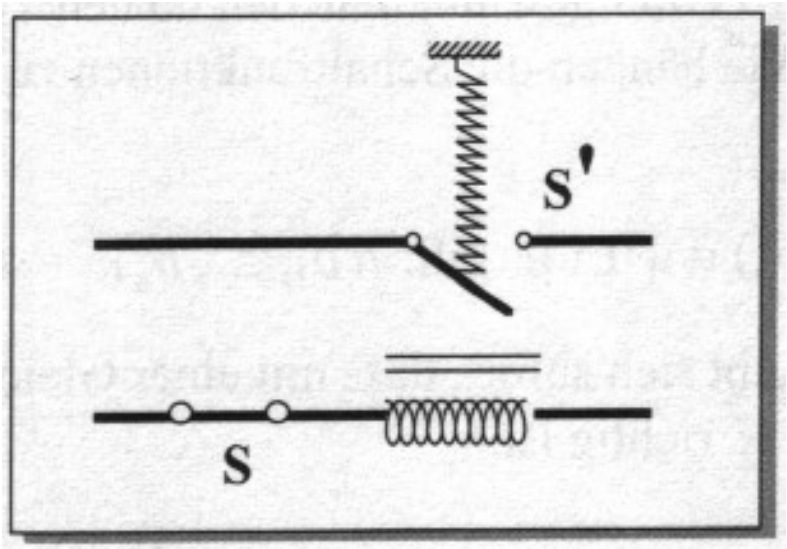


Negation

- Negation nicht durch Serien-Parallel-Schaltung realisierbar
- Wechselschaltung erforderlich
- Definition: Ist *S* ein Schaltglied, so sei *S*' dasjenige Schaltglied, welches genau dann offen ist, wenn *S* geschlossen ist
- *S*' heißt die Negation von *S*

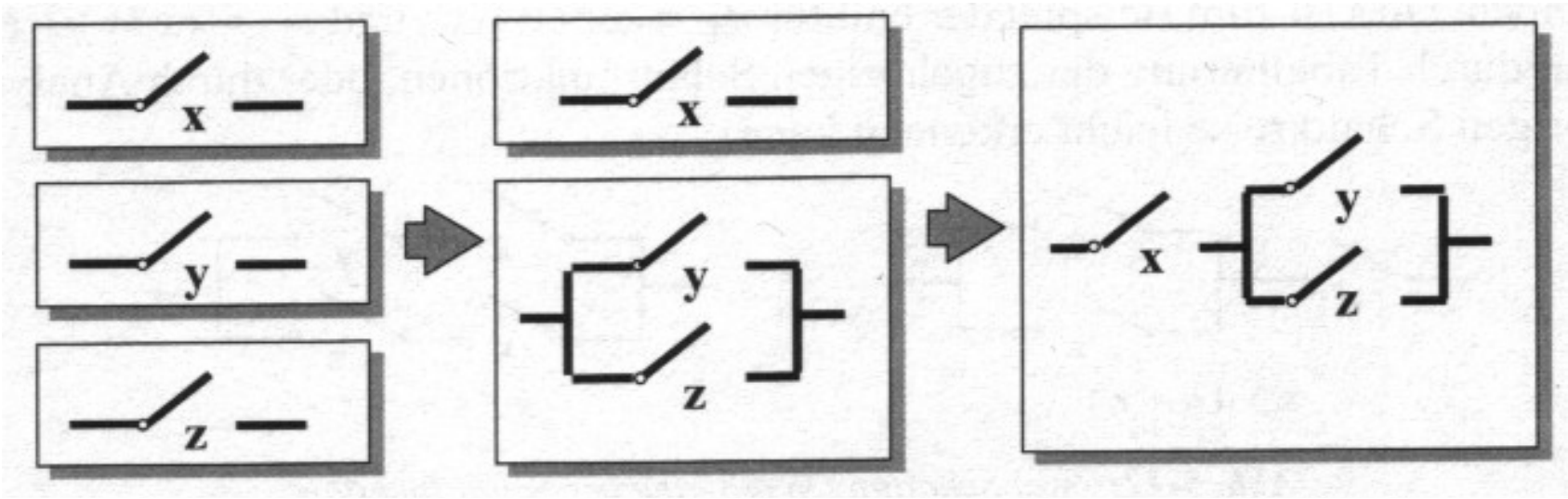
Technische Umsetzung

- Negation in der Frühzeit der Computertechnik durch Relais realisiert
  - Fließt Strom durch die von *S* eingeschaltete Spule, so wird durch die entstehende Magnetkraft der Schalter *S*' gegen die Federkraft geöffnet
- Später: Transistoren



Beispiel

- Realisierung der Booleschen Funktion $x \wedge (y \vee z)$

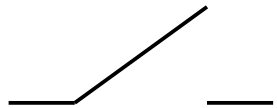


x, y, z $\qquad$ $x, (y \vee z)$ $\qquad$ $x \wedge (y \vee z)$

Übung

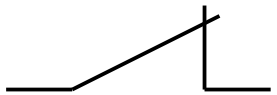
- Stellen Sie den Halbaddierer als Schalternetz dar

- Mit:



a

„Schließer“



$\overline{\mathbf{a}}$

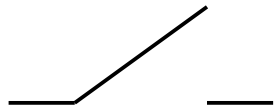
„Öffner“

- Summe: $s_i = x_i \oplus y_i$,
- Übertrag: $u_{i+1} = x_i \wedge y_i$

Übung

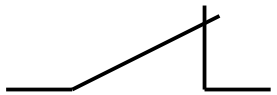
- Stellen Sie den Volladdierer als Schalternetz dar

- Mit:



a

„Schließer“



$\overline{\mathbf{a}}$

„Öffner“

- Summe: $s_i = x_i \oplus y_i \oplus u_i$,
- Übertrag: $u_{i+1} = (x_i \wedge y_i) \vee (x_i \vee y_i) \wedge u_i$

Von-Neumann-Rechner und Maschinensprache

Beispiel

- Berechnung eines Polynoms durch einen **Menschen** mit Hilfe eines Rechners

$$\boldsymbol{p} = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 =$$
$$\boldsymbol{((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0}$$

mit: $a_3 = 9$, $a_2 = 8$, $a_1 = 7$, $a_0 = 6$ und $x = 3$.

- Unser Rechner besitze ein sog. **Akkumulatorregister**, das auch als Anzeige beim Eingeben und Ausgeben von Zahlen dient
  - Vergleichbar mit der Eingabezeile eines Taschenrechners und der dazu gehörenden Speicherzelle

Operationen

- Unser Rechner kann u. a. folgende Operationen ausführen:

Taste	Variable	Wirkung
↓	M	Lädt Akkumulator mit dem Wert von M
↑	M	Speichert den Inhalt des Akkumulators in M
+	M	Addiert den in M gespeicherten Wert auf den im Akkumulator gespeicherten Wert
*	M	Multipliziert den in M gespeicherten Wert mit dem im Akkumulator gespeicherten Wert
etc.		

Polynomauswertung (1)

- Dem **Bediener** liegt die Lösung der Aufgabe in Form eines **Programms** vor
  - Dies ist ein Zettel mit Anweisungen und notierten Zahlenwerten
- Das Programm besteht aus so genannten **„Befehlen“**

Polynomauswertung (2)

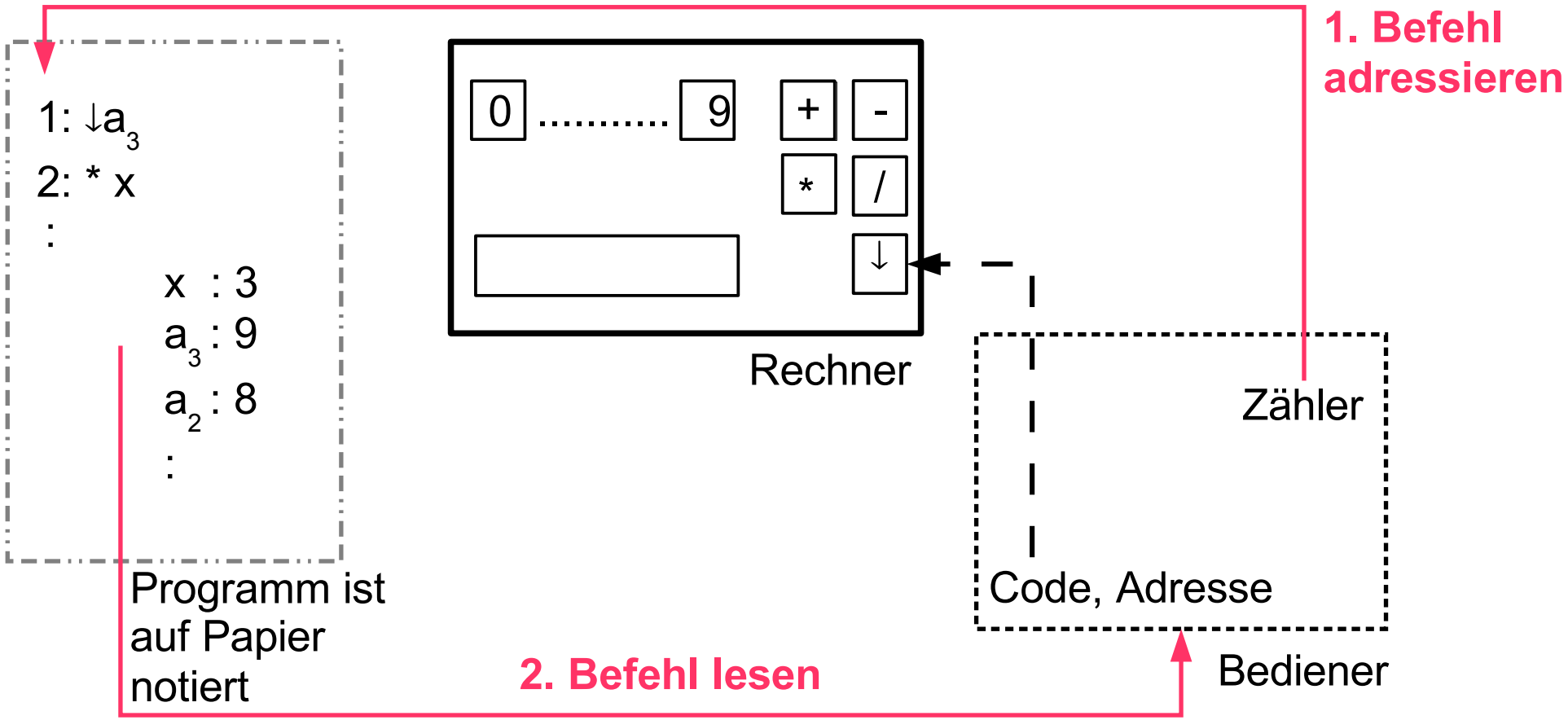
$p = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 = ((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0$

mit: $a_3 = 9$, $a_2 = 8$, $a_1 = 7$, $a_0 = 6$ und $x = 3$

Nr. des Befehls	Art der Operation	Name des beteiligten Operanden	Wert
1	$\downarrow a_3$	a_3	9
2	*x	x	3
3	$+a_2$	a_2	8
4	*x	x	3
5	$+a_1$	a_1	7
6	*x	x	3
7	$+a_0$	a_0	6

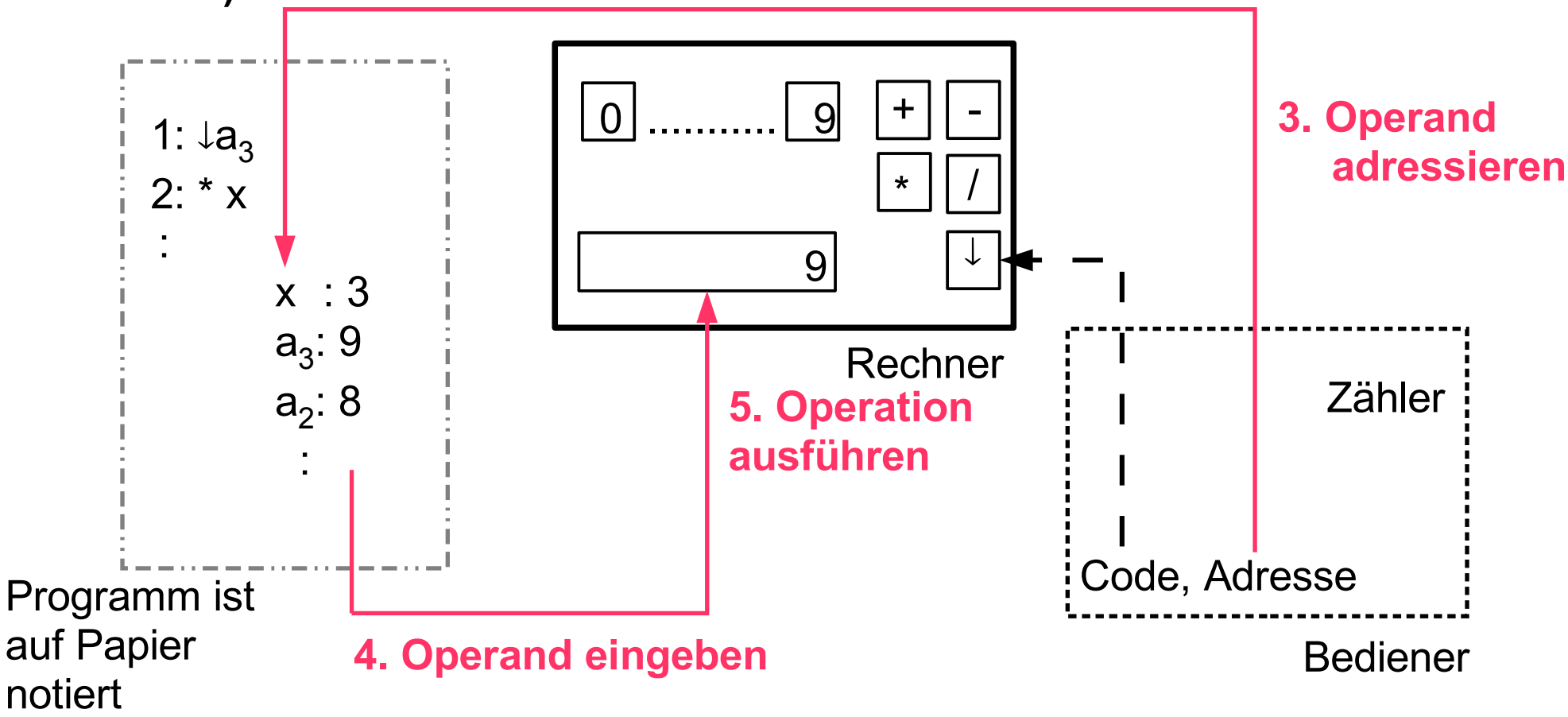
Tätigkeit des Bedieners (1)

- **Der Bediener liest den ersten Befehl vom Papier ab und interpretiert ihn**
  - Der Befehl ist aufgeteilt in den **Operationscode** (Name der Operation) und in die **Operandenadresse** (Name des Operanden)



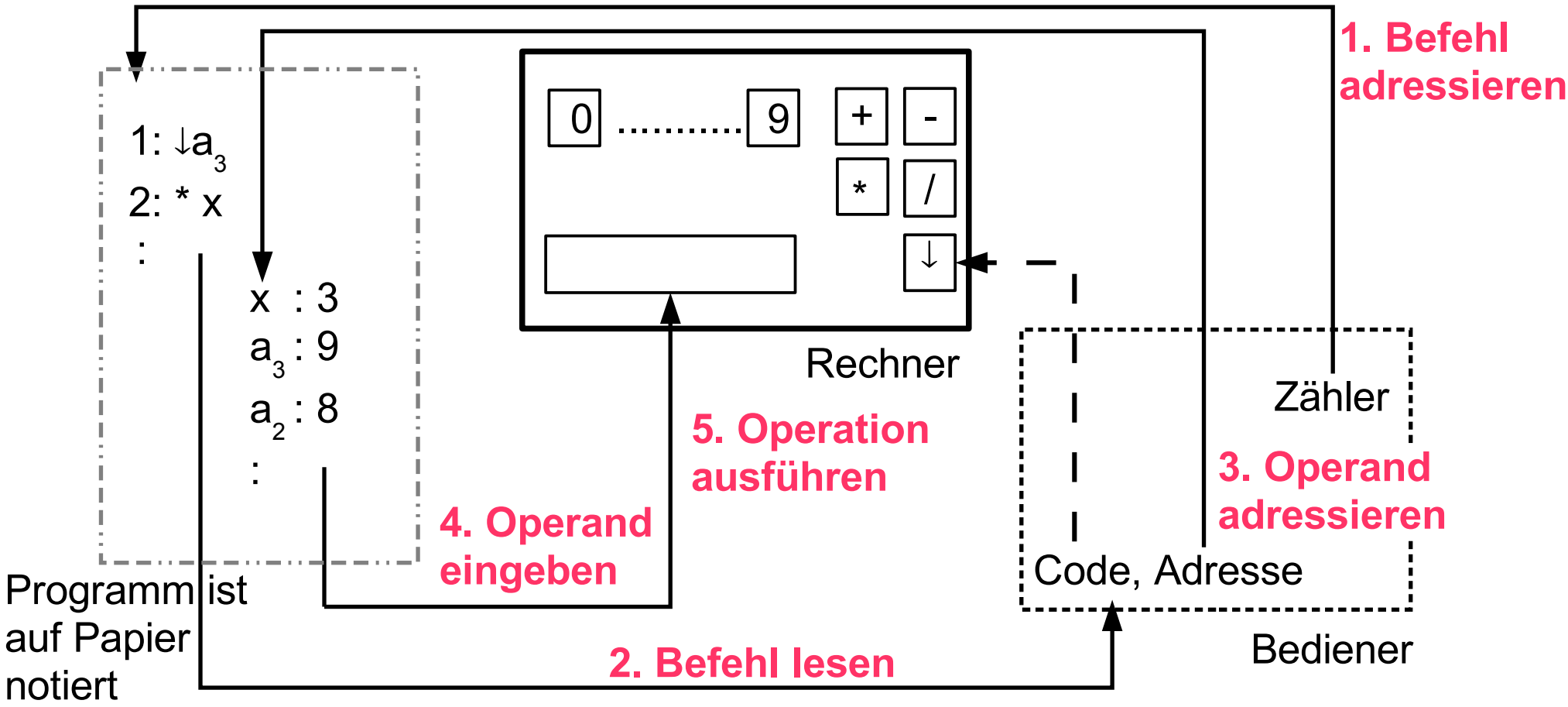
Tätigkeit des Bedieners (2)

- **Der Bediener sucht den zur Operandenadresse gehörenden Wert des Operanden**
- **Wert wird in den Rechner eingegeben und Befehl ausgeführt (Enter-Taste)**



Tätigkeit des Bedieners (3)

- Anschließend wird der nächste Befehl bearbeitet
- Nach Bearbeitung eines jeden Befehls wird der sog. **Befehlszähler** um eins hochgezählt (*inkrementiert*)



Von-Neumann-Rechner (1)

- Beim Von-Neumann-Rechner wird die Arbeit des Bedieners vom Rechenautomat übernommen
- An die Stelle des Bedieners tritt ein elektronisches Schaltwerk: das **Steuerwerk**
- Die Speicherung von *Programmbefehlen* und *Daten* erfolgt nicht länger auf Papier, sondern mit Hilfe eines elektronischen Speicherwerks, des so genannten **Hauptspeichers**
- Programmbefehle und Daten müssen in den Rechenautomat überführt, d. h. er muss **programmiert** werden

Von-Neumann-Rechner (2)

- Die Befehle und Daten des Programms werden im Rechenautomat als **Binärwörter** gespeichert
- Hierfür ist eine Codierung der Daten und Operationen der Befehle erforderlich
- Die Codierung der Daten erfolgt mit Hilfe von Dualzahlen
- Die Codierung der Operationen mit Hilfe einer so genannten **Codetabelle**

HALT	0001
↑	0101
↓	1000
+	1100
*	1110
⋮	

Beispiel

- Gegeben sei ein Rechner, der über einen Speicher der Wortlänge 16 Bits verfügt
- Sollen die Befehle jeweils in ein Speicherwort passen, so müssen die 16 Bits auf den Operationscode und die Operandenadresse aufgeteilt werden
  - Dies wird als "**Einadressrechner**" bezeichnet
- In unserem Beispiel sei folgende Aufteilung gewählt:
  - 4 Bit für den Operationscode
    - Somit sind $2^4 = 16$ Operationen codierbar
  - 12 Bit für die Operandenadresse
    - Somit $2^{12} = 4096$ verschiedene Operanden adressierbar

Befehlscodes und Codetabelle

- Die möglichen Operationen unseres Rechners seien u. a. durch die folgenden Binärwörter codiert:

Befehlscode	Speicherzelle	Wirkung
0001		Hält den Rechner am Ende des Programms an
0101	M	Speichert den Inhalt des Akkumulators in M
1000	M	Lädt den Akkumulator mit dem Inhalt von M
1100	M	Addiert den Inhalt von M zu dem Inhalt des Akkumulators
1110	M	Multipliziert den Inhalt von M mit dem Inhalt des Akkumulators und speichert das Resultat im Akkumulator
⋮	⋮	⋮

Programm zur Polynomauswertung (1)

$p = ((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0$

mit: $a_3 = 9$, $a_2 = 8$, $a_1 = 7$, $a_0 = 6$ und $x = 3$

Adresse (1.Teil)	Befehle und Operanden (1.Teil)	Adresse (2.Teil)	Befehle und Operanden (2.Teil)
00000	1000000000001001	01000	0001000000000000
00001	1110000000001101	01001	0000000000001001
00010	1100000000001010	01010	0000000000001000
00011	1110000000001101	01011	0000000000000111
00100	1100000000001011	01100	0000000000000110
00101	1110000000001101	01101	0000000000000011
00110	1100000000001100	01110	0000000000000000
00111	0101000000001110		

Befehls-code	Speicher-zelle	Wirkung
0001		HALT
0101	M	M := A
1000	M	A := M
1100	M	A := A+M
1110	M	A := A*M
⋮		

Programm zur Polynomauswertung (2)

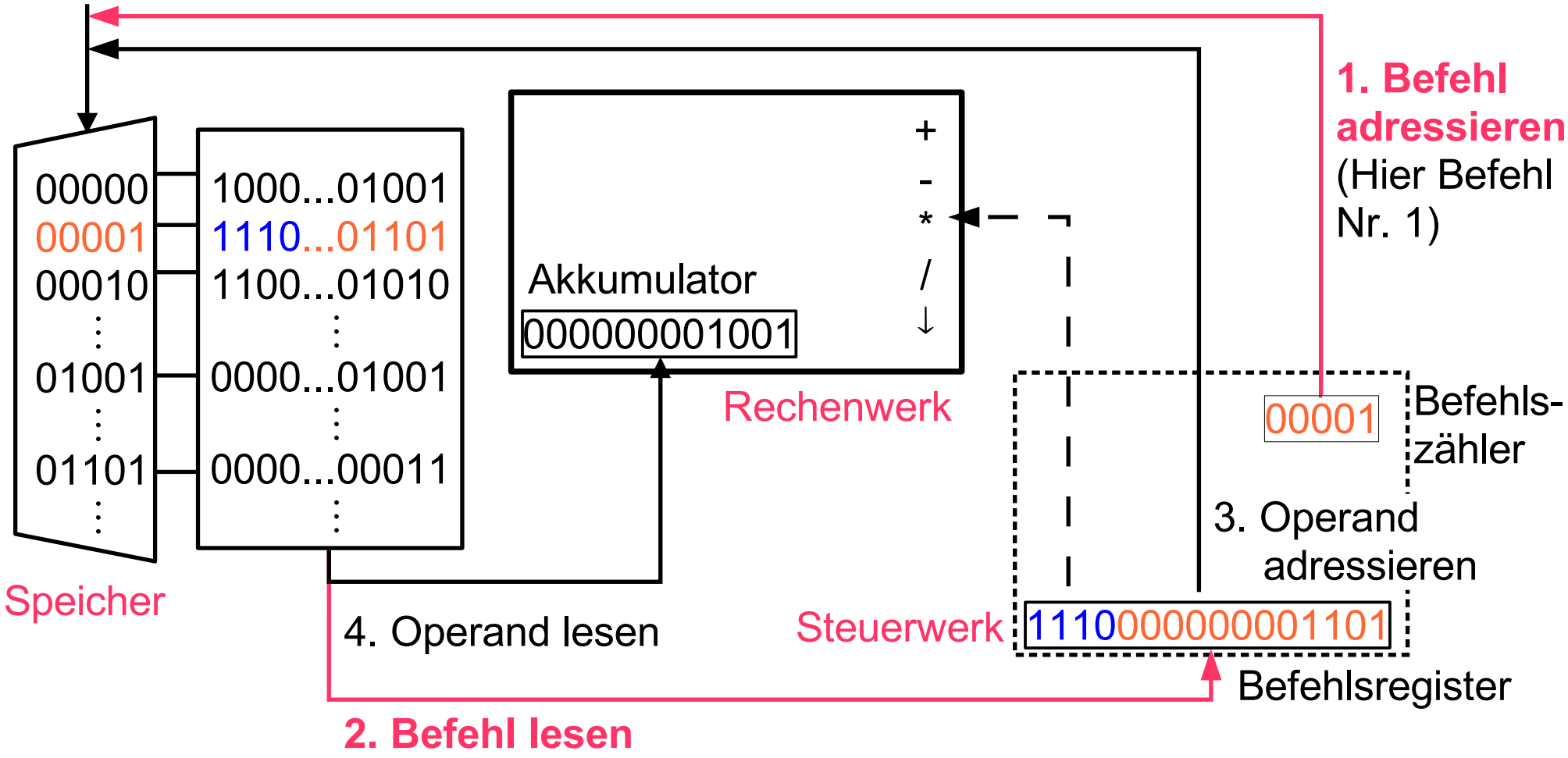
- Das Beispiel zeigt, dass die Befehle und Operanden des Programms als Binärwörter in demselben Speicher abgelegt werden
  - Die Befehle und Operanden sind folglich nur durch ihren Platz im Speicher, d. h. über ihre Adresse, voneinander zu unterscheiden
- Dies ist ein wesentliches Merkmal der **Von-Neumann-Architektur**
- Der abschließende HALT-Befehl im obigen Beispiel verhindert, dass der erste Operand als Befehl interpretiert wird

