



Rechnerstrukturen und Maschinennahe Programmierung

Prof. Dr.-Ing. Andreas Meisel

```
lar    r0,-variaPIeA
ldrh  r1,[r0]
ldr   r2,[r0]
str   r2,[r0,#VariableC-VariableA]
```

@ Zugriff auf Felder (Speicherzellen)

```
ldr   r0,=MeinHalbwortFeld
ldrh r1,[r0]
ldrh r2,[r0,#2]
mov  r3,#10
ldrh r4,[r0,r3]
```

```
double GetMeasurementV
  unsigned char ADC_S
  unsigned int ADC_V
  unsigned int ADC_W
  double Volts
```

```
ldrh r5,[r0,#2]
ldrh r6,[r0,#2]
strh r6,[r0,#2]

/* AD-Wandlung wird mit
out8(ADC_CONTROL,
      0x00);
   Auf das Ende der
do{
    ADC_Status = in8(ADC_REG_STATUS);
}while((ADC_Status & 0x80) == 0);
```



von u
einWor
0]
, #4]
r2
ldi r4, #10, #8]
ldr r5, [r0, #12]
adds r6, r4, r5



```
lNr in ADC_CONTROL gesetzt */
ling) */

Bits ausblend. */
menführen */
```



```
LOW);
HIGH);

lueHigh &
high << 8
MAX_VOLTAGE
```



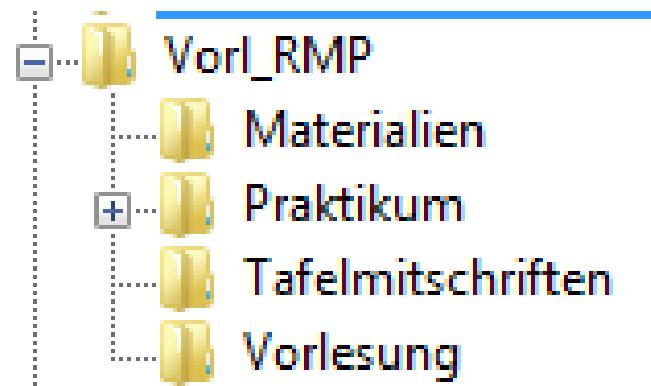
```
Bits ausblend. */
menführen */
```





Vorlesung

- Vortragsfolien → (vor der Vorlesung Pub-Verzeichnis)
- Vertiefungen → Tafel, (nach Vorlesung im Pub-Verzeichnis)
- Übungsaufgaben → Tafel, (nach Vorlesung im Pub-Verzeichnis)
- Programmierbeispiele → Tafel, (nach Vorlesung im Pub-Verzeichnis)