Das Steuerwerk

- **Ersetzt den Bediener** bei der Durchführung des Programms
- Engl. *Control Unit*, kurz *CU*
- **Enthält ein Register zum Zählen der Befehle:**
  - Den **Befehlszähler** (*Instruction Pointer* (*IP*))
  - Dieser erlaubt die Identifikation des nächsten auszuführenden Befehls
- Das Steuerwerk besitzt ebenfalls ein **Register für die Zwischenspeicherung von Befehlen**:
  - Das **Befehlsregister** (*Instruction Register* (*IR*))
  - Dieses erlaubt das Zerlegen des Befehls in den Operationscode und die Operandenadresse

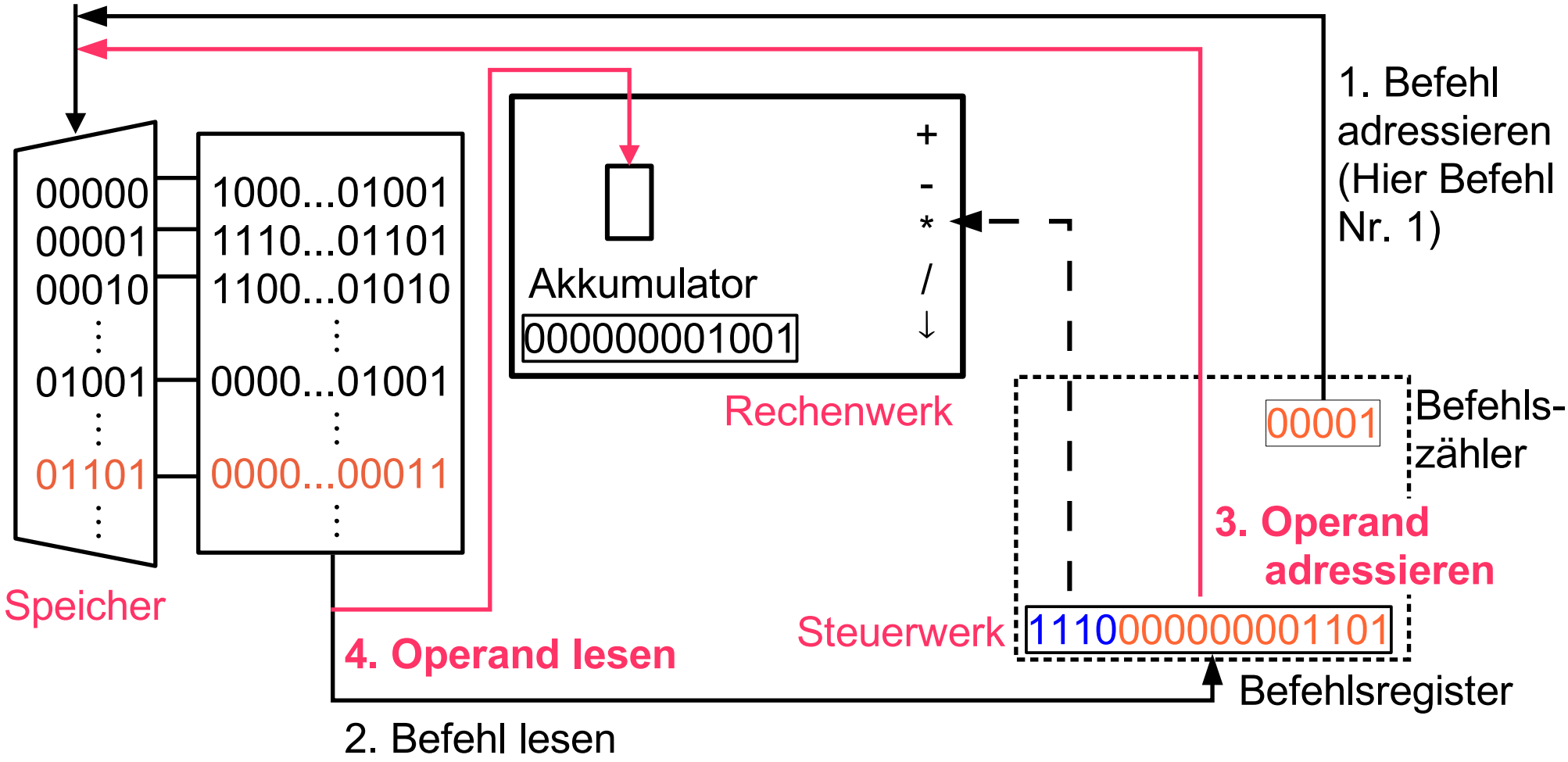
Die Funktionsweise des Steuerwerks (1)

- Der nächste **Befehl** wird aus dem **Speicher** gelesen, in das **Befehlsregister** geladen und decodiert



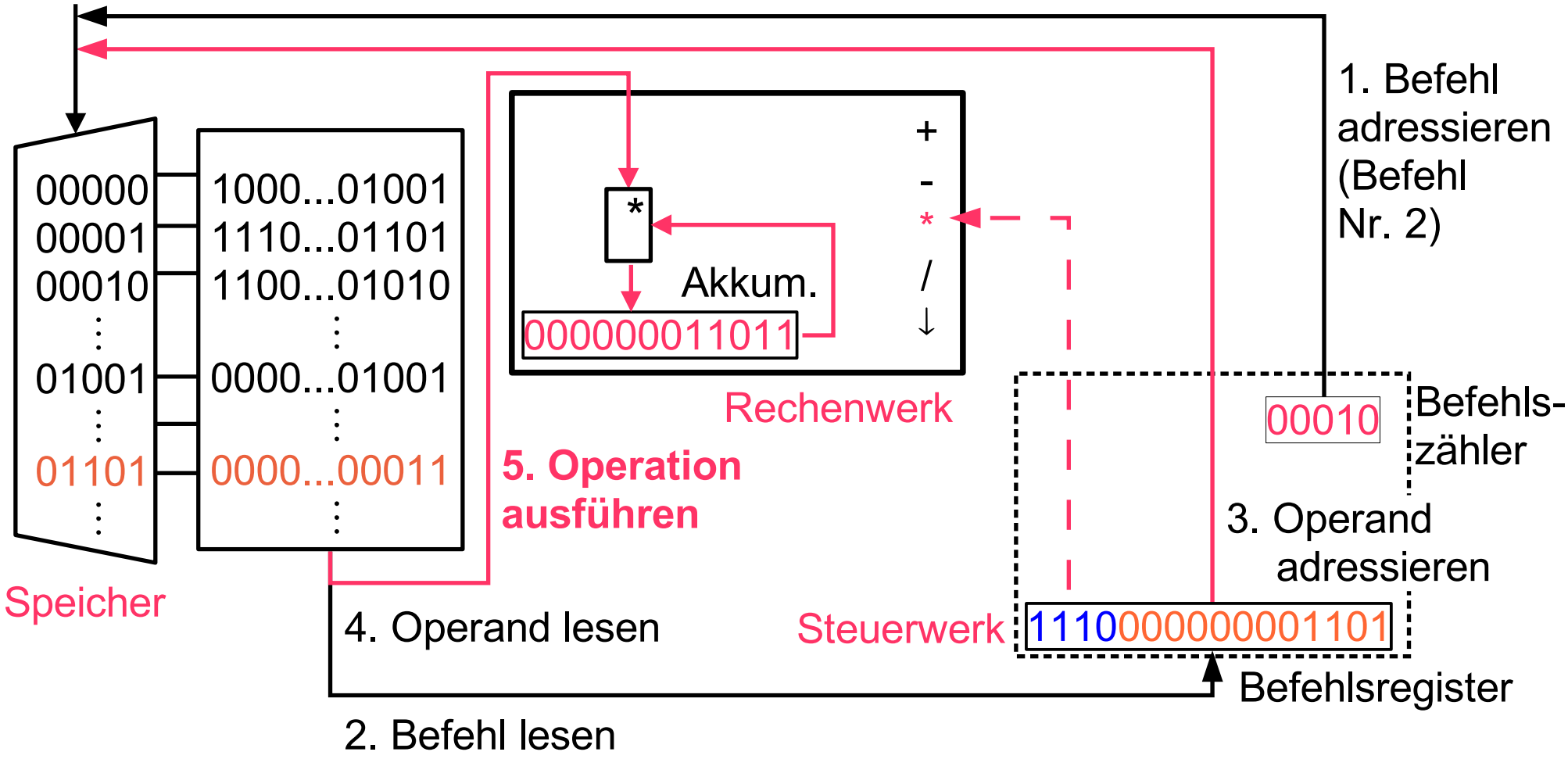
Die Funktionsweise des Steuerwerks (2)

- Der **Operand** wird aus dem Speicher gelesen



Die Funktionsweise des Steuerwerks (3)

- Die **Operation** wird ausgeführt (hier: A ← A*M)
- Der **Befehlszähler** wird danach **inkrementiert**



Programmierung

- Ein für den Menschen verständliches Programm besteht aus mit Hilfe von Buchstaben und Ziffern codierten Befehlen, Variablen, Konstanten oder Zahlen

- Ein für den Rechner verständliches Programm besteht aus Binärwörtern für die Operationen und Dualzahlen für die Werte der Operanden und Adressen

- Es ist daher erforderlich, dass für den Menschen verständliche Programm mittels **Codierung** in ein für die Maschine interpretierbares Programm in **Maschinensprache** zu übersetzen

Beispiel

Für den Menschen verständliche Form		Für die Maschine verständliche Form	
0	↓a3	00000	1000000000001001
1	*x	00001	1110000000001101
2	+a2	00010	1100000000001010
3	*x	00011	1110000000001101
4	+a1	00100	1100000000001011
5	*x	00101	1110000000001101
6	+a0	00110	1100000000001100
7	↑p	00111	0101000000001110
8	HALT	01000	0001000000000000
a3	9	01001	0000000000001001
a2	8	01010	0000000000001000
a1	7	01011	0000000000000111
⋮			

Codierung (1)

- **Die Codierung erfolgt in zwei Arbeitsgängen**
- **1. Arbeitsgang:**
  - Durchnummerierung der Zeilen des Programms sowie der Variablen

    ⇒ Jede Zeilennummer entspricht dann der Adresse einer Speicherzelle

Programm		Adresse	
0	↓a3	00000	0
1	*x	00001	1
2	+a2	00010	2
3	*x	00011	3
4	+a1	00100	4
5	*x	00101	5
6	+a0	00110	6
7	↑p	00111	7
8	HALT	01000	8
a3	9	01001	9
a2	8	01010	10
a1	7	01011	11
⋮			

Codierung (2)

- Auf diese Weise werden Variablennamen mit Speicherzellen bzw. deren Adressen assoziiert
  - Die Speicheradressen werden als Dualzahl geschrieben

Programm		Adresse	
0	↓a3	00000	0
1	*x	00001	1
2	+a2	00010	2
3	*x	00011	3
4	+a1	00100	4
5	*x	00101	5
6	+a0	00110	6
7	↑p	00111	7
8	HALT	01000	8
a3	9	**01001**	9
a2	8	**01010**	10
a1	7	**01011**	11
⋮			

Codierung (3)

- Die Zuordnungen zwischen den symbolischen Adressen und den dualen Speicheradressen werden danach in der sog. **Symboltabelle** erfasst

a_3	01001
a_2	01010
a_1	01011
a_0	01100
x	01101
p	01110

Codierung (4)

- **2. Arbeitsgang:**
  - **Codierung der Befehle**
    - Der Programmierer ordnet mittels der **Codetabelle** den symbolischen Operationscodes binäre Operationscodes zu
    - Danach werden die symbolischen Adressen (Variablennamen) in den Befehlen mit Hilfe der **Symboltabelle** in binäre Adressen übersetzt
  - **Codierung der Daten**
    - Schließlich werden dezimale Zahlenwerte in Dualzahlen umgewandelt

HALT	0001
↑	0101
↓	1000
+	1100
*	1110
⋮	

a_3	01001
a_2	01010
a_1	01011
a_0	01100
x	01101
p	01110

Übung

- Gegeben sei ein 8-Bit-Einadressrechner mit Codetabelle wie zuvor definiert und folgender Aufteilung der Befehle:
  - 4-Bit-Operandenadresse und 4-Bit-Operationscode
  - Die erste Adresse sei 0000
- Schreiben Sie den Ausdruck **a1*x + a2*y + a3** in der gegebenen Reihenfolge der Operationen als Programm auf und codieren Sie ihn danach in Maschinensprache
- Speichern Sie Zwischenergebnisse und das Resultat in der Speicherzelle M
- Verwenden Sie folgende Variablenwerte:
- $a1=27_{10}$, $a2=10_{10}$, $a3=3_{10}$, $x=5_{10}$, $y=2_{10}$

Maschinelle Codierung (1)

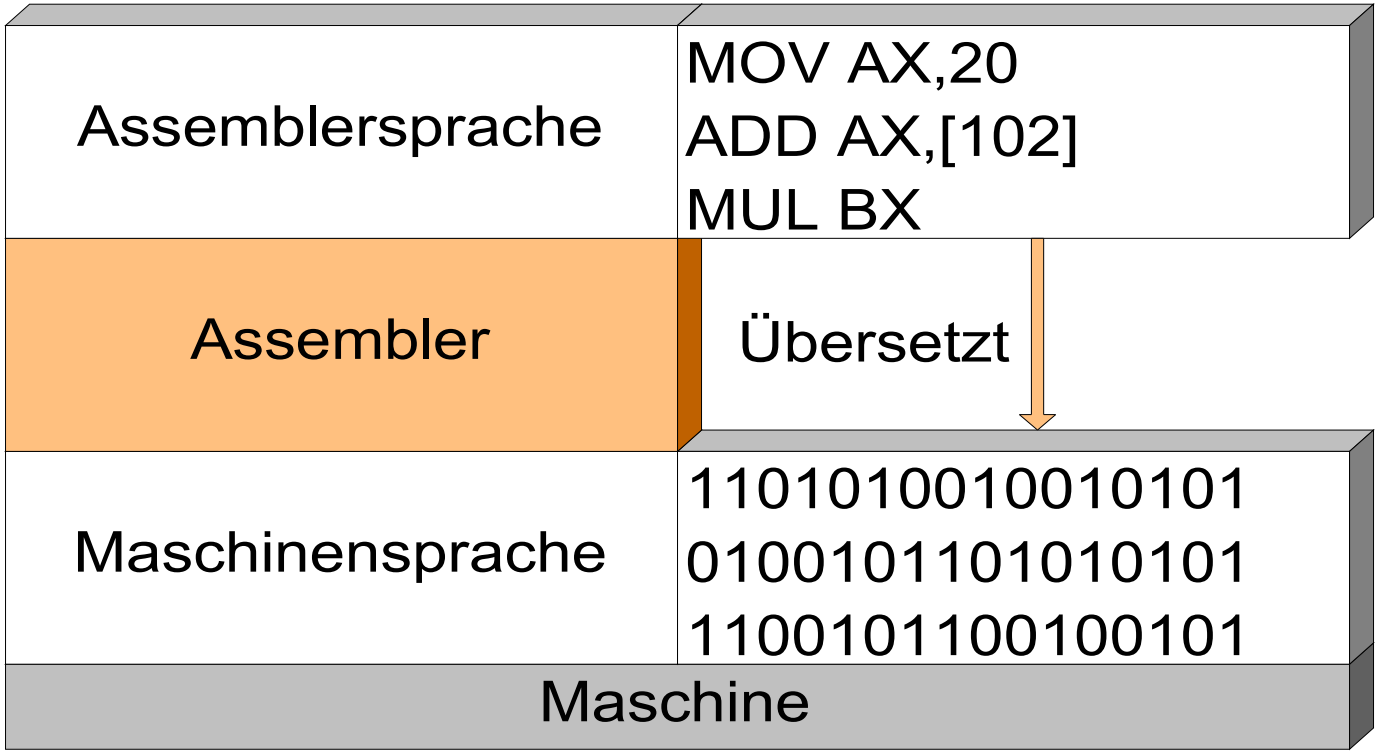
- Der oben beschriebene Codierungsvorgang für die Überführung eines Programms in die Maschinensprache folgt festen Regeln
  - Er ist somit durch einen **Algorithmus** beschreibbar und damit **automatisierbar**
- Der Codierungsvorgang kann daher mit Hilfe eines Programms vom Rechner selbst erledigt und der Mensch von dieser Tätigkeit befreit werden

Maschinelle Codierung (2)

- Ein Programm, welches dies tut, wird als **Assembler** bezeichnet
- Die für den Menschen verständliche Darstellung der Maschinensprache wird deshalb als **Assemblersprache** bezeichnet
- Ein Programm für die Umformung von Maschinensprache in Assemblersprache wird als **Disassembler** bezeichnet

Assemblerprogrammierung

- **Assemblerprogrammierung:**
  - Problemlösung wird in Assemblersprache formuliert
  - Assembler übersetzt Programm (Übersetzungseinheit, Modul) in Maschinensprache



Assembler

- Assembler für neue Prozessoren werden zunächst auf einem schon vorhandenen Rechnertyp entwickelt
- Dabei entsteht ein **Cross-Assembler**
  - Assembler der Maschinencode für einen anderen Prozessor generiert
- Cross-Assembler übersetzt Code des Assemblers für den Zielprozessor
- Danach steht Assembler als Programm auf dem neuen Rechner zur Verfügung
- Frei für den PC verfügbare Assembler:
  - MASM: Microsoft Macro Assembler
  - NASM: Netwide Assembler

Assemblerprogrammierung (1)

- Erlaubt unmittelbaren Zugang zur Hardware
  - Mithin eine sehr maschinennahe Programmierung
    - Z. B. um einen Treiber für eine Grafikkarte zu programmieren oder die USB-Schnittstelle anzusprechen
    - Maschinennahe Programmierung ist in Hochsprachen, wie z. B. Java, oftmals kompliziert oder sogar unmöglich

Assemblerprogrammierung (2)

- Gestattet die **Erzeugung sehr effizienten Codes**
  - Wichtig für zeitkritische Anwendungen
  - Z. B. für Multimedia-Encoder und -Decoder
- Übersetzungsprogramme für Hochsprachen (Compiler) optimieren heutzutage
  - Die Bedeutung der Assemblerprogrammierung nimmt daher in diesem Zusammenhang ab
- **Assemblerprogramme sind schwer portierbar**, da sie eine bestimmte Rechnerarchitektur voraussetzen

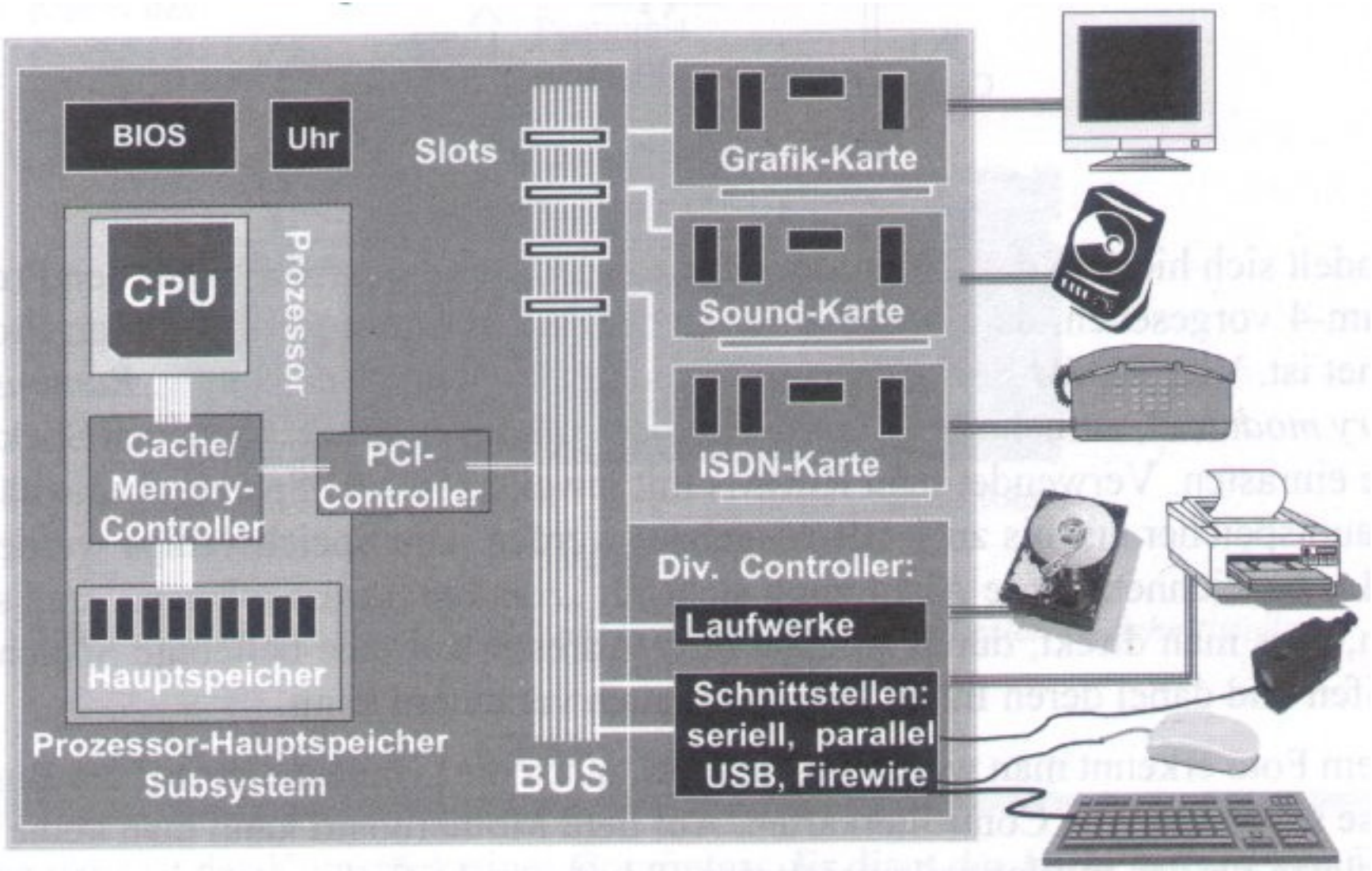
Assemblerprogrammierung (3)

- Assemblerprogrammierung beruht zu einem wesentlichen Teil auf der Arbeit mit **Registern**
  - Siehe **Akkumulatorregister** oben
- Register sind sehr schnelle **Hilfsspeicherzellen**, die direkt mit dem Rechenwerk verbunden sind
  - Moderne Prozessoren verfügen über mehr als ein Register
- Rechenwerk wird heutzutage als **ALU** bezeichnet
  - ALU: *Arithmetical Logical Unit*
  - Führt die arithmetischen und logischen Operationen durch

Assemblerprogrammierung (4)

- Die ALU und die Register sind wesentliche Bestandteile der **CPU**
  - CPU: Central Processing Unit

Rechnerkomponenten



[3]

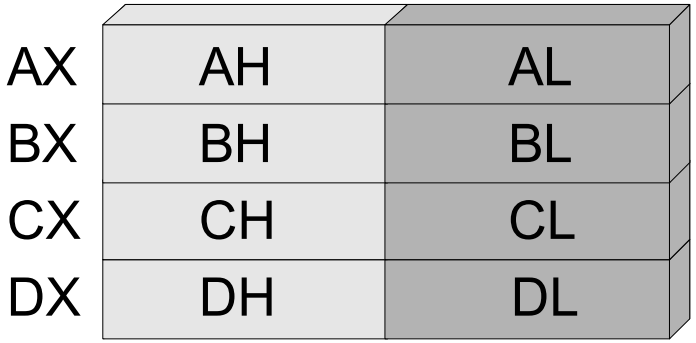
Register und Assemblerbefehle

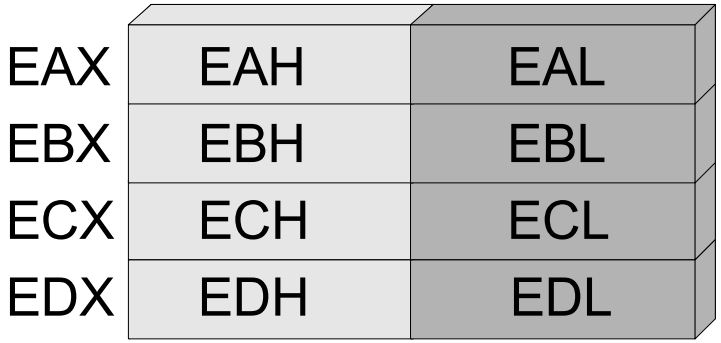
Register

- **Anzahl der Speicherstellen (Bits) gleich der Wortgröße**
  - 8, 16, 32, 64 Bit
  - Man spricht dann von 8-, 16-, 32-, 64-Bit-Rechner
- **Die Register des IBM PC sind in Gruppen aufgeteilt:**
  - Allzweckregister
  - Spezialregister

Allzweckregister (1)

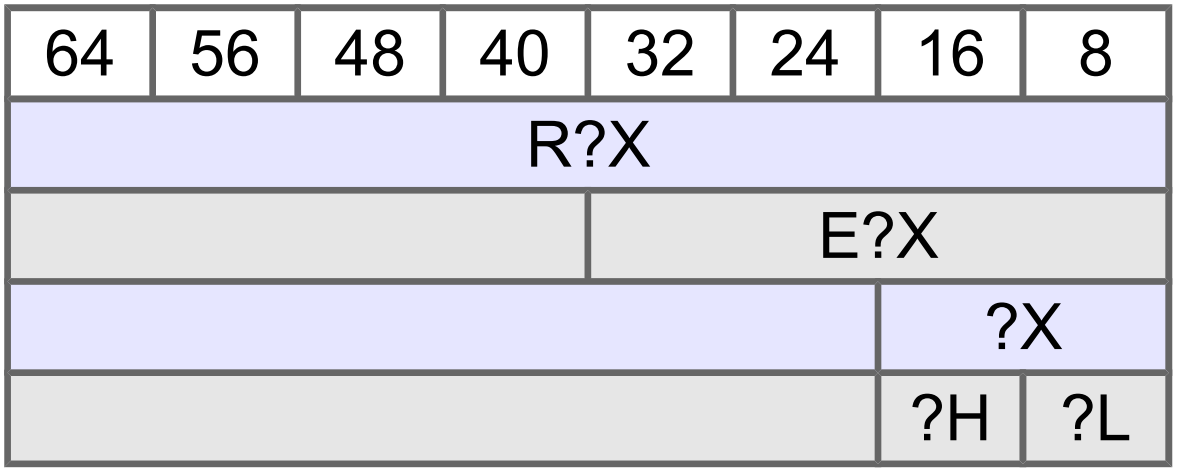
- **8086: 16-Bit breit**
  - AX, BX, CX, DX
- **Seit 80386: 32-Bit breit**
  - EAX, EBX, ECX, EDX
- **Register jeweils in zwei 8- oder 16-Bit-Register unterteilt**
  - Z. B. AH (A-High) und AL (A-Low)





Allzweckregister (2)

- **Seit AMD64 (ab 2000) und Intel-64 (ab 2004)**
  - Spezifikation und schrittweise Einführung 64-Bit breiter Register
  - RAX, RBX, RCX, RDX
  - AMD Opteron (2003), Intel Prescott (2004)
- **Register jeweils in 32/16/8-Bit-Register unterteilt**



Allzweckregister (3)

- Zusätzlich zu ihrer allgemeinen Anwendbarkeit dienen die Allzweckregister auch für spezielle Aufgaben
- **AX:** ***Accumulator***
  - Originäres Register für arithmetische Operationen
- **BX:** ***Base***
  - Normalerweise für indirekte Adressierung benutzt
- **CX:** ***Count***
  - Z. B. als Zähler für Schleifenoperationen
- **DX:** ***Data***
  - E/A-Adressen, Speicher für Überläufe

Assemblerbefehle (1)

- **Assemblerbefehle, die Operationen beschreiben, sind von der Form**

  `op Ziel, Quelle`

- Hat eine Operation zwei Argumente, so werden Ziel und Quelle verknüpft und das Ergebnis in Ziel gespeichert
- In einer Hochsprache könnte dies z. B. wie folgt geschrieben werden:

  `Ziel := Ziel op Quelle` (Pascal, Modula-2)

  `Ziel = Ziel op Quelle` (C, C++, Java, ActionScript)

- Datenquellen: Register, Speicherzellen, Konstanten

Assemblerbefehle (2)

- In Assembler wird immer ein Befehl pro Zeile geschrieben
- Fundamentale arithmetische Operationen:
  - **`ADD, SUB, MOV, INC, DEC, NEG`**
- Zur näheren Erläuterung des Programmcodes werden Kommentare benutzt
  - Kommentare werden mit Semikolon eingeleitet und erstrecken sich bis zum Zeilenende

Arithmetische Operationen: Beispiele

```
ADD AX, BX          ; AX  := AX + BX
SUB AH, AL          ; AH  := AH - AL
MOV AL, CL          ; AL  := CL
INC CX              ; CX  := CX + 1
DEC CL              ; CL  := CL - 1
NEG CX              ; CX  := -CX
```

- Ziel ist immer Register oder Speicherplatz
- Quelle kann auch Zahlenwert (Konstante) sein

Zahlenwerte, Adressen

- Zahlenwerte können binär, oktal, dezimal oder hexadezimal angegeben werden
  - Binär: nachgestelltes *b*, z. B. 0101101010110b
  - Oktal: nachgestelltes *o*, z. B. 176o
  - Dezimal: nachgestelltes *d*, z. B. 587d
  - Hexadezimal: nachgestelltes *h*, z. B. 122h, oder vorangestelltes *0x*, z. B. 0x122
  - Assemblerabhängig!
- Unterscheidung zwischen Adressen und Konstanten durch eckige Klammern
  - Beispiel:
    - 122: Konstante, [122]: Adresse

Beispiele

```
MOV AX, BX        ; AX   := BX
MOV AH, 122       ; AH   := 122
MOV AL, 0xCF      ; AL   := 0xCF (Hex)
MOV CX, [121]     ; CX   := Inhalt von Zelle 121

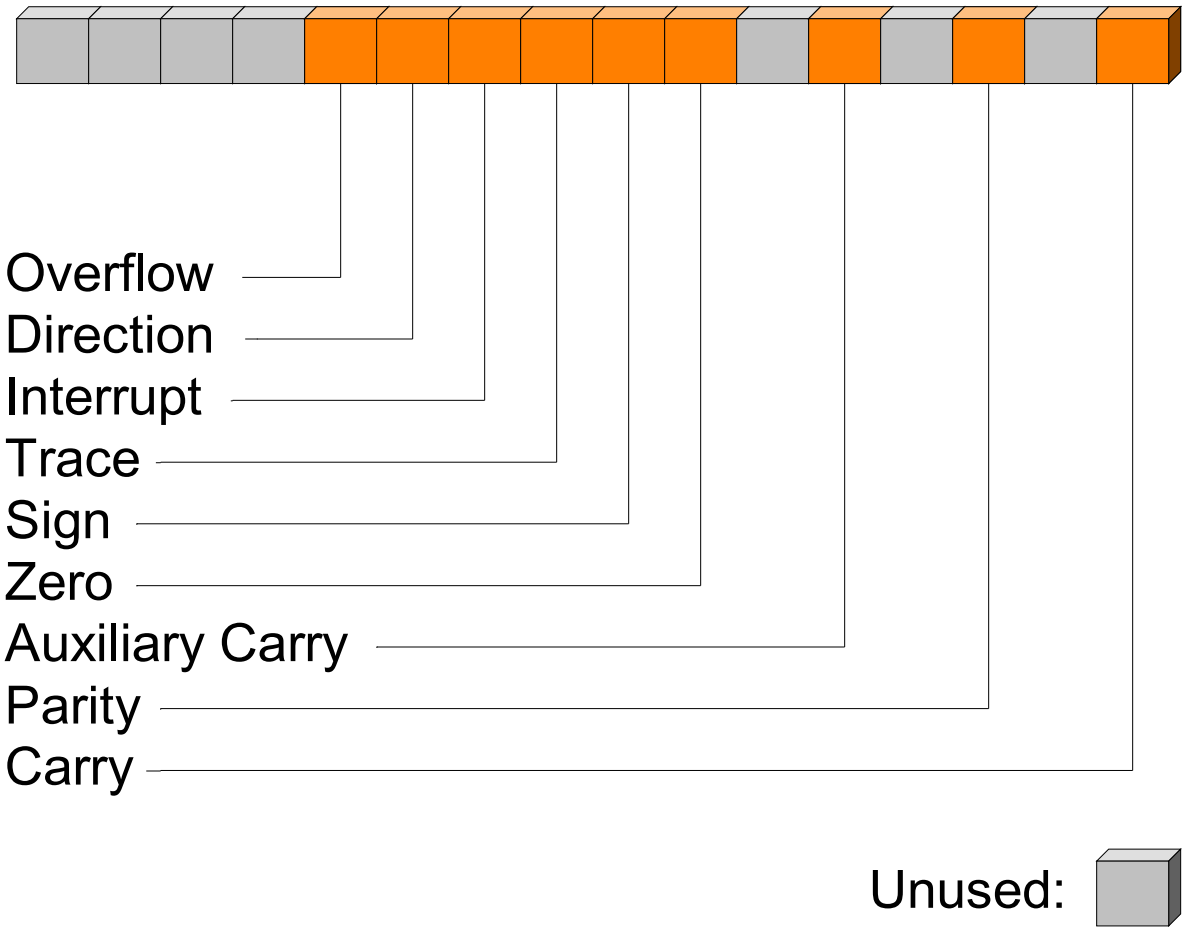
; Fehlerhafte Anweisungen:
MOV AX, BL        ; AX: 16 Bit, BL: 8 Bit
MOV 122, AH       ; Konstante kann kein Ziel
                  ; sein
```

Spezialregister

- **Wichtige Spezialregister in x86 Prozessoren:**
  - Instruction Pointer
    - Enthält die Adresse des gegenwärtig ausgeführten Befehls
    - Befehlszähler
  - Flag-Register

Flag-Register (1)

- Zeigt spezielle Situationen bei der Durch-führung arithmetischer Operationen an
- Situationen durch gesetzte Bits des Registers codiert



Flag-Register (2)

C	Carry	Bereichsüberschreitung für vorzeichenlose Zahlen
A	Auxiliary Carry	Bereichsüberschreitung für vorzeichenlose Nibbles (Hexziffer, 4-Bit, Halb-Byte)
O	Overflow	Bereichsüberschreitung für vorzeichenbehaftete Zahlen
S	Sign	Ergebnis ist negativ
Z	Zero	Ergebnis ist Null
P	Parity	Ergebnis ist Binärwort mit gerader Anzahl Einsen
D	Direction Flag	Legt fest, ob Strings bei String-Befehlen von links nach rechts oder umgekehrt abgearbeitet werden
I	Interrupt	Legt fest, ob der Prozessor auf Unterbrechungen (z. B. Tastatureingaben) reagieren soll oder nicht
T	Trace	Für Debugging: Erlaubt schrittweise Abarbeitung des Codes

Flag-Register (3)

C	Carry	Bereichsüberschreitung für vorzeichenlose Zahlen
O	Overflow	Bereichsüberschreitung für vorzeichenbehaftete Zahlen
S	Sign	Ergebnis ist negativ
Z	Zero	Ergebnis ist Null

- **Gestatten Interpretation einer arithmetischen Operation**
- **Prozessor kennt keine "Typen"**
  - Prozessor "weiß" nicht, ob ein Registerinhalt z. B. als natürliche oder ganze Zahl zu interpretieren ist
  - Registerarithmetik funktioniert immer

Flag-Register (4)

- **Beispiel:**
  - Arithmetik mit natürlichen, d. h. vorzeichenlosen Zahlen
  - *Carry- und Zero-Bit relevant!*
  - Mittels *Carry-Bit* lässt sich z. B. ablesen, ob Übertrag aus der höchsten Bitposition eingetreten ist
  - Dann Bereichsüberschreitung und Ergebnis ungültig
  - Zero-Bit zeigt an, ob Ergebnis gleich Null

Flag-Register (5)

- **Beispiel:**
  - Arithmetik mit ganzen (Zweierkomplement-) Zahlen
  - *Overflow-, Sign- und Zero-Bit relevant!*
  - Mittels Overflow-Bit lässt sich ablesen, ob Bereichsüberschreitung eingetreten ist

    => Ergebnis ungültig
  - Zero-Bit zeigt an, ob Ergebnis gleich Null
  - Sign-Bit zeigt an, ob Ergebnis negativ

Beispiel

- **16-Bit 8086:**

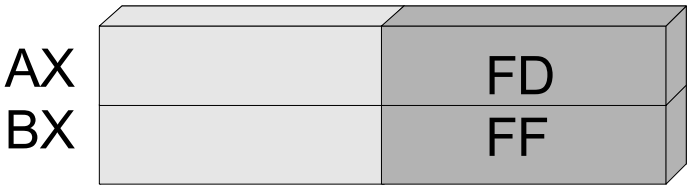
```
MOV AL, 0xFD
MOV BL, 0xFF
ADD AL, BL
```

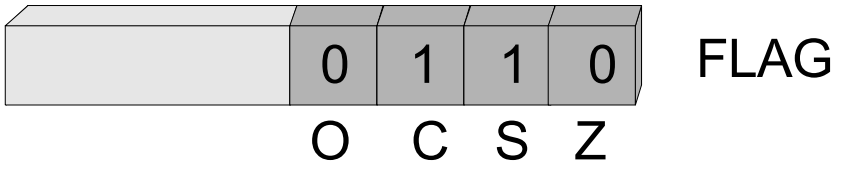
```
MOV AL, 253
MOV BL, 255
ADD AL, BL
```

Carry signalisiert
ungültiges Ergebnis

```
MOV AL, -3
MOV BL, -1
ADD AL, BL
```

Overflow signalisiert
ungültiges Ergebnis





Ergebnis = 252
Ungültig: C=1!

```
  11111101
 +11111111
 ---------
 111111100
```

Ergebnis = -4
Gültig, da O=0

Assemblerprogrammierung

Beispiel: Polynomauswertung

$$p = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 =$$

$$((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0$$

mit: $a_3 = 9$, $a_2 = 8$, $a_1 = 7$, $a_0 = 6$ und $x = 3$

- Addition und Multiplikation erforderlich

Multiplikation mit Dualzahlen

- **"Kleines 1-mal-1" für Dualzahlen:**

  0*0 = 0, 0*1 = 0, 1*0 = 0 und 1*1 = 1

- Rückführung auf Verschiebung (Shift) mit Addition
- **Beispiel: 86*21**
- $1010110_2 * 10101_2$

```
                1 0 1 0 1 1 0       10101
+             0 0 0 0 0 0 0         1010
+           1 0 1 0 1 1 0           101
+       0 0 0 0 0 0 0               10
+ 1 0 1 0 1 1 0                     1
-------------------------------------
  1 1 1 0 0 0 0 1 1 1 0₂ = 1806₁₀
```

- **Resultat wesentlich länger als die ursprünglichen Faktoren!**

Multiplikation/Division

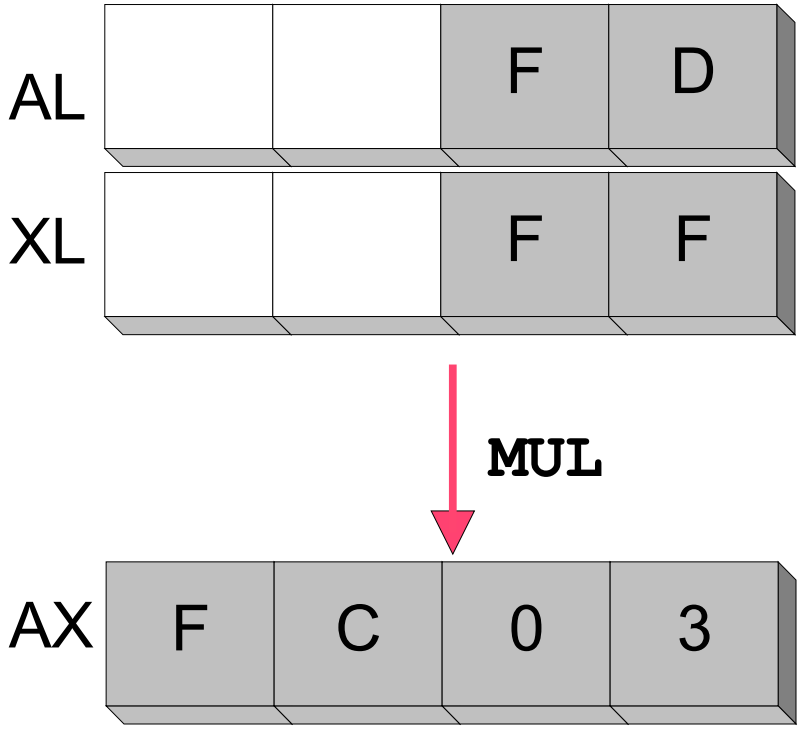
- **Problem:** Resultat passt möglicherweise nicht in das Zielregister
- Übliche Form **`op Ziel,Quelle`** daher nicht anwendbar

Multiplikation: Vorzeichenlose Zahlen (1)

- **8-Bit-Multiplikation**
  - 8-Bit-Multiplikand im AL Register erwartet
  - Multiplikator steht in einem 8-Bit-Allzweckregister oder im Speicher (keine Konstante erlaubt)
  - Assemblerbefehl:

```
MUL Multiplikator
```

  - Einadressbefehl
  - Ergebnis ist 16-Bit Wert in AX



Multiplikation: Vorzeichenlose Zahlen (2)

- **16-Bit-Multiplikation**
  - 16-Bit-Multiplikand in AX
  - 16-Bit-Multiplikator in einem Allzweckregister oder Speicher
  - Ergebnis ist 32-Bit-Zahl in DX:AX
    - Höherwertige 16 Bits sind in DX, niederwertige in AX
- **32-Bit-Multiplikation**
  - 32-Bit-Multiplikand in EAX
  - 32-Bit-Multiplikator in einem Allzweckregister oder Speicher
  - Ergebnis ist 64-Bit-Zahl in EDX:EAX
- **64-Bit-Multiplikation**
  - Dito mit 64-Bit-Multiplikand in RAX u. 64-Bit-Multiplikator
  - Ergebnis ist 128-Bit-Zahl in RDX:RAX

Division: Vorzeichenlose Zahlen

- **8-Bit-Division**
  - 16-Bit-Dividend in AX geteilt durch 8-Bit-Divisor in Allzweckregister oder Speicher:
    `DIV Divisor`
  - Quotient liegt in AL, Rest liegt in AH
- **16- bis 64-Bit-Division:**

Division	Divident	Divisor	Quotient in	Rest in
16-Bit	32-Bit in DX:AX	16-Bit	AX	DX
32-Bit	64-Bit in EDX:EAX	32-Bit	EAX	EDX
64-Bit	128-Bit in RDX:RAX	64-Bit	RAX	RDX

  - Divisor entstammt jeweils Allzweckregister oder Speicher

Vorzeichenbehaftete Zahlen

- **Multiplikation: `IMUL` (*integer multiply*)**
- **Division: `IDIV` (*integer divide*)**
  - Ansonsten wie zuvor

Beispiel: Polynomauswertung

$p = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 = ((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0$

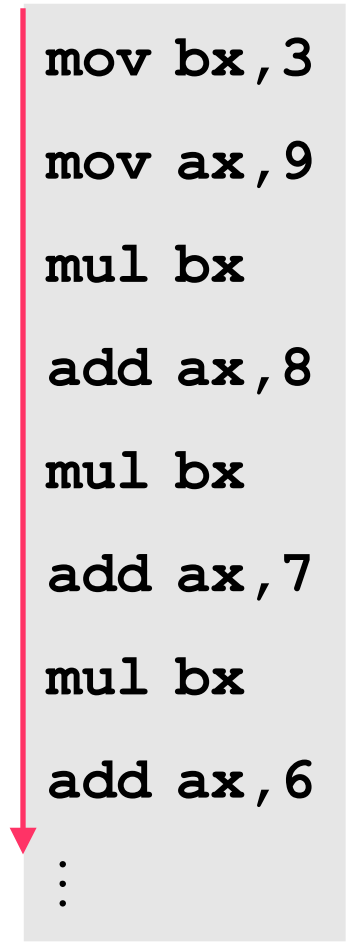
mit: $a_3 = 9$, $a_2 = 8$, $a_1 = 7$, $a_0 = 6$ und $x = 3$

Nr. der Operation	Art der Operation	Assembler	Kommentar
0	BX := 3	mov bx,3	;lade BX-Reg. mit 3: x=3
1	AX := 9	mov ax,9	;lade AX-Reg. mit 9: p=a3
2	*x	mul bx	;AX := AX * BX
3	$+a_2$	add ax,8	;addiere a2=8 zu AX
4	*x	mul bx	;AX := AX * BX
5	$+a_1$	add ax,7	;addiere a1=7 zu AX
6	*x	mul bx	;AX := AX * BX
7	$+a_0$	add ax,6	;addiere a0=6 zu AX

Sprungbefehle

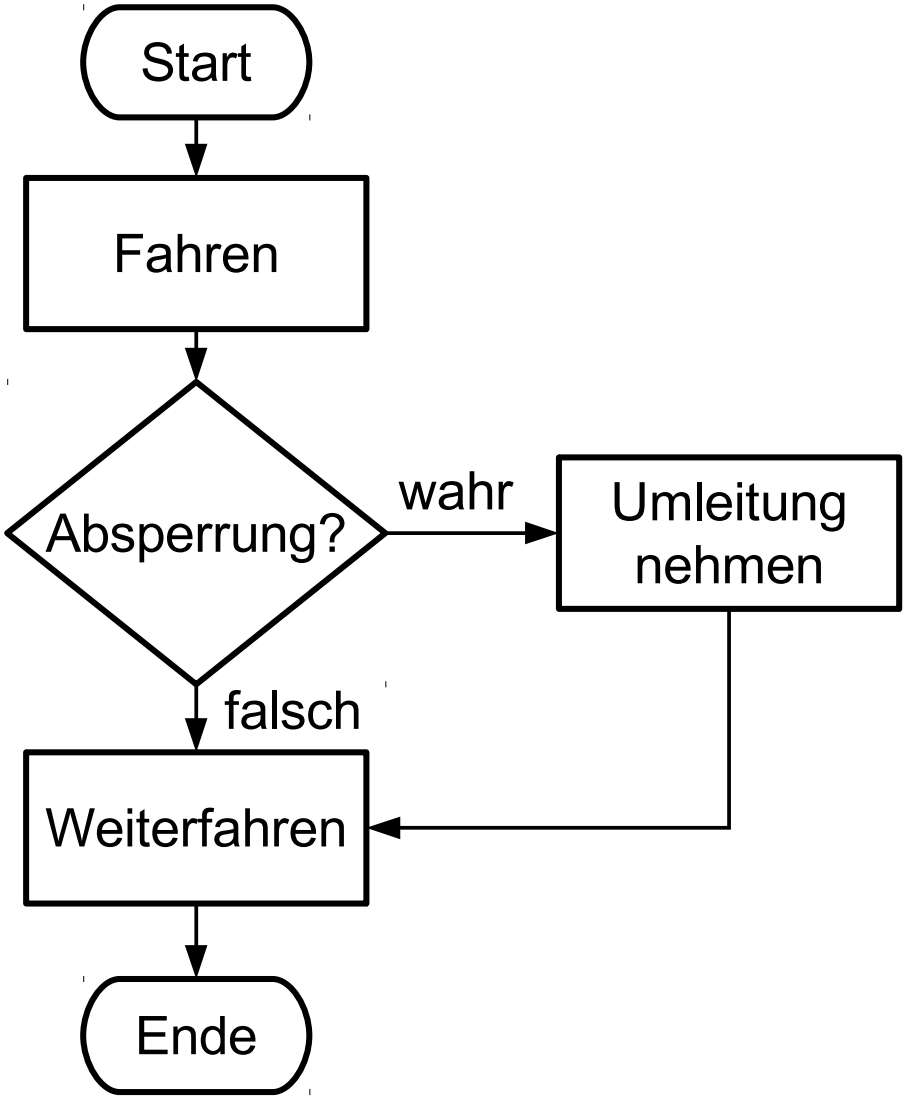
Assemblerbefehle: Ausführung

- Assemblerbefehle werden in der Reihenfolge ausgeführt, in der sie im Assemblercode erscheinen
- Häufig ist bei der Bearbeitung von Problemen eine Entscheidung zwischen mehreren Lösungswegen erforderlich
- Wir benötigen daher eine Möglichkeit, die normale Befehlsreihenfolge ändern zu können

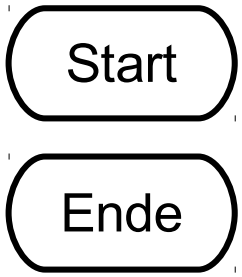


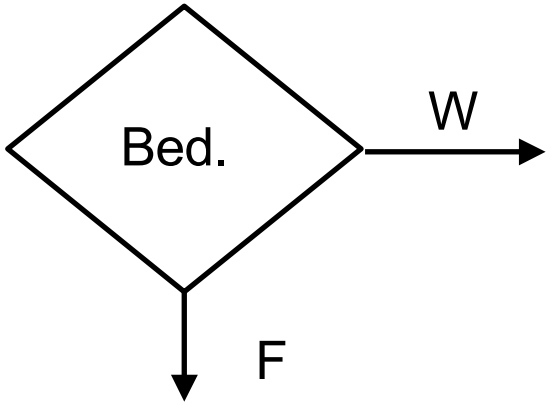
Beispiel

- **Flussdiagramm**
- Instruktive grafische Notation für einen **Algorithmus**
  - Detaillierte und explizite Vorschrift zur schrittweisen Lösung eines Problems



Flussdiagramm: Symbole

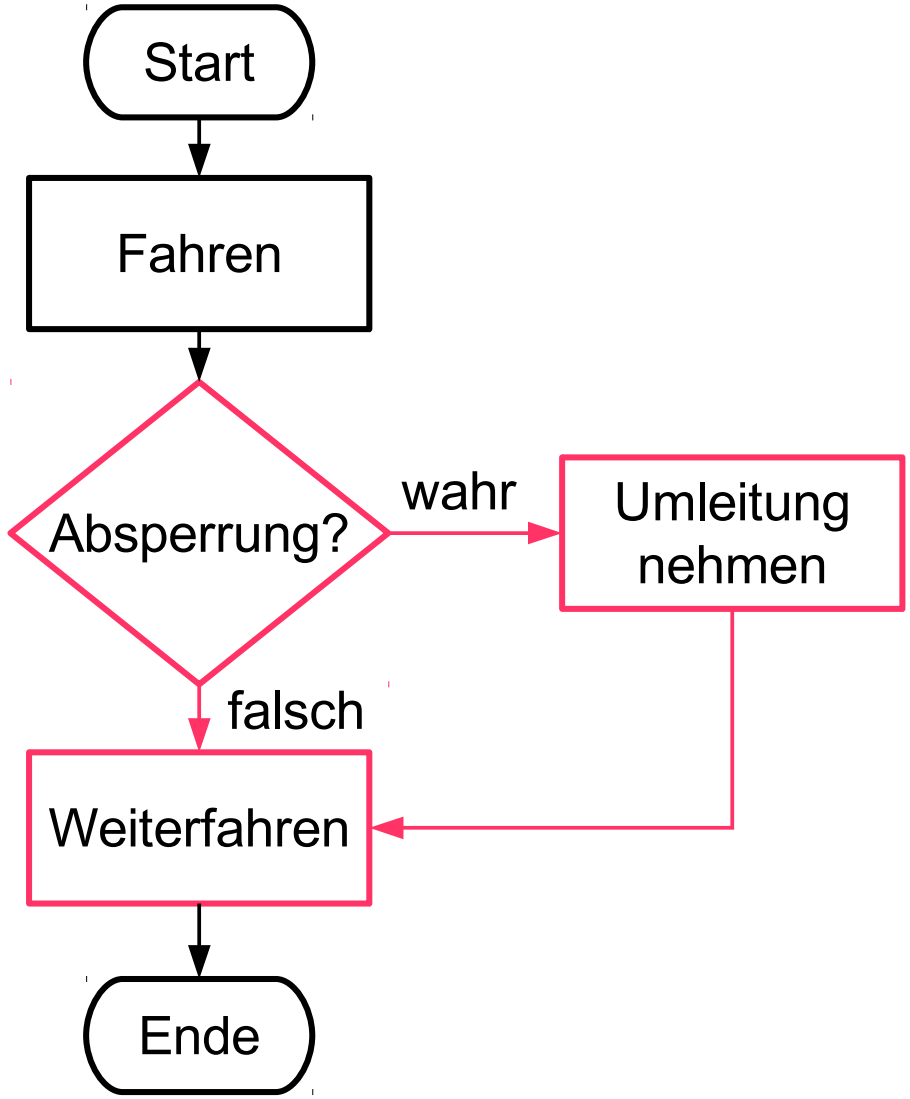




- **Zustand**
  (z.B. Anfangs- oder Endzustand)
- **Aktion** (z. B. Zuweisung)
- **Verbinder** verbindet Zustände und Aktionen; dargestellt durch Pfeil, der auf die nächste auszuführende Aktion zeigt
- **Entscheidung**: Ist Bedingung erfüllt folge W(ahr)-Zweig, sonst folge F(alsch)-Zweig

Beispiel

- Wie lässt sich eine Entscheidung in Assemblersprache programmieren?



Sprungbefehle (1)

- Sprungbefehle ermöglichen eine Unterbrechung der linearen Befehlsreihenfolge
- Programm wird an einer anderen, durch einen **Namen** markierten Stelle des Codes fortgesetzt
  - **Sprungmarke** oder **Label**
- Dies geschieht, wenn eine bestimmte logische Bedingung erfüllt ist

```
   mov bx,3
   mov ax,9
   ; falls Bedingung
   ; erfüllt gehe zu
   ; label1
   mul bx
   add ax,8
label1:
   mul bx
   add ax,7
   mul bx
   add ax,6
       ⋮
```

Sprungbefehle (2)

- **Sprungarten:**
  - Unbedingter Sprung
    - Sprung findet auf jeden Fall statt, d. h. es wird keine logische Bedingung getestet
    - `JMP` Befehl
  - Bedingter Sprung
    - Sprung findet nur dann statt, wenn eine bestimmte logische Bedingung erfüllt ist
    - Bedingung wird mittels `CMP` unter Auswertung des *Flag-Registers* geprüft
    - Falls Bedingung falsch wird Programm mit Befehl hinter der bedingten Sprunganweisung fortgesetzt
    - Es gibt unterschiedliche bedingte Sprungbefehle

Bedingte Sprünge

- **`JZ`: Jump on zero (auch als `JE`: Jump on equal)**
  - Wird ausgeführt, falls Zero-Flag nach Vergleich gesetzt ist
- **`JNZ`: Jump if no zero (auch als `JNE`: Jump on NOT equal)**
  - Wird ausgeführt, falls Zero-Flag nach Vergleich NICHT gesetzt ist

Beispiel

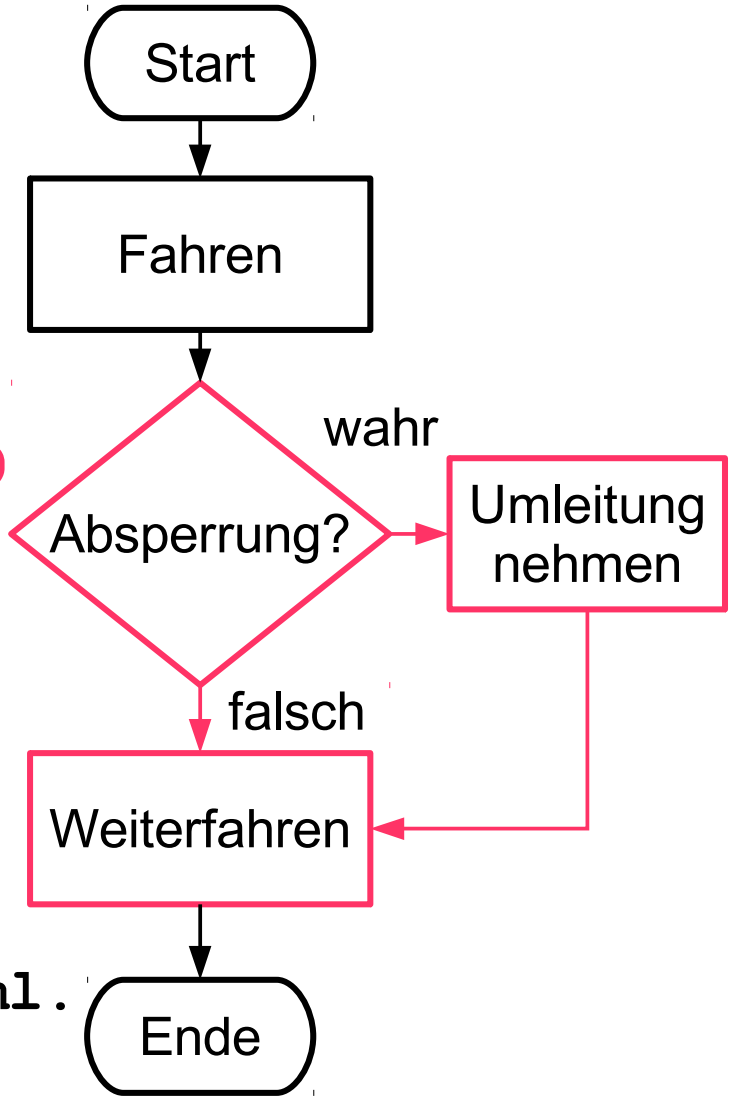
- Assembler:

```
; Befehle für "Fahren"
; ax > 0 :"Absperrung",
; ax = 0 :"Keine Absperrung"
cmp ax,0 ; Zero flag set if ax-0=0
jz weiterfahren
; Befehle für Umleitung
weiterfahren:
; Befehle für "Weiterfahren"
```

Label

- Hochsprache: `if (absperrung)… // uml.`

`… // weiterfahren`



Weitere Sprungbefehle (1)

- Sprungbefehle für den Vergleich **vorzeichenloser** Zahlen

Sprungbefehl	Bedeutung	Bedingung an Flags
`JA`	Jump Above (>0)	C=0 und Z=0
`JAE`	Jump Above or Equal (≥0)	C=0
`JB`	Jump Below (<0)	C=1
`JBE`	Jump Below or Equal (≤0)	C=1 oder Z=1

Weitere Sprungbefehle (2)

- Sprungbefehle für den Vergleich **vorzeichenbehafteter** Zahlen

Sprungbefehl	Bedeutung	Bedingung an Flags
`JG`	Jump Greater	S=0 und Z=0
`JGE`	Jump Greater or Equal	S=0
`JL`	Jump Less	S≠1
`JLE`	Jump Less than or Equal	S≠1 oder Z=1

Weitere Sprungbefehle (3)

- Sprungbefehle, mit denen sich einzelne Flags abprüfen lassen

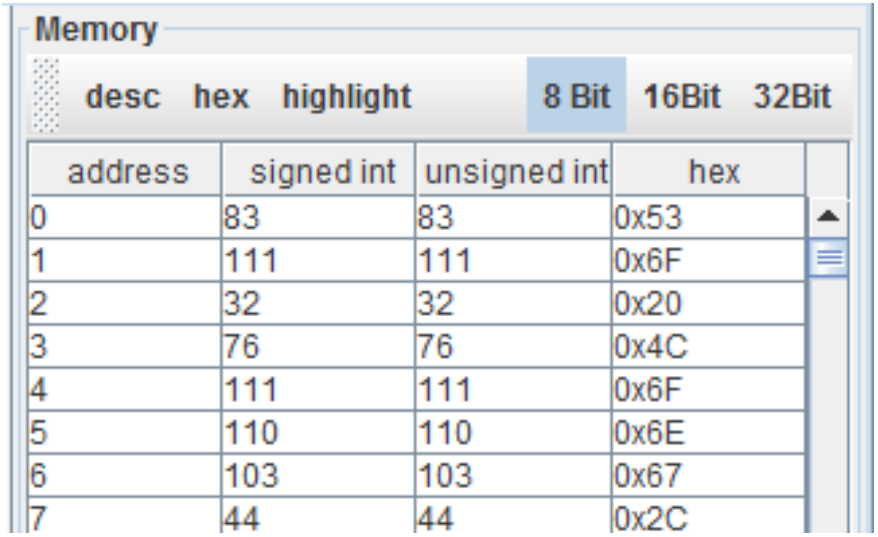
Sprungbefehl	Bedeutung	Bedingung an Flags
`JC, JO`	Jump if Carry/Overflow	C=1 bzw. O=1
`JNC, JNO`	Jump if not Carry/Overflow	C=0 bzw. O=0

Ein- und Ausgabe

- Wird über I/O-Ports abgewickelt
- Häufige Form: Memory-Mapped I/O
- Kommunikation mit Peripherie:
  - Keyboard, Festplatte, Soundkarte, Grafikkarte (VGA, HDMI), DVD-Player, Parallelport, USB-Port usw.,
- läuft über bestimmte Adressen im Hauptspeicher
- Beim PC der untere Adressbereich ab Adresse 0
- Jasmin folgt diesem Ansatz

Ein- und Ausgabe: Jasmin

- Keine Eingabe
- Ausgabe über den unteren Speicherbereich
- Bytes, die in die Adressen 0..n eingeblendet werden, erscheinen in der
  - **`Console`** oder im
  - (7-Segment-)**`Display`**
- Bytes für die **`Console`** werden als Latein-1-Codes interpretiert



address	signed int	unsigned int	hex
0	83	83	0x53
1	111	111	0x6F
2	32	32	0x20
3	76	76	0x4C
4	111	111	0x6F
5	110	110	0x6E
6	103	103	0x67
7	44	44	0x2C

Hauptspeicherzugriffe (1)

- Für das Schreiben und Lesen von Daten in bzw. aus dem Hauptspeicher werden die Adressen der zugehörenden Speicherzellen benutzt
- Adressen werden durch eckige Klammern kenntlich gemacht
- Beispiele:
  - Laden von zwei Byte aus Speicherzelle 100 in **AX**-Register

    ```
    mov ax,[100]
    ```

  - Speichern von zwei Byte in Speicherzelle 100 aus **AX**-Register

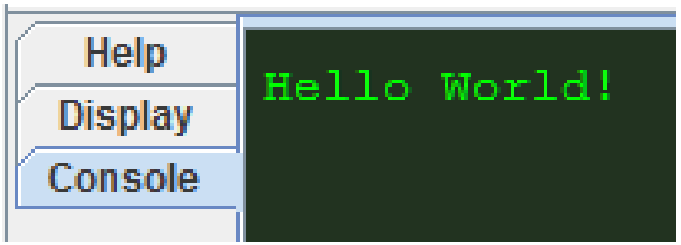
    ```
    mov [100],ax
    ```

Hauptspeicherzugriffe (2)

- Die Anzahl der gelesenen bzw. geschriebenen Bytes ist von der Registergröße abhängig
- **`al`**, **`bl`**,…: 1 Byte
- **`ax`**, **`bx`**,...: 2 Byte
- **`eax`**, **`ebx`**,...: 4 Byte
- Jasmin:
  - Zahlen werden immer als 4-Byte-Werte geschrieben
  - Je nach Größe der Zahl können Bytes höherer Adressen dabei 0 enthalten
  - MSB liegt in der höchstwertigen Adresse
  - Zahlen mit geringerer Bytezahl können durch „Überlappen“ der Speicherbereiche geschrieben werden

Beispiel

- Ausgabe von „Hello World“ auf die Jasmin-Konsole
- ```
  mov [0], 01001000b ; H
  mov [1], 01100101b ; e
  mov [2], 01101100b ; l
  mov [3], 01101100b ; l
  mov [4], 01101111b ; o
  mov [5], 00100000b ; sp
  mov [6], 01010111b ; W
  mov [7], 01101111b ; o
  mov [8], 01110010b ; r
  mov [9], 01101100b ; l
  mov [10],01100100b ; d
  mov [11],00100001b ; !
  ```
- Zeilenumbruch mit **LF**=10


  ```

Assemblerdirektiven

- Speicherbereiche können mit Hilfe einer Assemblerdirektive auch in einem Zug beschrieben werden
  - Assemblerdirektiven sind Befehle für den Assembler
- Direktiven für das Beschreiben des Speichers
  - **`db`**: Byte, **`dw`**: Word (16-Bit), **`dd`**: Double Word (32-Bit)
  - Jasmin schreibt ab Adresse 0
- Beispiele:
- `db 'So Long,', 10, 'and Thanks For All the Fish', 10`
- `dw 'So Long,', 10, 'and Thanks For All the Fish', 10`
- `dd 'So Long,', 10, 'and Thanks For All the Fish', 10`
  - Zeilenumbruch: 10

Indizierter Hauptspeicherzugriff

- Allzweckregister ermöglichen Hauptspeicherzugriff auf der Grundlage von Adressrechnungen
- Beispiel: Zugriff auf Speicherzelle 100 mit **BX**-Register

```
mov bx,100   ; Lade Adresse 100

mov ax,[bx]  ; Lade Wert aus Adresse 100
```

- Adressrechnung erlaubt das Berechnen von Speicherzellenpositionen im Hauptspeicher
  - Voraussetzung für das systematische Abarbeiten von einer Reihe von Zahlenwerten im Hauptspeicher

Reihenberechnungen

- Wir wollen die Summe der natürlichen Zahlen von 1 bis *n* berechnen:
  - D. h. $s = 1 + 2 + 3 + ... + n$
- Dies wird in der Mathematik als **Reihe** bezeichnet und mittels Summenzeichen wie folgt abgekürzt:

$$s = \sum_{i=1}^{n} i$$

- Für dieses Problem existieren unterschiedliche Lösungswege
  - **Ein iterativer Lösungsweg**
  - Ein rekursiver Lösungsweg: Dieser wird später betrachtet
  - Und eine geschlossene Lösung: $n*(n+1)/2$

Iterative Lösung

- Die iterative Lösung entspricht dem ausformulierten Summenzeichen
- Dieser Algorithmus lässt sich wie folgt beschreiben:
  - Initialisierung:
    - Setze Summe *s* auf 0
    - Setze Zähler (Zählvariable) *i* auf Startwert 1
  - Wiederhole *n* Mal:
    - Addiere *i* zu *s*
    - Inkrementiere *i*
- Die Berechnung beruht auf einer sogenannten **Schleife**

Assemblerprogramm

```
; Summe der natürlichen Zahlen 1..10
mov bx,0 ; bx = s
mov ax,1
addiere:
   add bx,ax
   inc ax
   cmp ax,10
   jle addiere
```

Schleife

LOOP-Befehl (1)

- Schleife mittels **`CMP`** umständlich
- x86 bietet für Schleifenstrukturen den **`LOOP`**-Befehl
- Der Befehl nutzt das **`CX`**-Register als Zählvariable (C: Count)
  - **`CX`** enthält Anzahl der verbleibenden Durchläufe (Iterationen)
- **`LOOP`**-Befehl dekrementiert **`CX`** und springt an den Anfang der Schleife falls **`CX`**>0
  - D. h. die Schleife wird rückwärts von *n*..1 durchlaufen!

LOOP-Befehl (2)

- Allgemeines Schema:

```
; Code vor der Schleife
mov cx,k ; k Durchläufe gewünscht
Schleife:
   ; Befehl_1
   ; Befehl_2
   ; ...
   ; Befehl_n
   loop Schleife
```

Zahlen summieren mit CX-Register

```
; Summe der natürlichen Zahlen 1..10
mov ax,0   ; ax = s = 0 zu Beginn
mov cx,10 ; Anzahl der Durchläufe ist 10
schleife:
   add ax,cx
   loop schleife
```

Summieren von Zahlen aus dem Hauptspeicher

- Bisher wurden fortlaufende Zahlenfolgen verarbeitet
- Beliebige Zahlenfolgen, z. B. 1, 12, 17,..., erfordern eine Zwischenspeicherung der Zahlen im Hauptspeicher
- Für das Schreiben und Lesen von Daten in bzw. aus dem Hauptspeicher werden die Adressen der zugehörenden Speicherzellen benutzt
- Benutzte Zahlenwerte: **1, 12, 17, 25, 32, 41, 50**
  - Deren Summe ist **178**

Aufsummieren von Zahlen aus dem Speicher

```
mov [100],1    ; Schreiben der Zahlenwerte
mov [102],12   ; Für EAX etc. Register 32-Bit-Werte,
mov [104],17   ; d. h. 4 Byte pro Adresse vorsehen
mov [106],25
mov [108],32
mov [110],41
mov [112],50

mov ax,0       ; Summe ist 0 zu Beginn
mov bx,100     ; Startadresse ist 100
mov cx,7       ; 7 Zahlenwerte sind zu verarbeiten

schleife:
  add ax,[bx]  ; Indizierter Speicherzugriff
  add bx,2     ; Adresse des nächsten Zahlenwerts
  loop schleife
```

Shift-Operationen (1)

- Verschieben Registerinhalt um eine oder mehrere Stellen nach rechts oder links
  - Shift um eine Position nach links entspricht Multiplikation mit zwei
  - Shift um eine Position nach rechts entspricht Division durch zwei
- Dabei fallen rechts oder links Bits heraus
  - Letztes herausgefallenes Bit gelangt in Carry-Flag
- Häufig benutzt als schnelle Variante für Multiplikation/Division mit Zweierpotenz

Shift-Operationen (2)

- Herausgefallene Bits ziehen auf der anderen Seite Lücken nach, die wieder aufgefüllt werden müssen
  - Art der Operation bestimmt, wie gefüllt wird
- **`SHR`**: Shift unsigned right
- **`SHL`**: Shift unsigned left
  - Lücken werden mit einer 0 gefüllt
  - Beispiel: **`SHL AL,1`**
- **`SAR`**: Shift arithmetic right
  - MSB-(Vorzeichen-)Bit bleibt erhalten
- **`SAL`**: Shift arithmetic left
  - Äquivalent zu **`SHL`**; Lücken werden mit 0 gefüllt

Unterprogramme

Überblick

- **Unterprogramme**
  - Einfache Unterprogramme
  - Parametrisierte Unterprogramme
- **Stapel**
- **Zeiger**
- **Rekursion**

Warum Unterprogramme?

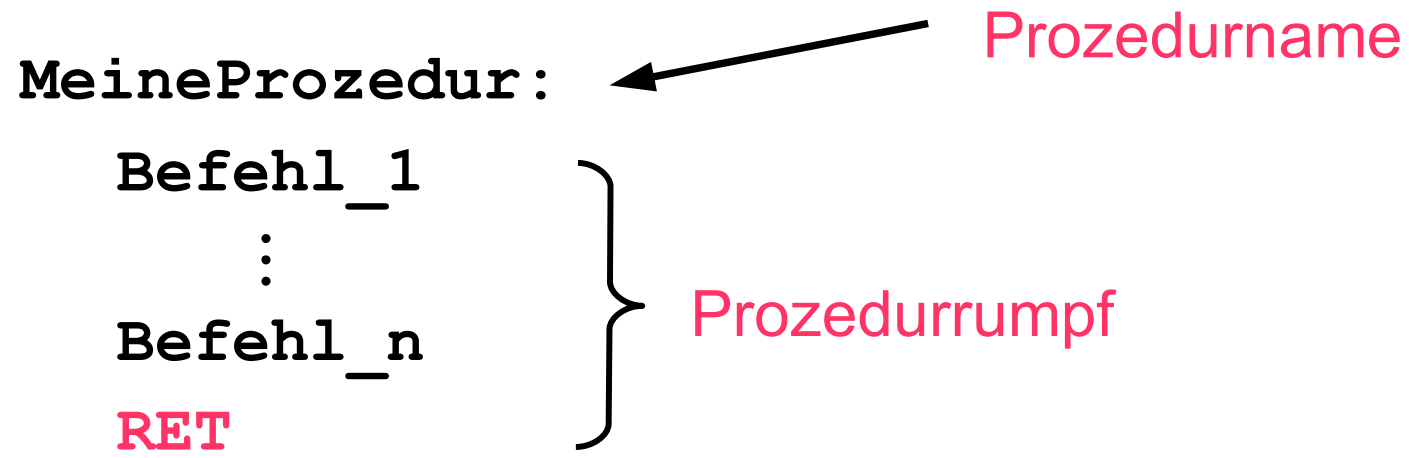
- Häufig wird an verschiedenen Stellen des Programms derselbe **Teilalgorithmus** benötigt
- Beispiele:
  - Polynomauswertung
  - Mathematische Funktion: z. B. sin(x)
- Mit den bisher eingeführten Assemblerbefehlen müssten diese Teilalgorithmen mehrfach im Code notiert werden
- Unnütze Schreibarbeit
- Unübersichtlicher Code
  - Schwierig zu warten: Bei Änderungen muss jede Kopie des Teilalgorithmus geändert werden

Unterprogramme anlegen und verwenden

- Mit Hilfe von Unterprogrammen lässt sich dieses Problem lösen
- Unterprogramme werden auch als **Prozeduren** (engl.: **procedure**) bezeichnet
- Bei der Verwendung von Prozeduren werden zwei verschiedene Phasen unterschieden:
  - Phase 1: Verbinden des Teilalgorithmus mit einem Namen
    - Sogenannte **Prozedurdeklaration**
  - Phase 2: Verwendung der Prozedur im Programm
    - **Prozeduraufruf** genannt

Prozeduren in x86-Assembler

- **Prozedurdeklaration**



- **Prozeduraufruf**

```
CALL MeineProzedur
```

Prozeduraufruf

- **`CALL MeineProzedur`**
  - Rettet den Befehlszählerinhalt (IP: instruction pointer)
    - Wird für den Rücksprung in das Programm nach Ende der Prozedur benötigt
  - Verzweigt zu der zum Prozedurnamen (Label) gehörenden Speicheradresse
    - Hier erster zu bearbeitender Befehl der Prozedur
- **`RET`**
  - Restauriert ursprünglichen Befehlszählerinhalt
  - Programm wird mit dem Befehl fortgesetzt, der auf das ursprüngliche **`CALL`** Kommando folgt

Polynomauswertung als Prozedur (1)

$p = ((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0$, mit: a_3=9, a_2=8, a_1=7, a_0=6 und x=3

```
; Hauptprogramm

; Befehle vor dem Prozeduraufruf
; ⋮
mov ax,3 ; x=3: Prozedur erwartet x in AX-Reg.

; Prozedurname ist "Polynom"
call Polynom ; springe in Prozedur "Polynom"

; Befehle nach dem Prozeduraufruf (bearbeitet
; nach RET in Prozedur)
; ⋮
```

Polynomauswertung als Prozedur (2)

$p = ((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0$, mit: a_3=9, a_2=8, a_1=7, a_0=6 und x=3

```
; Prozedur Polynom:
; Erwartet Wert für x in AX-Register

Polynom:
   mov bx,ax   ; x:=ax
   mov ax,9    ; Laden von a3
   mul bx      ; a3*x
   add ax,8    ; a3*x + a2
   mul bx      ; (a3*x + a2)*x
   add ax,7    ; (a3*x + a2)*x + a1
   mul bx      ; ((a3*x + a2)*x + a1)*x
   add ax,6    ; ((a3*x + a2)*x + a1)*x + a0 = p
               ; Resultat im AX-Register
   ret         ; Zurück zum Hauptprogramm
```

Seiteneffekte (1)

- Prozedur nutzt dieselben Register wie das aufrufende Programm
  - Erforderlich, da Anzahl der Register beschränkt
- Wichtige Registerinhalte werden dabei möglicherweise überschrieben
- Dies wird als **Seiteneffekt** bezeichnet
- Seiteneffekte vermeidbar durch Speichern der Registerinhalte der benötigten Register vor dem Aufruf der Prozedur

Seiteneffekte (2)

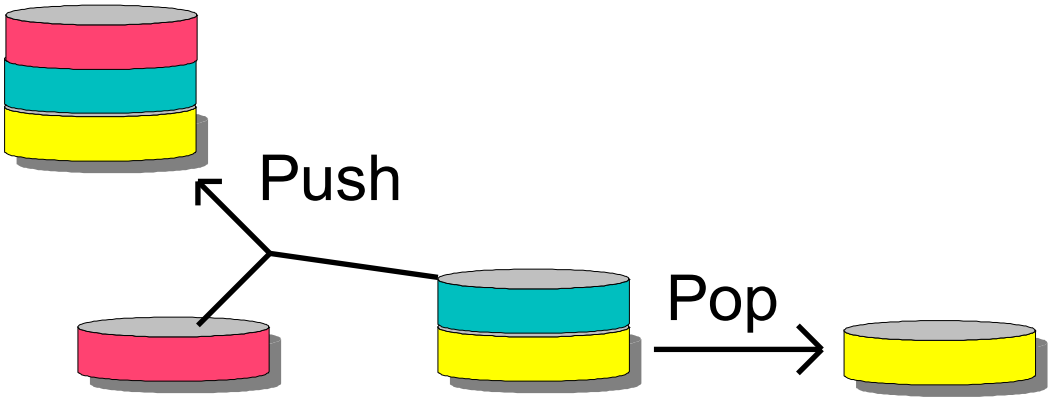
- Eine Möglichkeit: Speichern im Hauptspeicher an definierter Stelle
- Bequemer: **Stapel** (engl.: **stack**) benutzen
  - Vermeidet Adressrechnung
- Speichern auf Stapel ermöglicht sogenannten **Kontextwechsel**
  - Prozedur läuft in ihrem eigenen Kontext
  - Nach Abarbeitung der Prozedur wird ursprünglicher Kontext wiederhergestellt

Was ist der Stapel?

- Zum Programm gehörender, spezieller Speicherbereich, der das "Aufeinanderstapeln" von Binärwörtern erlaubt
  - Größe des Stapels, d. h. Anzahl der speicherbaren Binärwörter, üblicherweise vordefiniert
  - Kann aber auch vom Programmierer festgelegt werden

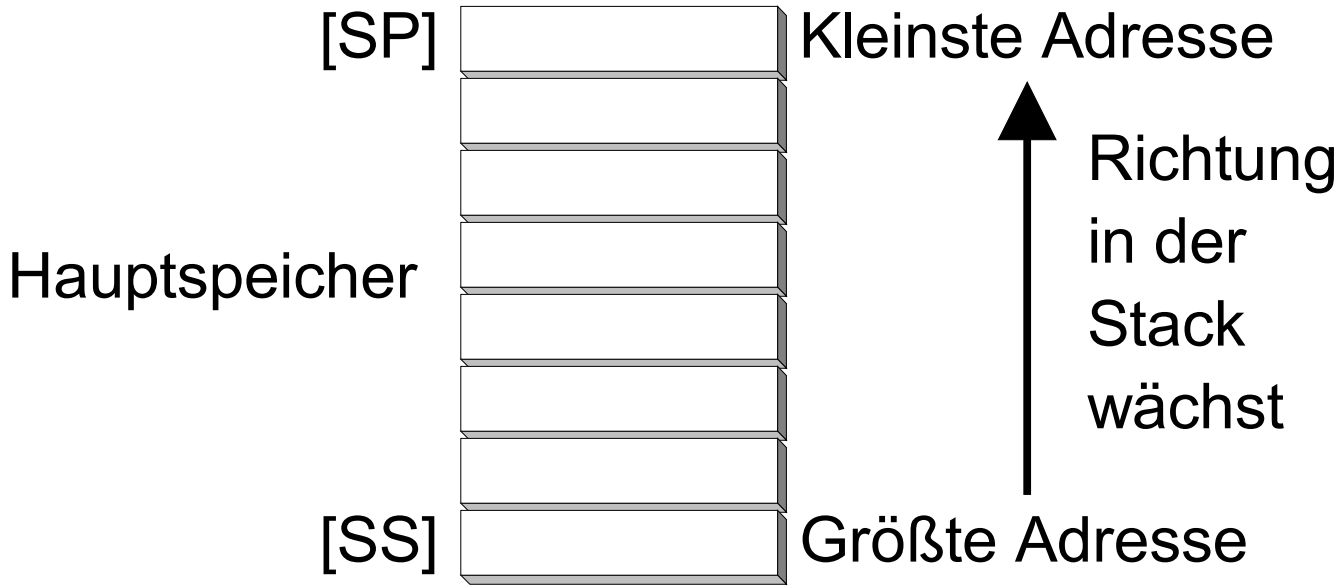
Stapeloperationen

- **Zwei elementare Operationen:**
  - **Push**
    - Legt ein neues Objekt oben auf den Stapel
  - **Pop**
    - Entnimmt das zuletzt auf den Stapel gelegte Element



Stapelregister

- **Stapel kontrolliert durch zwei Spezialregister:**
  - **Stapelsegmentregister** SS (engl.: *stack segment*)
    - Enthält Startadresse des Stapelsegments
  - **Stapelzeigerregister** SP (engl.: *stack pointer*)
    - Enthält Adresse des zuletzt auf den Stapel gelegten Elements



Stapel und x86 Assembler

- **PUSH**
  - Legt Inhalt von Register, Speicherwort oder Konstante auf den Stapel und dekrementiert Stack Pointer
  - Z. B. **`push ax, push [100], push 70`**
- **POP**
  - Entnimmt das zuletzt auf den Stapel gelegte Binärwort, lädt dieses in Register oder Speicher und inkrementiert Stack Pointer
  - Z. B. **`pop ax, pop [100]`**
- Jasmin: Registerwerte werden entsprechend der Registerbreite auf den Stapel gelegt bzw. von diesem gelesen, Zahlenwerte mit 32-Bit

Sicherung von Registerinhalten

```
; Prozedur Polynom:
; Erwartet Wert für x in AX-Register
; Rettet Wert von BX-Register auf Stapel

Polynom:
  push bx    ; sichere Inhalt von BX auf Stapel
  mov bx,ax  ; x:=ax
  mov ax,9   ; p=9: Initialisieren mit a3
  mul bx
    :        ; usw.
  add ax,6   ; Resultat im AX-Reg.
  pop bx     ; restauriere Inhalt von BX
  ret        ; Zurück zum Hauptprogramm
```

Stack und Prozeduraufruf

- Bisher ist nicht klar, wohin der Inhalt des Befehlszählers beim Prozeduraufruf gerettet wird
- **`CALL`** und **`RET`** nutzen den Stapel
- **`CALL:`**
  - Rettet den Inhalt des IP-Registers auf den Stapel und verzweigt zur Sprungmarke der Prozedur
- **`RET:`**
  - Nimmt die Rücksprungadresse vom Stapel und lädt diese in den Befehlszähler

Parametrisierte Prozeduren

Parametrisierte Prozeduren (1)

- **Bisher Eingabewert `x` in `AX`-Register erwartet**
  - **`AX`**-Register dient als Übergaberegister für **`x`**
- **`x` könnte mittels Stapel übergeben werden**
  - Dann Prozedur unabhängig von Annahmen hinsichtlich bestimmter, vorgegebener Übergaberegister

Parametrisierte Prozeduren (2)

- Bisher funktioniert Prozedur nur für die vorgegebenen Werte der Parameter $\mathtt{a}_0 \ldots \mathtt{a}_3$
- Was passiert, wenn andere Werte benötigt werden?
- Schlechte Lösung: Für jeden Parametersatz eigene Prozedur programmieren
- Besser: Auch diese Werte mittels Stapel als Eingabewerte an die Prozedur übergeben
- $\mathtt{a}_0 \ldots \mathtt{a}_3$ und **x** werden dann als **Prozedurparameter** bezeichnet
- Die Polynomauswertung ist dann eine **parametrisierte Prozedur**

Parametrisierte Prozeduren (3)

- **Vorteile:**
  - Beliebig viele Eingabewerte können übergeben werden
  - Position der Eingabewerte auf dem Stapel berechenbar
- Allgemein verwendbarer Mechanismus
- Parametrisierte Prozedur in Hochsprache:

```
Polynom(x,a0,a1,a2,a3);
```

  - Position der Parameter in der Liste legt Position auf dem Stapel fest

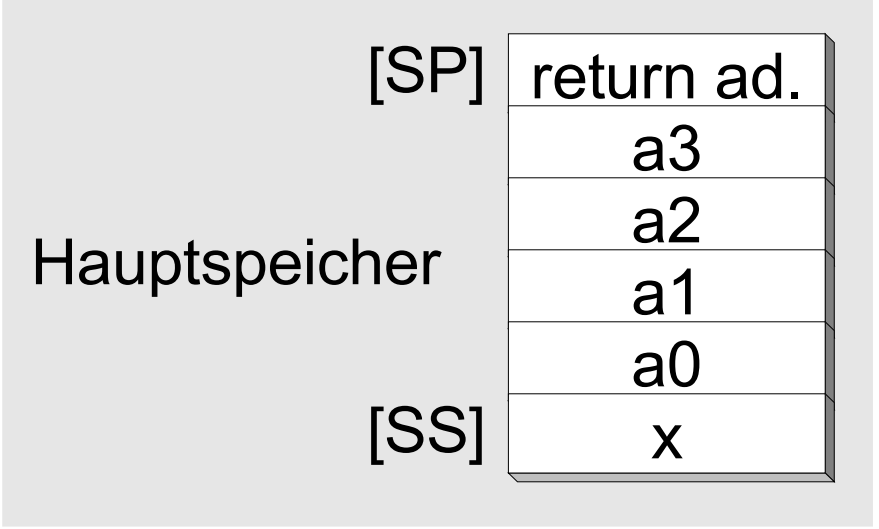
Funktionen

- Manche Prozeduren liefern einen Wert
- Man spricht dann von einer **Funktion**
- Beispiel: Polynomauswertung
  - Funktionswert (Rückgabewert) bisher in Register geliefert
    - Jetzt nicht mehr möglich
- Wie kann Funktionswert zurückgeliefert werden?
- Wieder mittels des Stapels!
- Parametrisierte Funktion in einer Hochsprache:

```
y = Polynom(x,a0,a1,a2,a3);
```

Parametrisierter Prozeduraufruf mittels Stapel (1)

- Alle Parameter werden vor dem Aufruf auf den Stapel gelegt
- Durch **`CALL`**-Befehl wird abschließend Rücksprungadresse auf die Parameterwerte gelegt
- Jasmin:
  - Zahlenwerte und Adressen werden als 32-Bit-Werte auf den Stapel gelegt
    - Jeder Wert nimmt also 4 Byte ein
  - Registerwerte hingegen entsprechend der Registerlänge
    - Bei AX-Register also z. B. 16-Bit, bei EAX-Register 32-Bit



Parametrisierter Prozeduraufruf mittels Stapel (2)

```
; Polynomauswertung mit Parameterübergabe durch
; Stapel

mov ax,1   ; Testwert in AX
mov bx,2   ; Testwert in BX

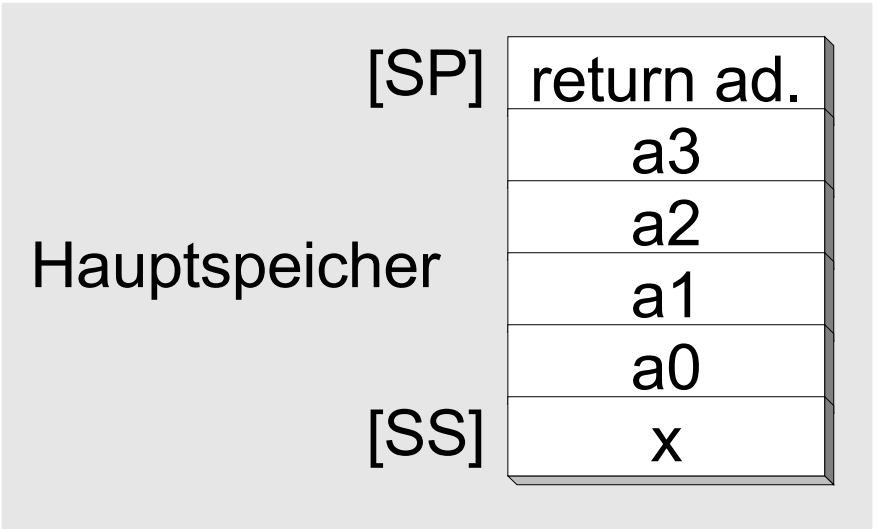
push 3     ; x  = 3 ; Jeweils 32-Bit-Wert
push 6     ; a0 = 6
push 7     ; a1 = 7
push 8     ; a2 = 8
push 9     ; a3 = 9

call Polynom ; Prozedurname ist "Polynom"

; Rückgabewert liegt auf dem Stapel
pop ax ; lies Wert nach AX
```

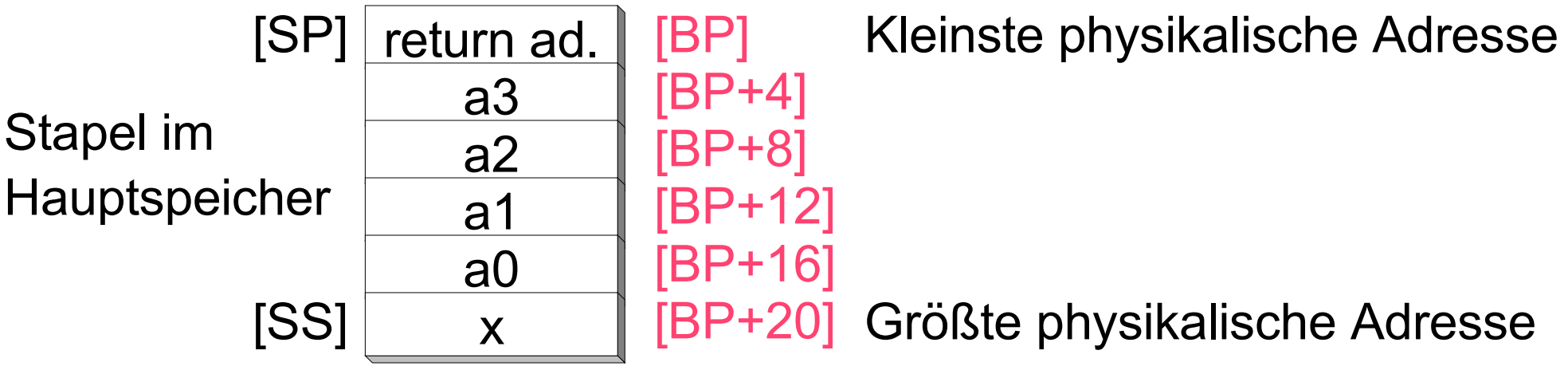
Parametrisierter Prozeduraufruf mittels Stapel (3)

- **Wie werden Parameterwerte in Prozedur angesprochen?**
- **Einfaches `POP` nicht sinnvoll**
  - Rücksprungadresse liegt zuoberst auf dem Stapel
    - Könnte verloren gehen!
    - Daten schützen, nicht "destruktiv" arbeiten
  - Parameterwerte werden häufig mehrmals benötigt
    - Zwischenspeicherung in Registern normalerweise nicht möglich, du zu wenige Register vorhanden
    - Zwischenspeicherung im Speicher verursacht unnötiges Umkopieren der Daten



BP-Register

- **BP: Base Pointer**
  - Spezialregister für den indizierten Stapelzugriff
    - Funktion vergleichbar mit der des BX-Registers für den indizierten Hauptspeicherzugriff
  - Ermöglicht "Hineinschauen" in den Stapel
  - BP Register wird mit dem Wert von SP initialisiert
    - SP hält Adresse des obersten Stapelelements



Parametrisierte Prozedur

```
; Prozedur Polynom: Parameterwerte auf Stapel
;                   Rückgabewert auf Stapel
Polynom:
  mov bp,sp        ; lade oberste Stapeladresse in BP
  push ax          ; sichere Inhalt von AX auf Stapel
  push bx          ; sichere Inhalt von BX auf Stapel
  mov bx,[bp+20]   ; bx := x
  mov ax,[bp+4]    ; AX mit a3 initialisieren
  mul bx
    ⋮              ; usw.
  add ax,[bp+16]   ; addiere a0
  mov [bp+20],ax   ; Resultat in AX auf Stapel retten
  pop bx           ; restauriere Inhalt von BX
  pop ax           ; restauriere Inhalt von AX
  ret 16           ; entferne 16 Byte bei Rücksprung
```

Entfernung von Parameterwerten

- Nach Rücksprung müssen Parameterwerte vom Stapel beseitigt werden
  - Anderenfalls würde Stapel mit jedem weiteren Prozeduraufruf immer weiter wachsen
- Eine Möglichkeit: Entsprechende Anzahl **`POP`**-Befehle im Hauptprogramm
- Bequemere Lösung:
- **`RET n`**
- Variante des **`RET`**-Befehls, die zusätzlich zur Rücksprungadresse *n* Bytes vom Stapel entfernt

Zeiger

Zeiger (1)

- Prozedurparameter kann auch Speicheradresse sein
- Dies wird als **Zeiger** (engl.: **pointer**) bezeichnet
- Zeiger ermöglichen indizierten Zugriff auf den Hauptspeicher aus der Prozedur heraus
- Anwendung:
  - Ablage der Prozedurparameter im Hauptspeicher
  - Übergabe der Parameteradressen mittels des Stapels an die Prozedur
  - Erlaubt das **Schreiben** von Werten in die Speicherzellen der Prozedurparameter

Zeiger (2)

- Liegen die Prozedurparameter nacheinander im Hauptspeicher, so reicht es, nur die Adresse des ersten Parameters an die Prozedur bzw. Funktion zu übergeben

- Die Adressen der übrigen Parameter können berechnet werden

Zeiger (3)

- **Werte auf dem Stapel**
  - Erlaubt nur Lesen der Parameterwerte
  - Werteparameter
  - "Call by value"

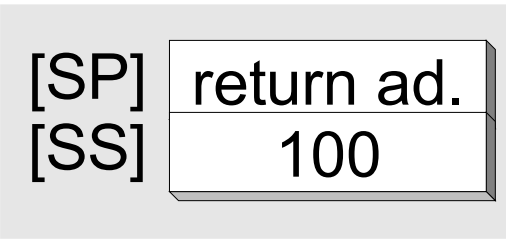
- **Adresse(n) auf dem Stapel**
  - Erlaubt Lesen UND Schreiben der Parameterwerte
  - Referenzparameter
  - "Call by reference"

Zeigerparameter (1)

```
; Polynomauswertung mit Zeigerparameter auf dem
; Stapel. Parameterwerte im Hauptspeicher.
mov ax,1   ; Testwert in AX
mov bx,2   ; Testwert in BX

mov [100],3    ; x  = 3: X ist Referenzparameter
mov [102],6    ; a0 = 6: a0 "                  "
mov [104],7    ; a1 = 7: a1 "                  "
mov [106],8    ; a2 = 8: a2 "                  "
mov [108],9    ; a3 = 9: a3 "                  "
push 100       ; Zeiger auf ersten Parameter

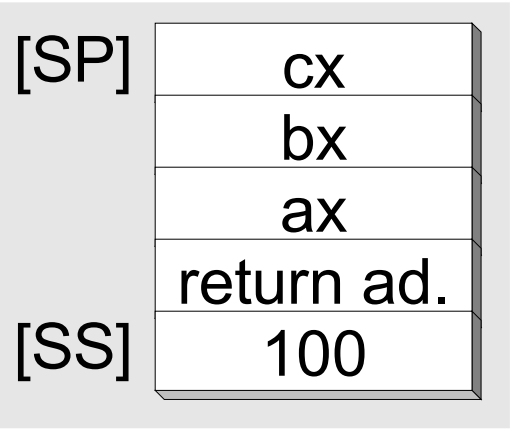
call Polynom   ; Polynom ändert
               ; jetzt Wert von x
mov ax,[100]   ; Hole Ergebnis aus x
 :             ; weitere Befehle...
```



Zeigerparameter (2)

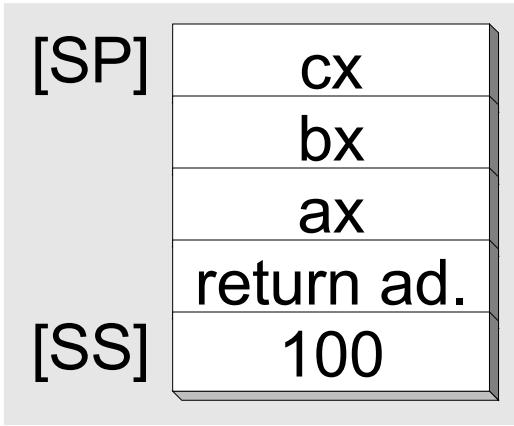
```
; Prozedur Polynom: Erwartet Zeiger auf
; Parameter auf dem Stapel

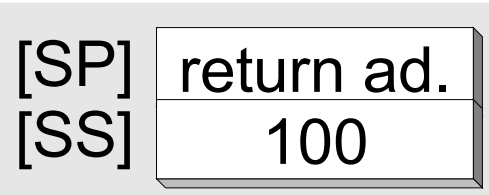
Polynom:
   mov bp,sp ; lade oberste Stapeladresse in BP
   push ax   ; sichere Inhalt von AX bis DX
   push bx   ; auf Stapel
   push cx
   push dx
   mov bx,[bp+4] ; lade Startadresse
   mov cx,[bx]   ; cx:=x
   mov ax,[bx+8] ; p mit a3 initial.
   mul cx
   ; Fortsetzung nächste Folie
```



Zeigerparameter (3)

```
add ax,[bx+6] ; addiere a2
mul cx
add ax,[bx+4] ; addiere a1
mul cx
add ax,[bx+2] ; addiere a0: Ergebnis in AX
mov [bx],ax   ; Ergebnis speichern in x
pop dx          ; restauriere Inhalt von DX bis AX
pop cx
pop bx
pop ax
ret 4           ; entferne zusätzlich 4 Byte
                ; bei Rücksprung
```





Rekursion

Rekursion

- Bei allen bisherigen Beispielen wurden Unterprogramme im Hauptprogramm aufgerufen
- Unterprogramme können auch in Unterprogrammen aufgerufen werden
  - D. h. Prozeduren und Funktionen können auch in Prozeduren und Funktionen benutzt werden
- Mehr noch: *Ein Unterprogramm kann sich selbst aufrufen!*
- Man spricht dann von einem *rekursiven Aufruf* dieses Unterprogramms
- Der zugehörende Algorithmus ist ein *rekursiver Algorithmus*

Beispiel

- Wir wollen wieder die Summe der natürlichen Zahlen von 1 bis *n* berechnen:
  - D. h. $s = 1 + 2 + 3 + ... + n$

- Für dieses Problem existieren unterschiedliche Lösungswege
  - Ein iterativer Lösungsweg
    - Diesen haben wir schon betrachtet
  - **Ein rekursiver Lösungsweg**
    - Diesen behandeln wir jetzt

Rekursive Lösung

- Die Lösung zu vielen Problemen lässt sich auch rekursiv formulieren
  - Häufig existiert sogar zunächst nur eine rekursive Lösung
- Bei der rekursiven Lösung wird die Lösung für den Wert *n* auf die Lösung für den Wert *n*-1 zurückgeführt
- Für unser Beispiel ergibt sich mit dieser Methode:
  - s(n) = s(n-1)+n
- Außerdem benötigen wir noch eine Aussage über den Anfang der Rekursion:
  - s(1) = 1

Beispiel für *n*=4

- Die Lösung ergibt sich durch sukzessives Einsetzen der Funktion *s* in die Funktion *s*:
- s(4) = s(3)+4
  - mit: s(3) = s(2) +3
- s(4) = (s(2) + 3) + 4
  - mit: s(2) = s(1) +2
- s(4) = ((s(1) + 2) + 3) + 4
  - mit s(1) = 1
- s(4) = ((1) + 2) + 3) + 4
- In Assembler umgesetzt bedeutet dies, dass mittels **`CALL`**-Anweisung die Funktion *s* in der Funktion *s* aufgerufen wird

Assembler

```
; Summe der Zahlen von
; 1 bis 4 mittels Rekursion
;
; Parameterwertübergabe
; und Rückgabe des Funktions-
; werts komplett mit AX
; Daher Kontextwechsel einfach
; Dies ist nur eine mögliche
; Form der Implementierung!

; Hauptprogramm:
mov ax,4  ; Summe 1 bis 4
call Sum  ; s hier Sum genannt
; Programmende
```

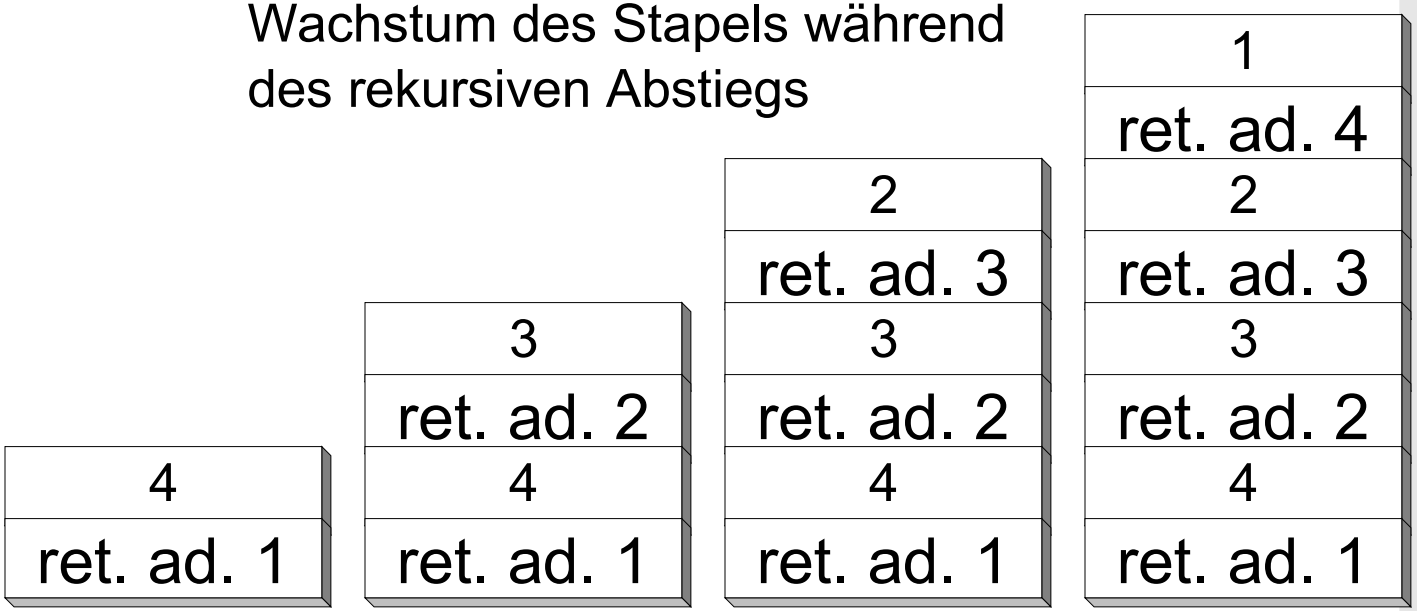
```
Sum:
  push ax
  dec ax
  cmp ax,0
  ja  Rekursion
  pop ax
  jmp Ende;
  Rekursion:
    call Sum
    mov bp,sp
    add [bp],ax
    pop ax
  Ende:
    ret
```

Ablauf

- Mit jedem Aufruf der Funktion `sum` wird der aktuelle Wert von *n* auf den Stapel gelegt und danach *n* dekrementiert
- Dies wird wiederholt, solange *n* > 0
  - **`cmp ax,0`**: Setzt Zero-Bit in Flag-Register, falls `ax-0=0`. Lässt Inhalt von **`ax`** unberührt
  - **`ja label`**: Sprung zum Label **`label`**, falls Zero-Bit *nicht* gesetzt ("Jump Above"), sonst Fortsetzung mit nächsten Befehl
  - **`jmp label`**: Unbedingter Sprung zum Label **`label`**
- Danach wird der Stapel sukzessive abgebaut
- Der jeweils herunter genommene Wert wird zur Summe addiert

Stapel: Rekursiver Abstieg

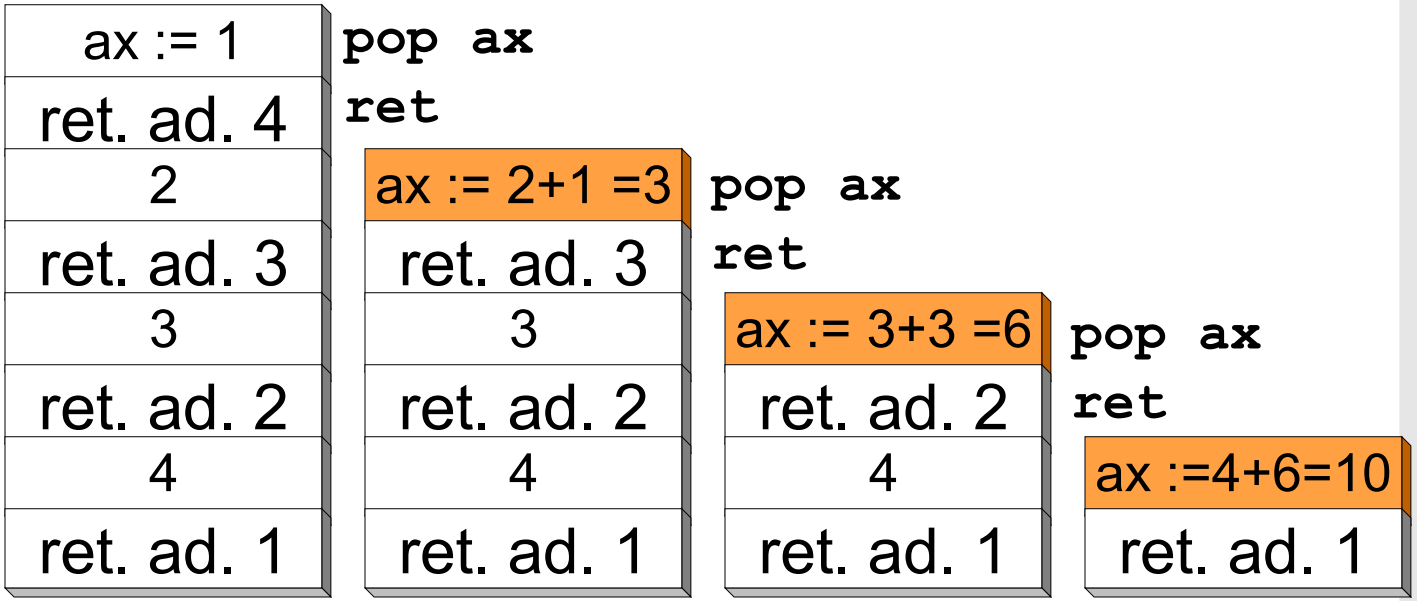
Wachstum des Stapels während des rekursiven Abstiegs




```
Sum:
   push ax
   dec ax
   cmp ax,0
   ja  Rekursion
   pop ax
   jmp Ende;
   Rekursion:
      call Sum
      mov bp,sp
      add [bp],ax
      pop ax
   Ende:
      ret
```

Stapel: Rekursiver Aufstieg

Abbau des Stapels während des rekursiven Aufstiegs



- Zwischensumme aus Register AX wird zum Wert der obersten Zelle im Stapel addiert und Resultat in Register AX geladen

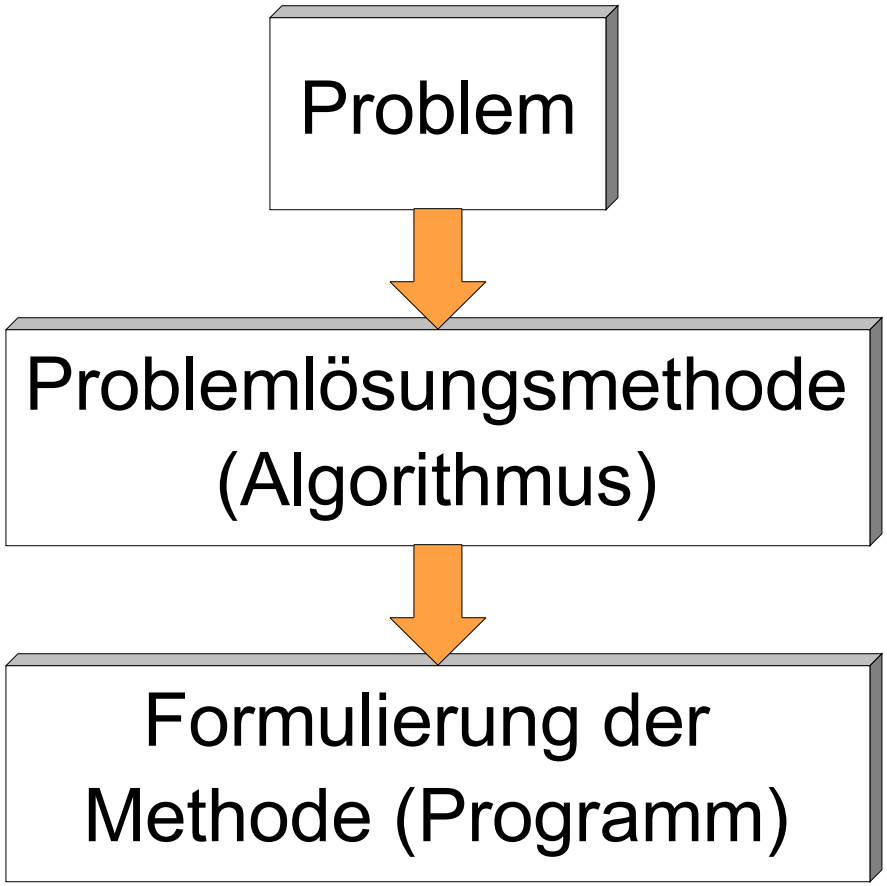
```
Sum:
   push ax
   dec ax
   cmp ax,0
   ja  Rekursion
   pop ax
   jmp Ende;
   Rekursion:
      call Sum
      mov bp,sp
      add [bp],ax
      pop ax
   Ende:
      ret
```

Aufwand

- Der **Kontextwechsel**, d. h. die Übergabe der Parameterwerte sowie das Sichern und Wiederherstellen der Registerwerte verursachen Operationen und Speicherzugriffe
- Rekursive Lösung können daher rechenaufwendiger als iterative oder – erst recht – geschlossene Lösungen sein!
- Sofern nur eine rekursive Lösung zu einem Problem existiert, ist es deshalb häufig sinnvoll, zur Beschleunigung eines Verfahrens nach einer iterativen Lösung zu suchen
- Darüber hinaus sind rekursive Lösungen häufig komplex und in ihrer Wirkung schwer vorstellbar
  - Potentielle Fehlerquelle!

Programmierung

Programmierung (1)



Programmierung

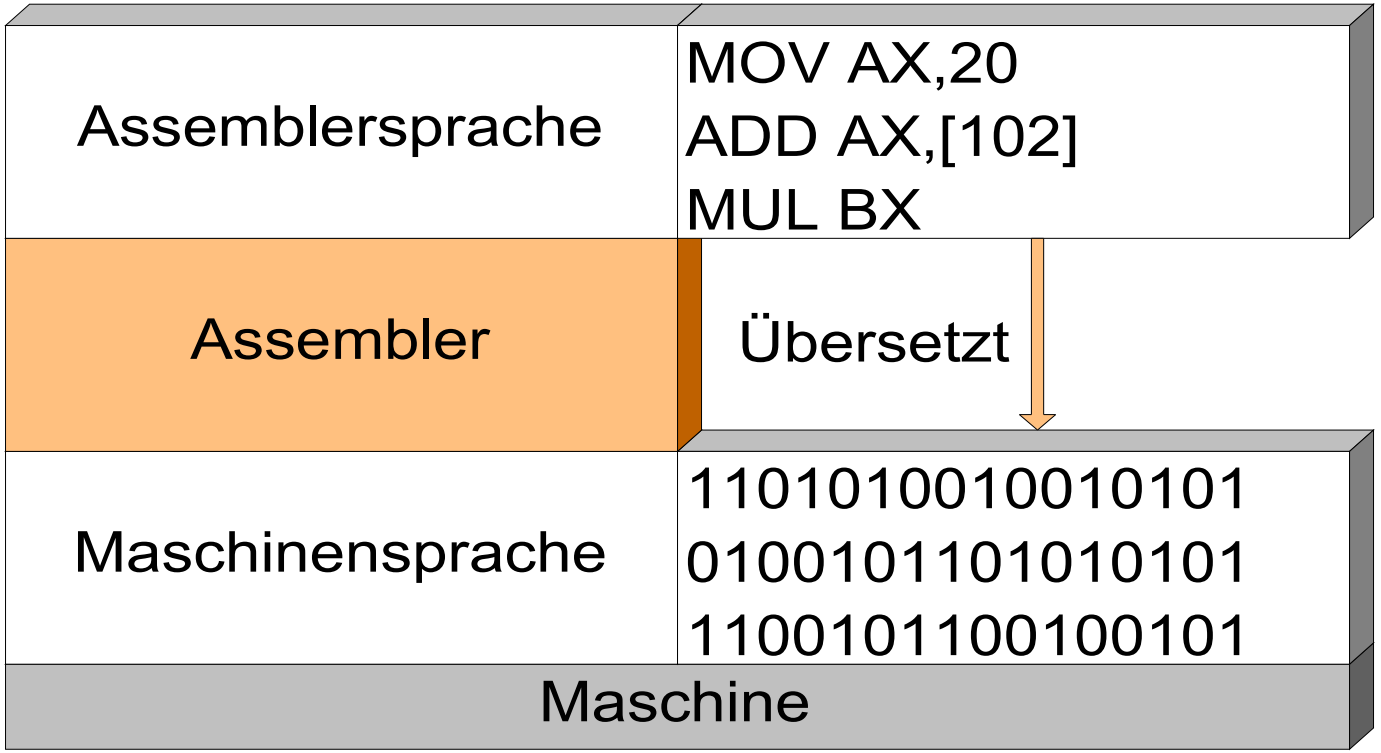
- Ein für den Menschen verständliches Programm besteht aus mit Hilfe von Buchstaben und Ziffern codierten Befehlen, Variablen, Konstanten oder Zahlen

- Ein für den Rechner verständliches Programm besteht aus Binärwörtern für die Operationen und Dualzahlen für die Werte der Operanden und Adressen

- Es ist daher erforderlich, dass für den Menschen verständliche Programm mittels **Codierung** in ein für die Maschine interpretierbares Programm zu übersetzen

Assemblerprogrammierung

- **Assemblerprogrammierung:**
  - Problemlösung wird in Assemblersprache formuliert
  - Assembler übersetzt Programm (Übersetzungseinheit, Modul) in Maschinensprache



Binder und Lader

- Binder (engl.: *linker*) generiert aus Maschinensprachemodulen ein ausführbares Programm (engl.: *executable*)
- Das Programm wird durch ein Ladeprogramm in den Hauptspeicher des Rechners geladen und zur Ausführung gebracht
  - Das Ladeprogramm ist Teil des Betriebssystems, z. B. Windows
  - Dateien mit Endung .exe oder .com werden automatisch durch Ladeprogramm gestartet

Assemblerprogrammierung (1)

- Berechnungen werden mittels Registerarithmetik durchgeführt
- Je nach Zahlenbereich werden dabei Register unterschiedlicher Breite eingesetzt
- z. B. AX: -32768 bis 32767 oder 0 bis 65535
- z. B. AL: -128 bis 127 oder 0 bis 255
- Die Kontrolle über die Wahl der korrekten Registerbreite obliegt dem Programmierer
  - Bei falscher Registerbreite ist Überlauf des Zahlenbereichs möglich

Assemblerprogrammierung (2)

- Beispiel Multiplikation
  - Verteilung des Ergebnisses über Register abhängig von Wortlänge der Operanden
  - Ergebnis z. B. in AX oder verteilt in AX und DX
- Beispiel Speicher
  - Anzahl der erforderlichen Speicherzellen zu je 1 Byte abhängig von der Wortlänge des zu speichernden Wertes

Assemblerprogrammierung (3)

- Es bleibt festzuhalten:
  - Die Interpretation und Verwaltung der Daten obliegt dem Programmierer
  - Die Interpretation wird durch die Daten im Flag-Register ermöglicht
    - Carry-Bit, Overflow-Bit, Sign-Bit etc.

Hochsprachenprogrammierung

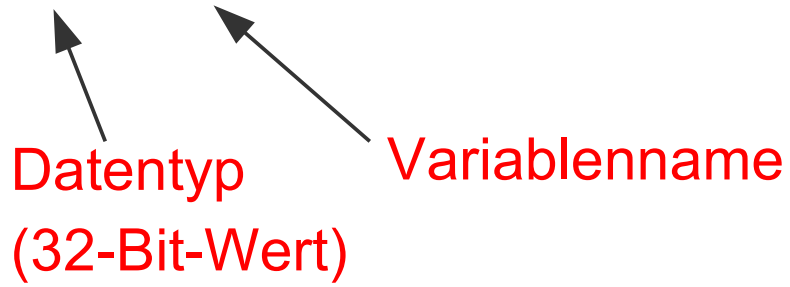
Hochsprache (1)

- Hochsprachen erlauben eine Abstraktion von Registerarithmetik und Speicherzellen
- Die Abstraktion wird durch das Einführen von **Datentypen** ermöglicht
- Diese gestatten eine automatische Auswertung von Operationsergebnissen
- Datentypen vereinfachen die Programmierung daher erheblich

Hochsprache (2)

- Die Abstraktion von Registern und Speicherzellen erfolgt durch Bereitstellung von **Variablen**
- Variablen besitzen einen Namen und einen Datentyp
- Beispiel:

```
int i;
```



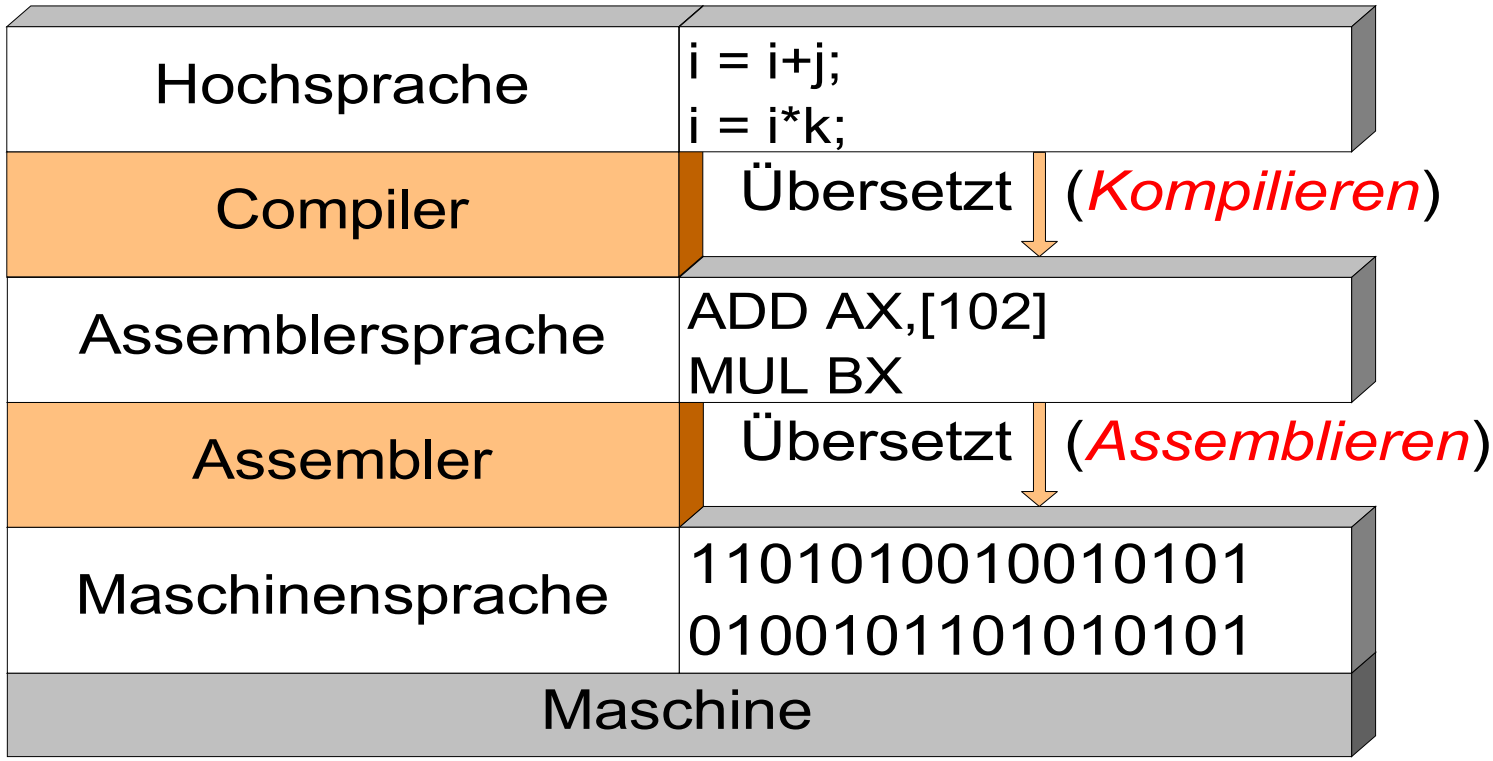
- Die Variable `i` wird hier als Container für einen ganzzahligen Wert vereinbart

Hochsprache (3)

- Während des Programmablaufes kann sich der Wert von `i` im Speicher oder in einem Register befinden
  - Der genaue Ort wird vom Übersetzungsprogramm (Compiler, s. u.) verwaltet
- Der Programmierer braucht sich hierum nicht zu kümmern

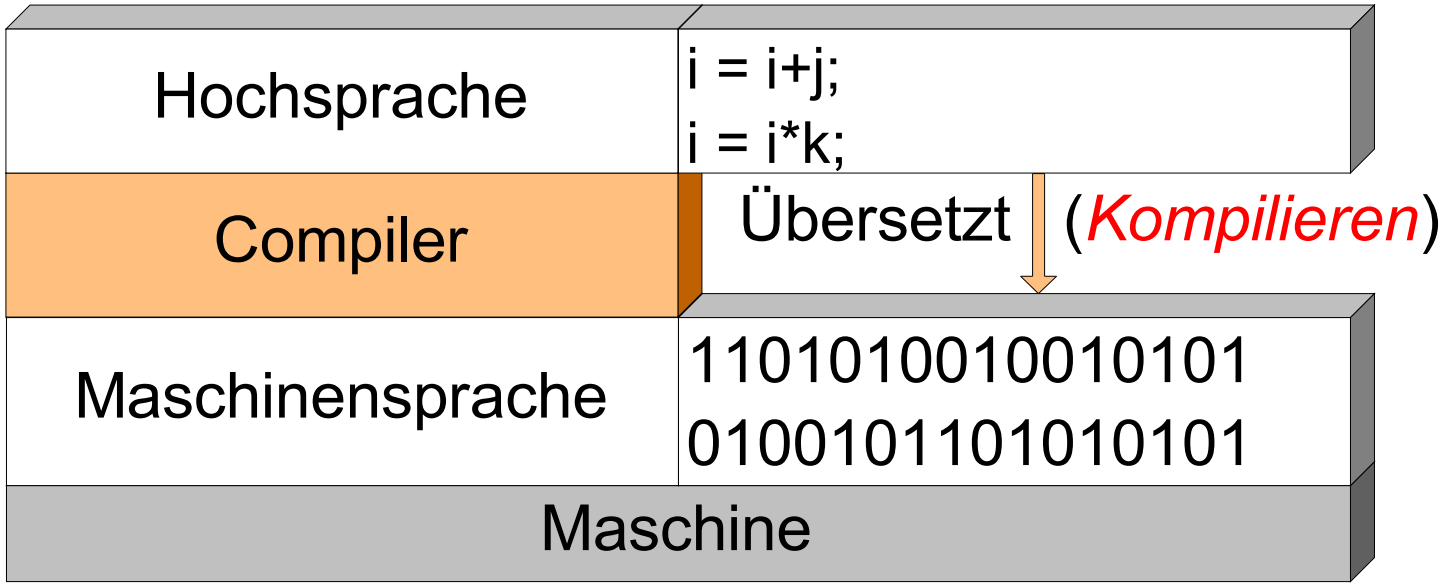
Hochsprachenprogrammierung (1)

- Jedes Hochsprachenprogramm ist zunächst eine reine Textdatei, die übersetzt werden muss
- Die Übersetzung erfolgt durch ein spezielles Programm: den **Compiler**
  - Der Compiler überprüft das Programm ebenfalls auf **syntaktische Korrektheit**



Hochsprachenprogrammierung (2)

- Die Kompilier- und Assemblierschritte werden heutzutage zumeist zu einem Schritt zusammengefasst
- Teilweise erfolgt die Kompilierung sogar transparent im Hintergrund einer Entwicklungsumgebung

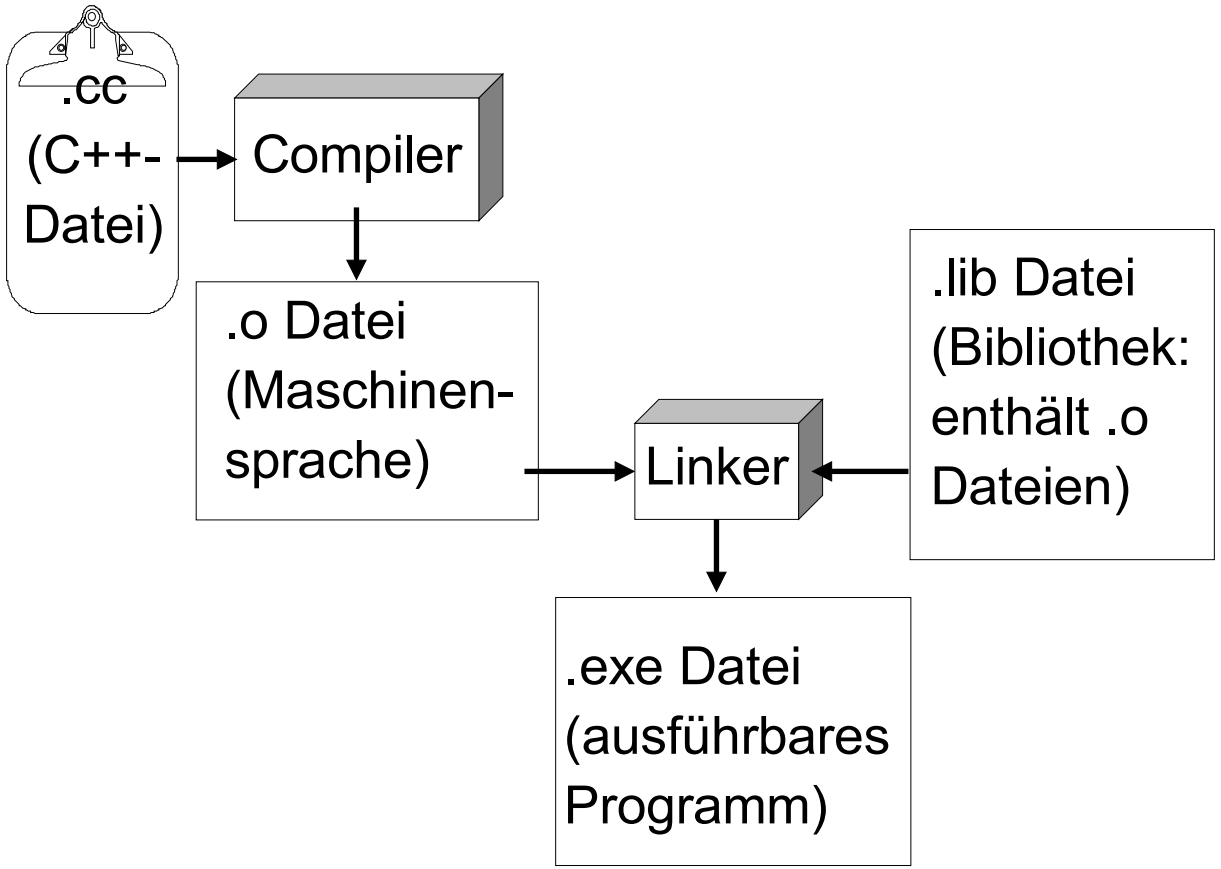


Hochsprachenprogrammierung (3)

- Compiler liefert Maschinensprachemodul, die sogenannte **Objektdatei**
  - Häufige Endungen: `.o`, `.obj`
- Programm benötigt i. d. R. Standardfunktionen
  - Z. B. Eingabe-Ausgabe-Funktionen
- Derartige Standardfunktionen werden in Bibliotheken vorgehalten
  - Endungen unter DOS, Windows: `.lib`, `.dll`
- Dies spart unnötige Übersetzungsvorgänge und das Vorhalten des zugehörenden Quellcodes

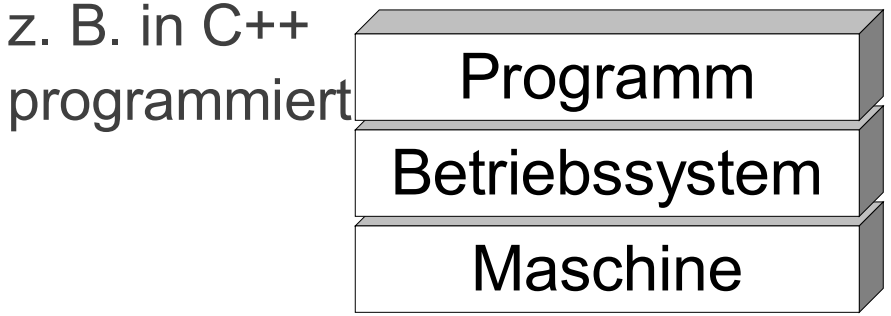
Hochsprachenprogrammierung (4)

- Die benötigten Bibliotheken werden durch ein weiteres spezielles Programm - den **Binder** (engl.: *Linker*) - mit den Objektdateien zu einem vollständigen Programm zusammengebunden
- Unter DOS, Windows:
  `.exe` Datei



Hochsprachenprogrammierung (5)

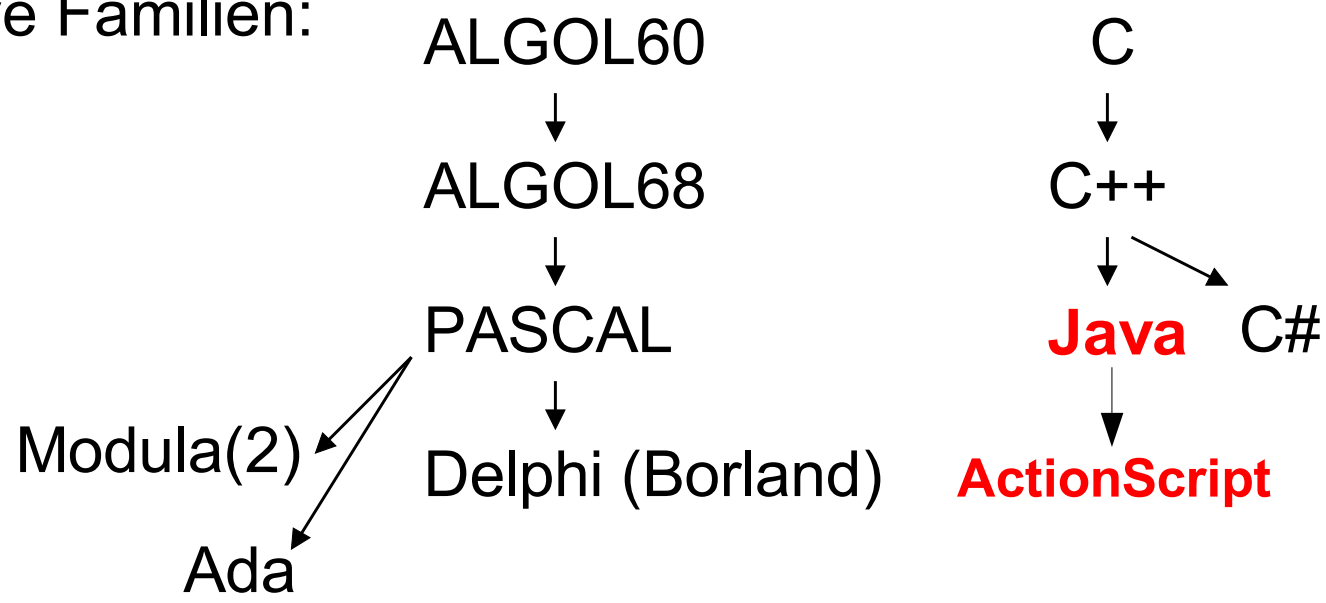
- Das Resultat des Übersetzungsvorganges ist ein Programm, welches wie jedes andere Computerprogramm auf dem Zielrechner ausgeführt werden kann



- Das Programm bekommt vom Betriebssystem Ressourcen (Speicher, Rechenzeit etc.) zugewiesen

Imperative Hochsprachen

- Imperativ ≙befehlsorientiert
- Wichtige imperative Familien:



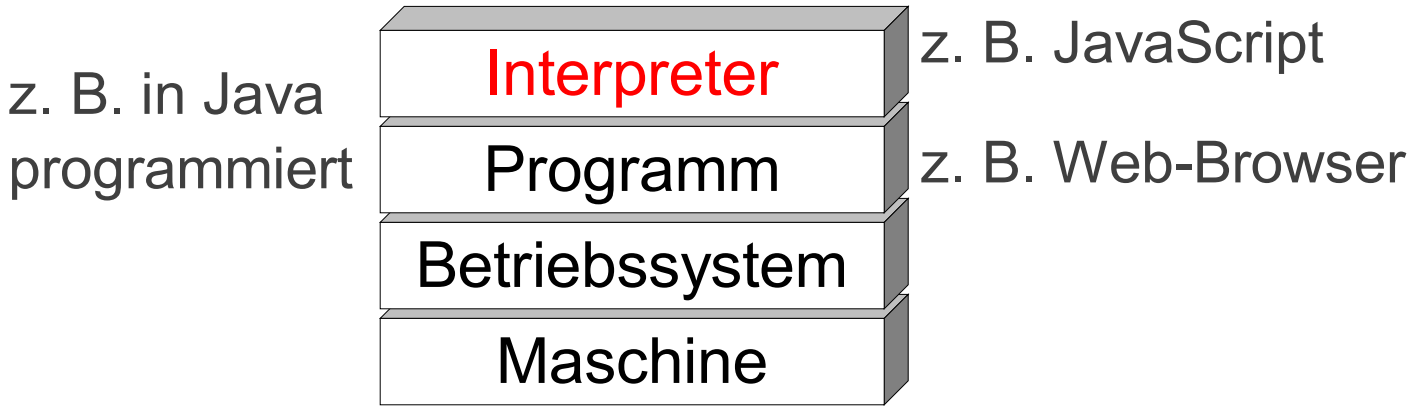
- Weitere imperative Sprachen: FORTRAN, BASIC, ...
- Andere Sprachfamilien:
  - Z. B. *deklarative* Programmiersprachen (bspw. Prolog)

Interpreter (1)

- Compiler erzeugen eigenständige Programme
  - Diese werden unabhängig vom Compiler ausgeführt
- Dies ist nicht die einzige Möglichkeit, Algorithmen vom Rechner ausführen zu lassen
- Alternative: Interpretation
  - Bei der Interpretation werden die Programmanweisungen von einem speziellen Programm einzeln in Maschinensprache übersetzt
  - Jede übersetzte Zeile wird direkt danach ausgeführt
  - Ein derartiges Programm wird **Interpreter** genannt

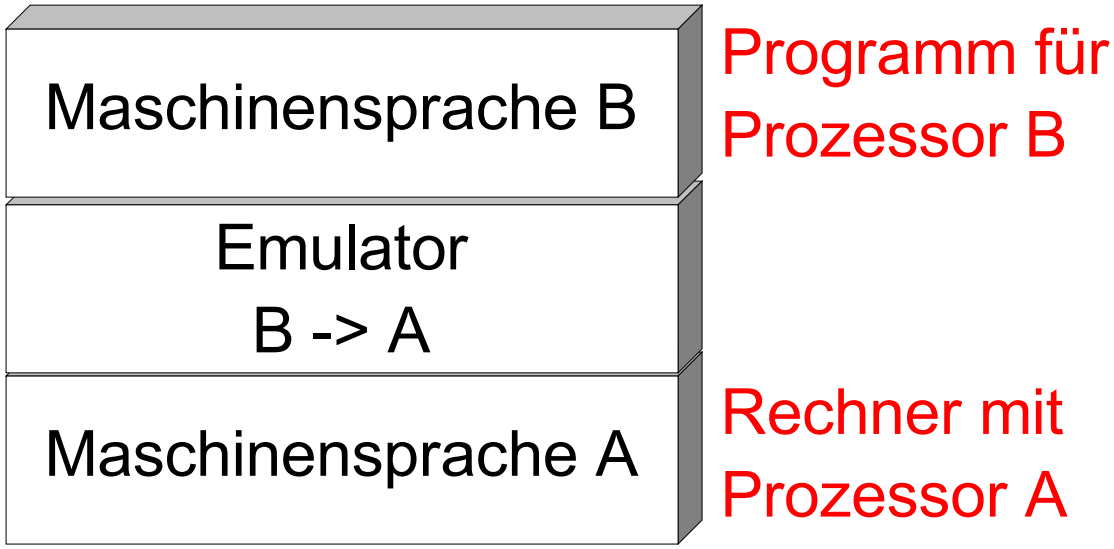
Interpreter (2)

- Beispiele für interpretierte Sprachen:
  - JavaScript, ActionScript 2, Perl, PHP etc.
- Interpreter kann Teil eines anderen Programms sein
  - Z. B. Web-Browser: Enthält Interpreter für JavaScript
  - ActionScript-Interpreter in Adobe Flash



Emulation (1)

- Interpreter lassen sich auch für die Übersetzung von Maschinensprachen einsetzen
- Interpreter bildet dann einen bestimmten Rechnertyp auf einem anderen Rechnertyp nach
- Dies wird als **Emulation** bezeichnet

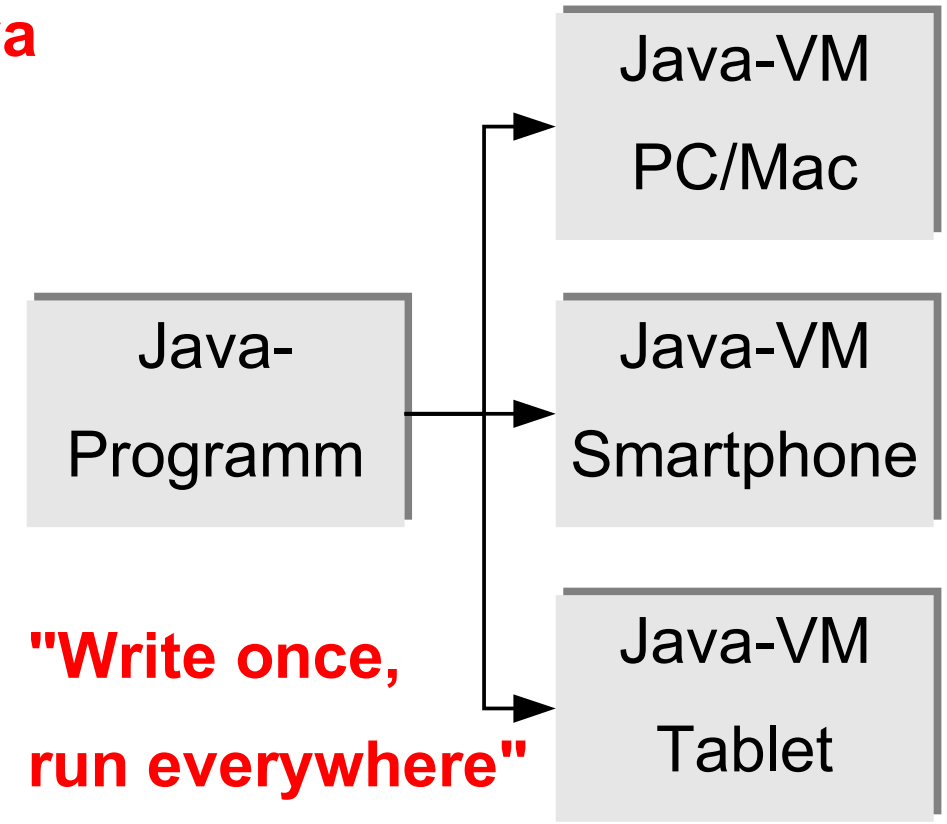


Emulation (2)

- Dient auch dazu, maschinenunabhängige Programmierung zu ermöglichen
- Programm wird für **virtuelle Maschine** entwickelt
- Virtuelle Maschine (VM) wird zumeist nicht in Hardware realisiert
  - Für alle interessierenden Rechnerplattformen werden Emulatoren dieser virtuellen Maschine in Software implementiert
- Programm wird dann der jeweiligen Software-VM zur Ausführung übergeben

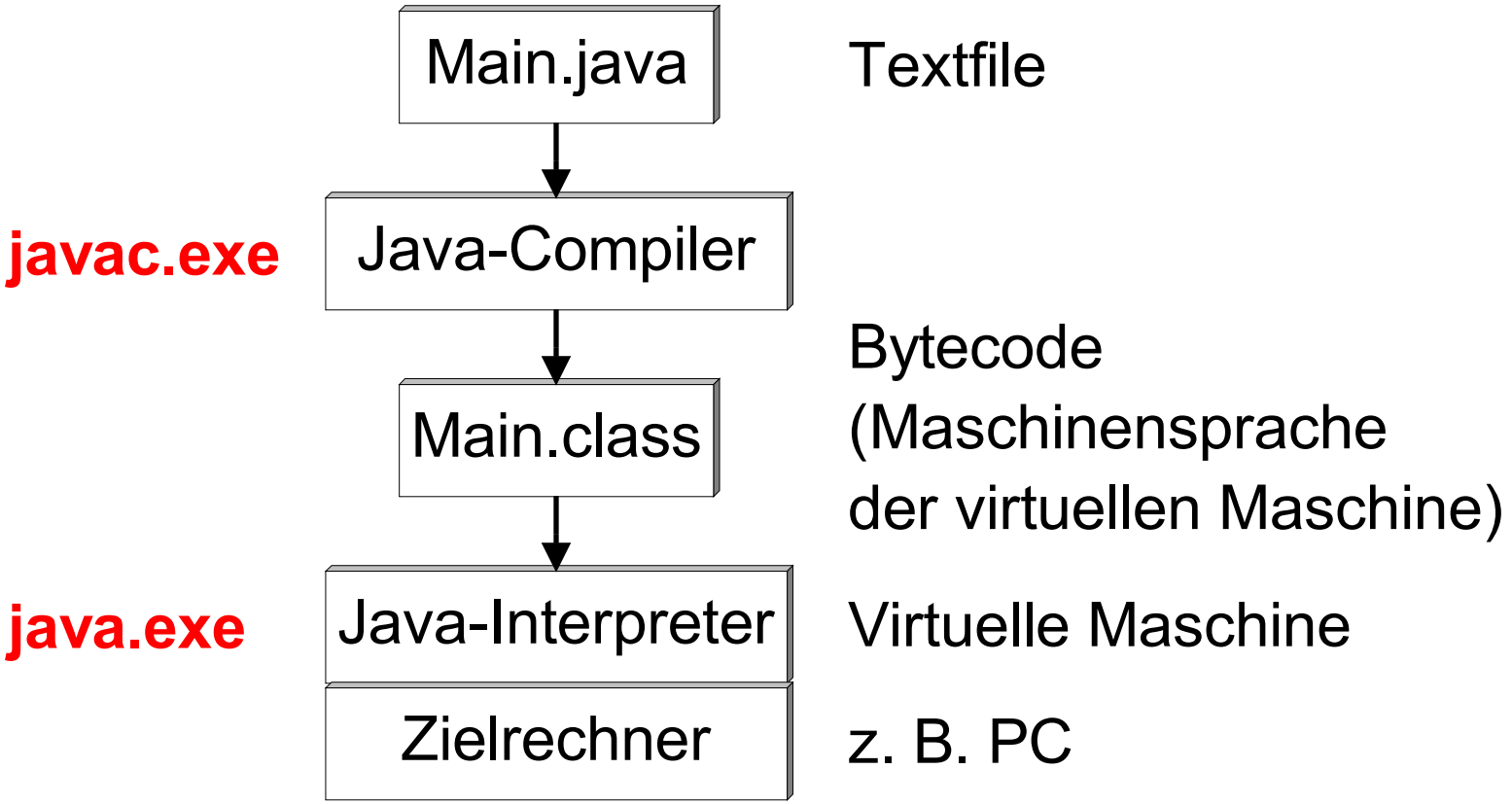
Emulation (3)

- Beispiel: **Java**



- Grundlage des Konzeptes: Effiziente Verteilung von Programmen über das Internet durch Plattformunabhängigkeit!

Java Virtual Machine



- Entsprechend bei ActionScript und Flash Player oder AIR-Laufzeitumgebung

Ein einfaches Java Programm

HelloWorld.java:

```
public class HelloWorld {

    // this is a comment

    public static void main(String[] args) {

        /*
            this is another comment
        */

        System.out.println("Hello World!");
    }
}
```

Ein einfaches Java Programm

HelloWorld.java:

```
public class HelloWorld {

    // this is a comment

    public static void main(String[] args) {

        /*
            this is another comment
        */

        System.out.println("Hello World!");
    }
}
```

- Programmname
- Entspricht Dateiname ohne `.java`

Ein einfaches Java Programm

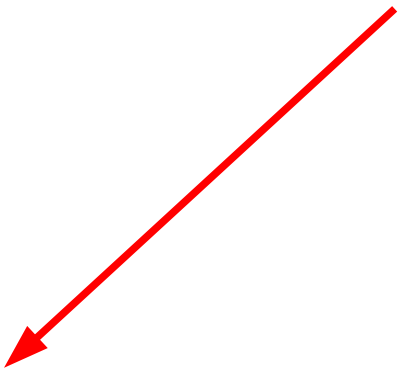
```
public class HelloWorld {

    // this is a comment

    public static void main(String[]

        /*
           this is another comment
        */

        System.out.println("Hello Worl
    }
}
```



- Dies bezeichnet ein **Unter-programm**
- In Java **Methode** genannt
- Die **`main`**-Methode wird automatisch ausgeführt, sobald das Programm gestartet wird

Ein einfaches Java Programm

```
public class HelloWorld {

    // this is a comment

    public static void main(String[]

        /*
           this is another comment
        */

        System.out.println("Hello World. ");
    }
}
```

- Diese Anweisung gehört zur Methode **`main`**
- Anweisungen werden in der Reihenfolge ihres Auftretens im Code ausgeführt

Ein einfaches Java Programm

```
public class HelloWorld {

    // this is a comment

    public static void main(String[]

        /*
           this is another comment
        */

        System.out.println("Hello World!");
    }
}
```

- Anweisungen werden durch Semikolons voneinander getrennt
- Das Semikolon gibt an, wo eine Anweisung endet und die nächste beginnt

Ein einfaches Java Programm

- Anweisungen werden mittels geschweifter Klammern **{ }** gruppiert
- Die Anweisungen zwischen diesen Klammern gehören zur Methode **`main`**

```
public static void main(String[] args) {

    /*
       this is another comment
    */

    System.out.println("Hello World!");
}
}
```

Ein einfaches Java Programm

```
public class HelloWorld {

    // this is a comment

    public static void main(String[] args) {

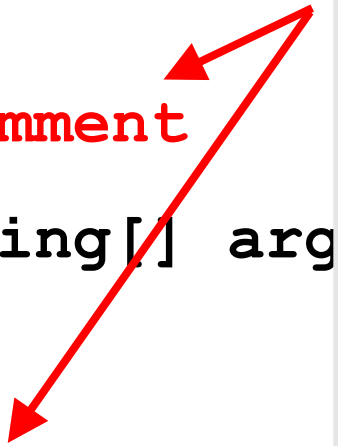
        /*
            this is another comment
        */

        System.out.println("Hello World!");
    }
}
```

- Die Anweisungen zwischen diesen Klammern gehören zum Programm **`HelloWorld`**

Ein einfaches Java Programm

```
public class HelloWorld {
  // this is a single-line comment
  public static void main(String[] arg
    /*
      this is another comment
    */
    System.out.println("Hello World!")
  }
}
```



- Diese Zeilen sind **Kommentare**
- Der Compiler ignoriert diese Zeilen
- Kommentare werden benutzt, um den Code näher zu erläutern. Sie stellen keine Anweisungen dar!

Warum Kommentare?

- Erläuterungen für sich selbst und für andere
  - Der/die Entwickler(in) muss den Code auch nach einem halben Jahr noch verstehen können
  - Kollege, der den Code ggf. weiterentwickelt, muss diesen ebenfalls verstehen können
- Kommentare erläutern schwierige Codeabschnitte
  - Sie liefern die zum Verständnis erforderliche Hintergrundinformation
- Ein falscher, nicht mehr aktueller oder irreführender Kommentar ist schlechter als gar kein Kommentar!

Einrückungen

- Compiler übersetzt Code wortweise

```
public class              HelloWorld {
   // this              is a comment
            public static void
  main(String[] args)
                     {/*
          this
 is another    comment */ System.ou
"Hello World!");   }}
```

- Für den Compiler ist dieser Code genau so gut lesbar, wie der vorherige
- Einrückungen dienen dazu, Programmcode für den Menschen lesbarer zu machen!

Ein einfaches Java Programm

```
public class HelloWorld {

  // this is a single-line c

  public static void main(St

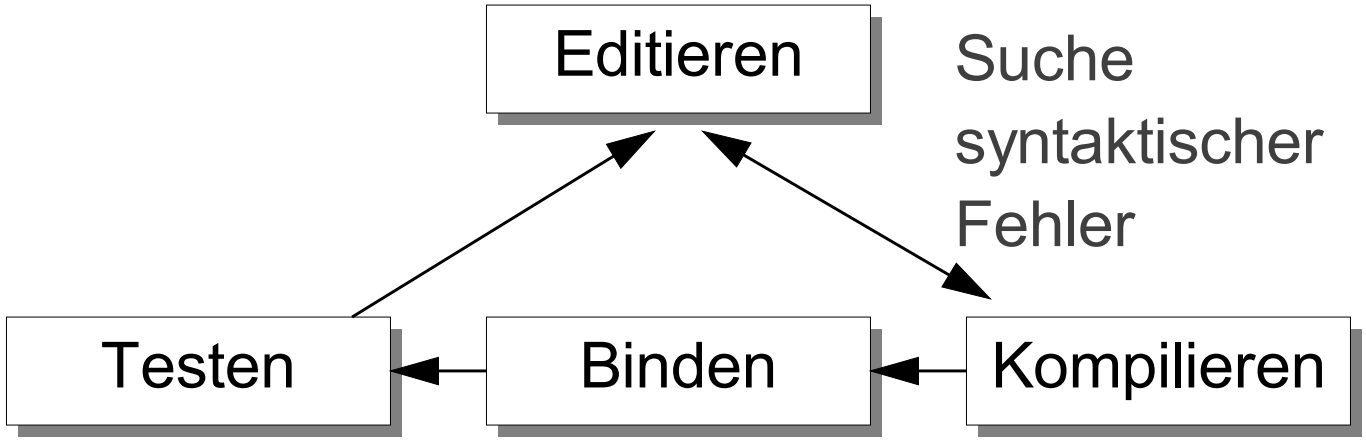
    /*
      this is another commen
    */

    System.out.println("Hello World!");
  }
}
```

- Dies ist ein Ausgabebefehl
- Mittels dieser Anweisung kann eine Zeichenkette auf der Konsole ausgegeben werden
- Zeichenketten werden von Anführungszeichen eingeschlossen
- Die Zeichenkette lautet hier: *Hello World!*

Entwicklungszyklus

- Während der Programmentwicklung wird der Programmtext solange editiert, bis sich dieser fehlerfrei kompilieren lässt und das Programm fehlerfrei läuft



Datentypen

Datentypen (1)

- Programme verarbeiten Daten
- Daten können unterschiedlichen Klassen bzw. Typen angehören
  - Wahrheitswerte, Zahlen, Zeichenketten, Bilder, Musikstücke, Videos etc.
- Im Computer werden alle diese Daten in derselben Weise gespeichert: als **Binärwörter**
- Datentypen erlauben die Klassifikation von Daten
- Datentypen legen fest, wie Daten zu interpretieren sind
  - Sie legen die Eigenschaften von Daten einer Klasse fest

Datentypen (2)

- Das Binärwort 1111 1111 1001 1001 kann unterschiedlich interpretiert werden:
  - Als ganze Zahl: -103
  - Als natürliche Zahl: 65433
  - Als Zeichenkette bestehend aus zwei Zeichen:
    - 1111 1111 = 255 = 'ÿ'
    - 1001 1001 = 153 = '™'
  - Als Teil einer Folge von Binärwörtern, die zu einem langen Datenblock, z. B. einem Video, gehören
  - etc.

Datentypen (3)

- Der Typ eines Datenwerts definiert:
  - Die erforderliche Menge Speicherplatz für die Speicherung dieses Werts
  - Den Wertebereich der Datenwerte
  - Die Menge der erlaubten Operationen, die mit Daten dieses Typs durchführbar sind
    - Bilder können z. B. komprimiert werden, aber keine Zahlen

Datentypen (4)

- Assemblerprogrammierung:
  - Der Programmierer muss selber wissen, wie ein Datenwert zu interpretieren ist
- Hochsprache:
  - Der Programmierer **definiert** durch Zuweisen eines Datentyps, wie ein Datenwert zu interpretieren ist
  - Die Maschine
    - übernimmt die Interpretation des Datenwerts
    - überprüft den korrekten Gebrauch des Datenwerts
  - Auf diese Weise nimmt die Maschine dem Programmierer Arbeit ab

Arten von Datentypen

- Die Menge der Datentypen kann in zwei verschiedene Gruppen aufgeteilt werden:
  - **Primitive Datentypen bzw. Grundtypen**
    - In der jeweiligen Programmiersprache vordefinierte Standardtypen
    - Nah an der Hardware definiert
    - Besitzen eine festgelegte Anzahl Bytes
  - **Abstrakte Datentypen**
    - Erlauben es eigene, ggf. komplexe Datentypen zu definieren
    - Dies erfolgt auf der Grundlage primitiver Datentypen

Primitive Datentypen

Wahrheitswert

- Typ: **`boolean`**
- Einfachster Datentyp
- Kennt nur die zwei Wahrheitswerte **`true`** und **`false`**
- Verknüpfung über logische Operationen
  - Und, Oder, Negation

Datentypen für ganze Zahlen (1)

- Engl.: *Integer number*
- Allgemein gilt: Keine Nachkommastellen möglich!
- Zahlen der Länge 8-Bit: **`byte`**
  - -128 bis 127
  - Entspricht einer Speicherzelle bzw. AH, AL etc.
- Zahlen der Länge 16-Bit: **`short`**
  - -32,768 bis +32,767
  - Entspricht zwei Speicherzellen bzw. AX, BX etc.
- Für alle Typnamen gilt:
  - Groß- und Kleinbuchstaben werden in Java unterschieden

Datentypen für ganze Zahlen (2)

- Zahlen der Länge 32-Bit: `int`
  - -2^{31} bis $2^{31}-1$
  - Entspricht vier Speicherzellen bzw. EAX, EBX etc.
- Zahlen der Länge 64-Bit: `long`
  - -2^{63} bis $2^{63}-1$
  - Entspricht acht Speicherzellen bzw. RAX, RBX etc.

Datentypen für ganze Zahlen (3)

- Wahl zwischen **`byte`**, **`short`**, **`int`**, **`long`** abhängig vom darzustellenden Zahlenbereich und von der Rechnerarchitektur
  - Bei einfachen Berechnungen sollte Datentyp mit Registerbreite der Rechnerarchitektur korrespondieren
  - Möglichst "kleiner" Datentyp kann bei großen Datenmengen, wie z. B. bei Videos, wichtig sein

Gleitkommazahlen (1)

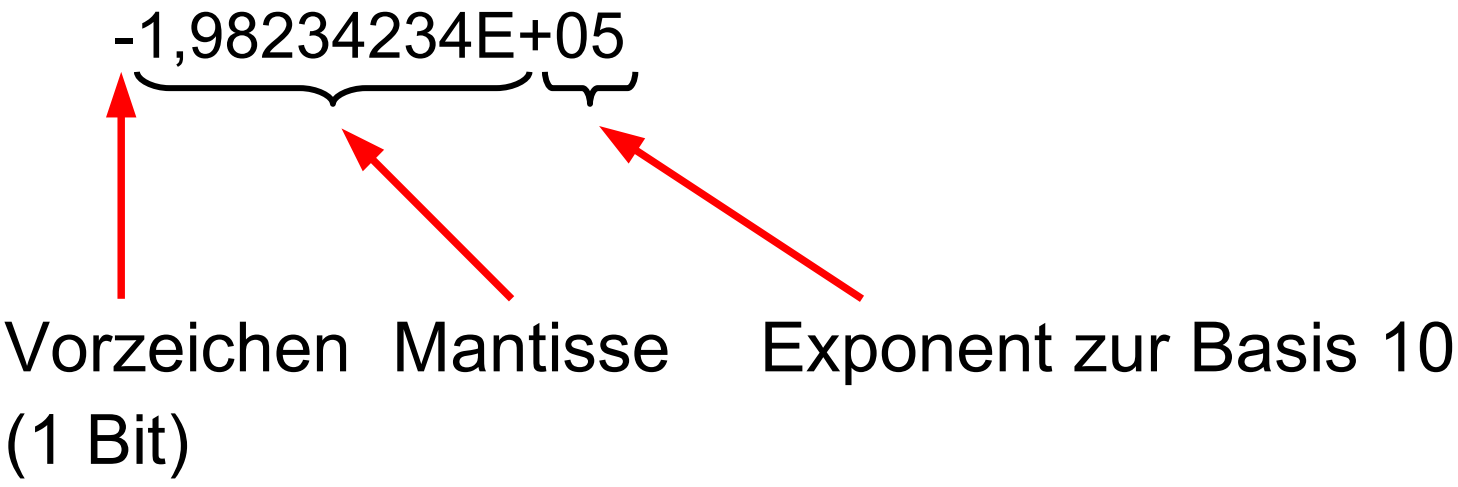
- Reelle Zahlen im Computer durch Binärwörter dargestellt
- Folglich ist nur eine feste Anzahl von Stellen vorhanden
- Wie lassen sich reelle Zahlen zweckmäßig durch Binärwörter darstellen?
- Eine Möglichkeit: Für die Nachkommastellen wird eine feste Anzahl Stellen reserviert, der Rest für die Stellen vor dem Komma
- Dies wird als **Festkommazahl** bezeichnet
- **Problem**: Für einige Anwendungen ist die Anzahl der Nachkommastellen unnötig groß (z. B. Astronomie), für andere Anwendungen die Genauigkeit zu gering (z. B. Chip-Layout)

Gleitkommazahlen (2)

- Deshalb ist eine Darstellung zweckmäßiger, bei der sich die Genauigkeit an der Größe des Zahlenwerts orientiert
- Die Genauigkeit ist dann bei kleinen Zahlen groß und bei großen Zahlen klein
- Die Gleitkommadarstellung erfüllt diese Bedingung

Gleitkommazahlen (3)

- Eine Gleitkommazahl besteht aus drei Teilen:



- Im Computer werden Gleitkommazahlen als Dualzahlen dargestellt

Gleitkommazahlen (4)

- Jede Gleitkommazahl kann in eine Gleitkommazahl mit einer Stelle vor dem Komma umgewandelt werden
  - Beispiel: $384 \times 10^6 = 3{,}84 \times 10^8$
- Dies wird als **normierte Gleitkommazahl** bezeichnet
- Dualzahldarstellung:
- $3{,}84 \times 10^8{}_{10} = 1{,}01101111000011011E11100_2$
  - Dezimalzahl: Exponent zur Basis 10
  - Dualzahl: Exponent zur Basis 2

Gleitkommazahlen (5)

- Die normierte Gleitkommazahl $\pm 1, m_1 m_2 m_3 \ldots m_n \cdot 2^E$ stellt den Zahlenwert

$$(-1)^v \cdot (1 + m_1 \cdot 2^{-1} + \ldots + m_n \cdot 2^{-n}) \cdot 2^E$$

mit:
v : Vorzeichen
$m_1 \ldots m_n$: Mantisse
E : Exponent

dar

Gleitkommazahlen (6)

- Im Rechner müssen Gleitkommazahlen mit Binärwörtern fester Länge dargestellt werden
  - 32-Bit, 64-Bit
- Wie werden Bits auf Vorzeichen, Mantisse und Exponent verteilt?
- IEEE Short Real Format für 32-Bit Zahlen
  - Vorzeichen: 1 Bit, Mantisse: 23 Bit, Exponent: 8 Bit
- IEEE Long Real Format für 64-Bit Zahlen
  - Vorzeichen: 1 Bit, Mantisse: 52 Bit, Exponent: 11 Bit
- (IEEE: *Institute of Electrical and Electronics Engineers*)

Datentypen für Gleitkommazahlen

- Engl.: *Floating point number*
- Zahlen der Länge 32-Bit: **`float`**
  - ±1.401e-45 bis +3.403e+38
- Zahlen der Länge 64-Bit: **`double`**
  - ±4.941e-324 bis 1.798e+308
- Gleitkommazahlen werden mit Punkt geschrieben
- Beispiele:
  - 123.0 -123.5 -198234.234 0.00000381
  - 1.23e+02 -1.235e+02 -1.98234234e+05 3.81e-06
    - e: Exponent zur Basis 10

Exponentialdarstellung: Übung

- Schreiben Sie die folgenden Zahlen aus, d. h. ohne Exponentialdarstellung:

- 9.15E+03 -7.01E+02 3.1415927E+05 1.51E-04

- Der Exponent bezeichnet die Anzahl der Stellen, um die das Komma nach links oder rechts zu verschieben ist
  - Positiver Exponent: Komma nach rechts verschieben
  - Negativer Exponent: Komma nach links verschieben

Gleitkommazahlen: Umwandlung

- Bei Verknüpfungen zwischen ganzen und Gleitkommazahlen werden ganze Zahlen automatisch in Gleitkommazahlen umgewandelt

- Für die Umwandlung von Gleitkommazahlen in ganze Zahlen stehen in Java Rundungsfunktionen zur Verfügung:

  - **`round, floor, ceil`**

Gleitkommazahlen: Genauigkeit (1)

- Die Genauigkeit von Gleitkommazahlen ist beschränkt, da die Anzahl der Stellen endlich ist
  - Bei Umwandlungen treten Ungenauigkeiten auf
- Während längerer Berechnungen können sich Ungenauigkeiten zu signifikanten Fehlern aufschaukeln!
- Vorsicht auch bei Vergleichen für Programmverzweigungen
  - Test auf Gleichheit vermeiden!

Gleitkommazahlen: Genauigkeit (2)

- Beispiele:
  - 87345600...0 an Stelle von 87345621...3
  - 1/3 als Dezimalzahl: 0.33333333333333333....
  - Pi: 3,141592..., e: 1,414213...
    - Für die exakte Darstellung ist eine unendliche Anzahl von Nachkommastellen erforderlich
  - 0,1 z. B. wegen beschränkter Stellenzahl der Dualmantisse nicht exakt darstellbar
    - Wird durch 0,0999999940 bzw. 0,1000000015 angenähert

Schriftzeichen

- Typ: **`char`**
- Dient zum Speichern einzelner Schriftzeichen
  - In Java wird 16-Bit Unicode Zeichensatz verwendet
- Werte vom Typ **`char`** werden in der Form '**`x`**' notiert
  - D. h. Buchstaben werden von einfachen Anführungszeichen eingefasst
- Es gibt drei Arten von **`char`**-Werten:
  - Einfache Zeichen
  - So genannte Escape-Sequenzen für Steuerzeichen
  - Unicode-Steuersequenzen

Escape-Sequenzen

`\b`: Rückschritt

`\t`: Tabulator

`\n`: Zeilenvorschub

`\f`: Seitenvorschub

`\r`: Zeilenrücklauf

`\"`: Doppelter Anführungsstrich

`\'`: Einfacher Anführungsstrich

`\\`: Backslash

`\uxxxx`: Steuersequenz für Abruf von Zeichen aus dem Unicode-Zeichensatz mittels vierstelliger Hexadezimalzahl

Zeichenketten

- Setzen sich aus einer Folge von Schriftzeichen zusammen
- Stellen in Java keinen primitiven Datentyp dar
  - Klasse: **`String`**
- Beispiel: "Dies ist eine Zeichenkette"
  - Zeichenketten werden von doppelten Anführungszeichen eingefasst

Zeichenketten: Formatierung

- Zeichenketten können mit Hilfe von sogenannten "Escape-Sequenzen" formatiert werden

- Beispiele:

```
"\"Diese Zeichenkette steht in Anführungsstrichen\""

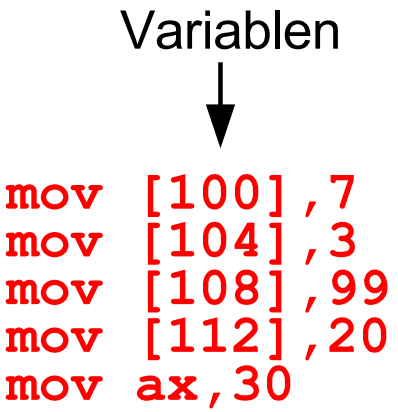
"Ein Schrägstrich und ein
  Anf\u00FChrungsstrich: \\\""
```

  - Angabe von Umlauten entweder direkt oder über Unicode-Escape-Sequenzen

Variablen

Variablen

- Assemblerprogrammierung:
  - Programmierer speichert Daten im Hauptspeicher oder in Registern
- Hochsprache:
  - Programmierer benutzt eine abstrakte "Schublade", in welche Datenwerte gelegt werden können
  - Diese Schublade trägt einen Namen und wird als **Variable** bezeichnet
  - Variablen ermöglichen eine Abstraktion von Speicherzellen und Registern



Variablen und Datentypen

- Variablen werden als zu einem bestimmten Typ gehörend vereinbart
- Dieser Vorgang wird als **Deklaration** bezeichnet
- Der Datentyp legt fest:
  - Wie viel Speicherplatz die Variable benötigt
    - Die "Größe der Schublade"
  - Welche Operationen mit dieser Variable erlaubt sind
- Erst nach der Deklaration kann eine Variable benutzt werden!
- Eine Variable darf im gleichen Gültigkeitsbereich nur einmal deklariert werden

Deklaration von Variablen

- In Java besteht eine Deklaration aus dem **Datentyp** der Variable, dem **Variablenname** und einem abschließenden Semikolon
  - Eine Variablendeklaration ist eine Anweisung für den Compiler
- Beispiel:

```
int myVariable;
```

  - Deklaration einer Variable mit Namen **`myVariable`**, die ganze Zahlen vom Typ **`int`** (d. h. Integer) aufnehmen kann

Variablenname entspricht Start-adresse eines Blocks im Hauptspeicher

```
mov [100],7
```

Datentyp legt fest, wie viele Bytes ein Wert dieser Variable im Hauptspeicher einnimmt

Mehrfachdeklarationen

- Deklarationen von Variablen desselben Typs können in der gleichen Zeile stehen, wenn die Variablennamen durch Kommata voneinander getrennt werden
- Beispiel:
  - An Stelle von

```
int var1;
int var2;
int var3;
```

  - kann folgendes geschrieben werden:

```
int var1, var2, var3;
```

Initialisierung einer Variablen (1)

- Nach der Deklaration ist der Wert einer Variable zunächst noch undefiniert
  - Schublade (Speicherzelle) kann leer sein, es kann sich aber auch irgendein Wert darin befinden
- Mittels der **Initialisierung** wird die Variable auf einen definierten Anfangswert gesetzt
  - Die zu dieser Variable gehörende Speicherzelle enthält dann den ihr zugewiesenen Wert
- Beispiel: Initialisierung einer Variable **`myVariable`** mit dem Wert `0`

```
myVariable = 0;
```

Initialisierung einer Variablen (2)

- Die Initialisierung kann auch als Teil der Deklaration geschehen:

```
int myVariable = 0;
```

  - Dies wird häufig auch als **Definition** bezeichnet

- Es können auch mehrere Variablen in der gleichen Zeile deklariert und initialisiert werden:

```
int myVar1=0, myVar2=10, myVar3=-5;
```

- Auch eine Mischung aus einfachen Deklarationen und Definitionen ist möglich:

```
int myVar1, myVar2=10, myVar3;
```

Variablenname

- In Java erlaubt: Die Buchstaben **`a`** bis **`z`** bzw. **`A`** bis **`Z`**, die Ziffern **`0`** bis **`9`** und die Sonderzeichen _ und **`$`**
  - Groß- und Kleinbuchstaben werden unterschieden
- Nicht erlaubt: Leerzeichen und reservierte Wörter
  - Z. B. **`int`**, **`float`**, **`if`**, **`while`**, **`class`**, etc.
- Variablenname darf nicht mit einer Ziffer anfangen
- Instruktive Namen wählen!

Variablennamen: Konventionen

- `myVariable`
- `my_variable`
- `imyVariable` (Ungarische Notation)

Ungarische Notation (1)

- In Projekten arbeiten viele Programmierer häufig am gleichen Programmcode
- Ein einheitlicher Stil und ein einheitliches Format für die Bezeichner unterstützt die Verstehbarkeit des Codes
  - Wichtig für effiziente Programmierung und Qualitätssicherung
- Die Ungarische Notation definiert eine Konvention für die Namensgebung von Bezeichnern
  - Die Ungarische Notation ist z. B. das Standard-Format für die Software-Entwicklung bei Microsoft

Ungarische Notation (2)

- Konventionen für die Namensgebung von
  - Typen
  - Variablen
  - Konstanten
  - Unterprogrammen (Prozeduren, Funktionen, Methoden)
  - Parameter
  - Klassen

Ungarische Notation (3)

- Einige Beispiele:
  - `c` `char`
  - `b` `boolean`
  - `by` `byte`
  - `i` `int`
  - `w` `unsigned int bzw. word`
  - `l` `long`
  - `s` `Zeichenkette (string)`
  - `lp` `32-bit long pointer`
- Abkürzungen können auch kombiniert werden:

```
int *lpiData; // 32-bit pointer auf int in C++
```

Abkürzungen werden dem Bezeichner vorangestellt!

Variablendeklaration: Zusammenfassung

```
1.  Datentyp  Variablenname;

2.  Datentyp  Variablenname = Anfangswert;

3.  Datentyp  Variablenname1, Variablenname2, ...;

4.  Datentyp  Variablenname1 = Anfangswert1,
              Variablenname2 = Anfangswert2, ...;

5.  Datentyp  Variablenname1 = Anfangswert1,
              Variablenname2, ...;
```

Übung

```
// Gibt es hier falsche Deklarationen?

long good-by;

short shval = 0;

double bubble = 0, trouble = 8

byte the bullet;

int double;

char thisMustBeTooLong;

int 8ball;
```

Variablenwert ausgeben (1)

- Wie bei Zeichenketten
- Anstelle von Zeichenkette Variablennamen benutzen

```
long hoursWorked = 40;

System.out.print("Hours Worked: ");

System.out.println(hoursWorked);
```

- Der Wert von **`hoursWorked`** wird implizit in eine Zeichenkette umgewandelt
  - **`print()`**: Drucke Zeichenkette
  - **`println()`**: Drucke Zeichenkette plus neue Zeile

Variablenwert ausgeben (2)

- Zeichenketten können mittels des **+**-Operators auch mit Variablenwerten verbunden werden

- Anstatt

```
System.out.print("Hours Worked: ");
System.out.println(hoursWorked);
```

- kann man folgendes schreiben:

```
System.out.println("Hours Worked: " + hoursWorked);
```

Beispielprogramm (1)

```
class example {

 public static void main ( String[] args ) {

   long hoursWorked = 40;
   double payRate = 10.0, taxRate = 0.10;

   System.out.println("Hours Worked: " + hoursWorked);
   System.out.println("Pay Amount: " + (hoursWorked * payRate));
   System.out.println("Tax Amount: " + (hoursWorked * payRate *
                      taxRate) );

 }
}
```

- Multiplikation mittels "*", Ausdruck mittels "(...)"
- Um den Wert einer Variable zu lesen oder zu ändern, wird ihr Name in eine Anweisung eingesetzt
- Was gibt das Programm aus?

Beispielprogramm (2)

```
class example {

 public static void main ( String[] args ) {

   long hoursWorked = 40;
   double payRate = 10.0, taxRate = 0.10;

   System.out.println("Hours Worked: " + hoursWorked);
   System.out.println("pay Amount: " + (hoursWorked * payRate));
   System.out.println("tax Amount: " + (hoursWorked * payRate *
                      taxRate) );

 }
}
```

- Anweisung kann auf mehrere Zeilen verteilt werden, solange dabei keine **Schlüsselwörter**, **Bezeichner**, **Zeichenketten** oder **Zahlen** getrennt werden
  - Im Prinzip überall, wo ein Leerzeichen im Code steht
- Abgetrennte Teile gegenüber der ersten Zeile einrücken, um Lesbarkeit zu erhöhen!

Gibt es hier Fehler?

```
cla
    ss example
{
  public static void main ( String[] args )
  {
    long   hoursWorked = 40;
    double payRate     = 10.0, taxRate = 0.10;

    System.out.println("Hours
        Worked: " + hoursWorked );

    System.out.println("pay Amount  : " + (hours
        Worked * payRate) );

    System.out.println("tax Amount  : " + (
        hoursWorked * payRate * taxRate) );
  }
```

Zuweisung

- Bisher wurde der Variablenwert nur bei der Initialisierung geändert
- In derselben Weise – aber ohne erneute Angabe des Typs - wird einer Variable während der Programmausführung ein neuer Wert zugewiesen:

```
myVariable = 0;
```

- Die entsprechende Operation heißt **Zuweisung**
- Syntax:

```
variableName  =  expression;
```

Variablenname ↑ auswertbarer Ausdruck

Zuweisungsoperator in Java

Zuweisung mit Auswertung eines Ausdrucks

- Einer Variable kann auch der Wert eines Ausdrucks zugewiesen werden
- Beispiel:

```
sum = 32 + 8;
```

- Die Auswertung und Zuweisung erfolgt in zwei Schritten:

1. Ausdruck `32 + 8` wird zu `40` ausgewertet
2. Das Ergebnis `40` wird der Variable `sum` zugewiesen

Beispielprogramm

```
class example {
   public static void main ( String[] args ) {
     long hoursWorked = 40;
     double payRate = 10.0, taxRate = 0.10;
     double amount = 0;
     System.out.println("Hours Worked: " + hoursWorked);
     amount = hoursWorked * payRate;
     System.out.println("Pay Amount: " + amount);
     amount = amount * taxRate;
     System.out.println("Tax Amount: " + amount );
   }
}
```

Zuweisung mit Auswertung der Variable selbst

- Teil des Ausdruckes kann auch die Variable sein, der das Ergebnis zugewiesen wird

```
amount = amount * taxRate;
```

- In diesem Fall wird die Variable zunächst ausgelesen und ihr Wert bei der Auswertung des Ausdruckes verwendet
- Das Ergebnis der Auswertung wird wieder in die Variable geschrieben

Literatur (1)

[1] Carter, Paul A.: *PC Assembly Language*, 2002, siehe unter "Literatur" auf der DV-Homepage

[2] DIN 44300

[3] Gumm, H.-P.; Sommer, M.: *Einführung in die Informatik*, Oldenbourg, 2002

[4] Flik, Th.: *Mikroprozessortechnik*. 6. Aufl. Berlin: Springer, 2001

[5] Hyde, Randall: *The Art of Assembly Language Programmin*g, http://webster.cs.ucr.edu

[6] Kjell, Bradley: Introduction to Computer Science using Java, 2004 (auf der Webseite des Kurses)

Literatur (2)

[7] Liebig, H.; Flik, Th.: *Rechnerorganisation*. 2. Aufl. Berlin: Springer, 1993

[8] Liebig, H.; Thome, S.: Logischer Entwurf digitaler Systeme, 3. Aufl. Berlin: Springer, 1996

[9] Rechenberg, P.: *Was ist Informatik?* 3. Aufl. München: Hanser, 2000

[10] Rechenberg, P., Pomberger, G.: *Informatik-Handbuch*, Hanser, 1999