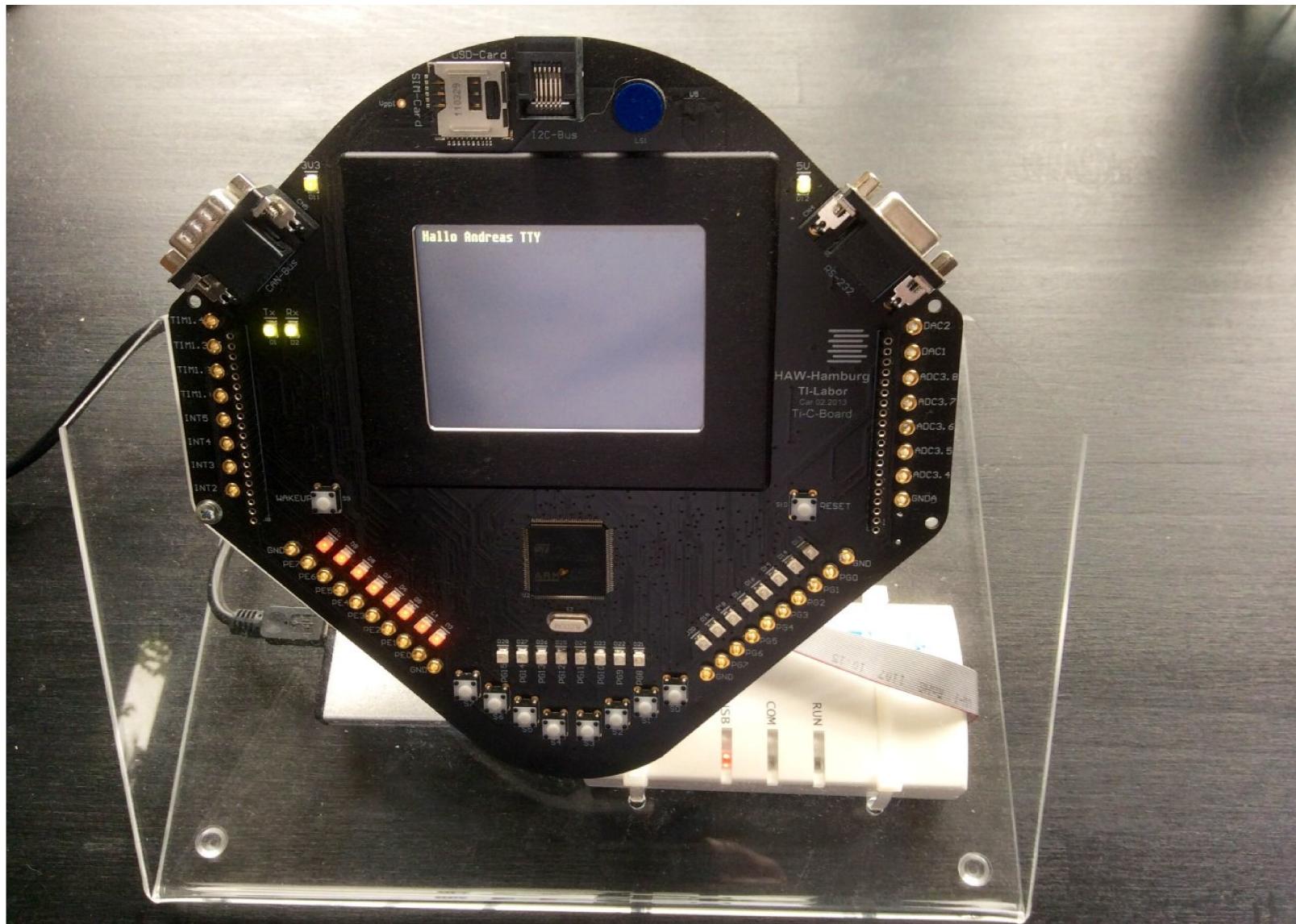
Klausur

- Teilnahmevoraussetzung (PVL) : erfolgreich bestandenes Praktikum
- benotet



Praktikum

- Anwesenheitspflicht (Prüfungsvorleistung)
- Sie müssen angemeldet sein (sollte bereits erfolgt sein)
- Gruppenarbeit: je **2** Personen arbeiten zusammen
- 4 Versuche (Assembler, C, IO-Programmierung,)
- Versuch thematisch vorbereiten → Test zu Beginn des Praktikums (15 Min.)
→ pünktlich erscheinen
- 50% der Punkte aller Tests müssen erreicht werden





1. Grundlagen des „maschinennahen Programmierens“

1.1 Inhalt und Aufbau der Vorlesung

1.1.1 Grundlegende Funktionsweise von Computern

Inhalte:

- einfaches Computermodell :
Speicher, CPU, ALU, Steuereinheit, Busse, IO-Einheiten
- Datencodierung und Befehlscodierung auf Maschinenebene
- Grundbefehlssatz von Computern
- Adressierungsarten für den Datenzugriff

vermittelte Fähigkeiten:

- *Maschinenprogramme und Speicher dumps lesen können*
- *maschinennahe Operationen lesen und programmieren können*



1.1.2 Assemblerprogrammierung am Beispiel des Cortex-M4-Prozessors

Inhalte

- Register
- Basisadressierungsarten
- elementarer Befehlssatz
- Unterprogrammtechniken auf Maschinenebene
- Ein-/Ausgabeoperationen

vermittelte Fähigkeiten

- Assembler-Programme programmieren und debuggen können
- Assemblerprogramme strukturiert aufbauen können
- Unterprogramm-Techniken auf Maschinenebene einsetzen können



1.1.3 ***Grundkonzepte der maschinennahen Computerprogrammierung***

Inhalte:

- notwendige Fähigkeiten maschinennaher Hochsprachen
- Strukturierung, Typen und Abstraktion
- Kennzeichen prozeduraler Programmiersprachen

vermittelte Fähigkeiten:

- *Algorithmen strukturiert umsetzen können*
- *Daten strukturieren können*



1.1.4 Programmiersprache C

Inhalte:

- grundlegende Sprachkonstrukte von C
- Zeiger und Zeigerarithmetik
- Parameterübergabe bei C-Funktionen
- Modularisierung von C-Programmen
- Datenstrukturierung in C-Programmen
- Maschinennahe Programmierung mit C

vermittelte Fähigkeiten:

- *C-Programme schreiben und debuggen können*
- *C-Programme strukturieren und modularisieren können*
- *Zeiger verstehen*
- *IO-Einheiten programmieren können*



1.2 Anwendungsgebiete der maschinennahen Programmierung

Hardware-Einbindung / Gerätetreiber

- Digitale/Analoge Schnittstellen (Meßsignalaufnehmer, Audiointerface)
- Kommunikationsschnittstellen (USB, COM, Netzwerkkarten,...)
- Mediainterfaces (Soundkarten, Joystick, MIDI, Video, ...)





Embedded Controller (Prozessor u. E/A-Schnittst. auf einem Chip für wenige Euro)

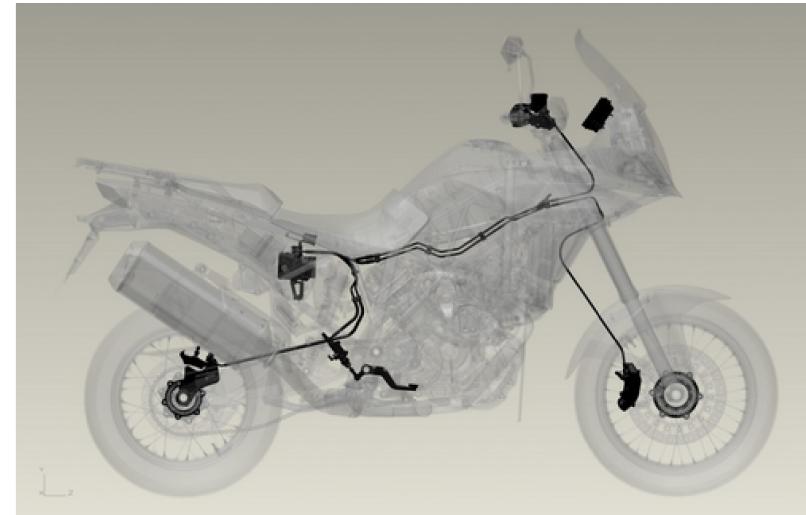
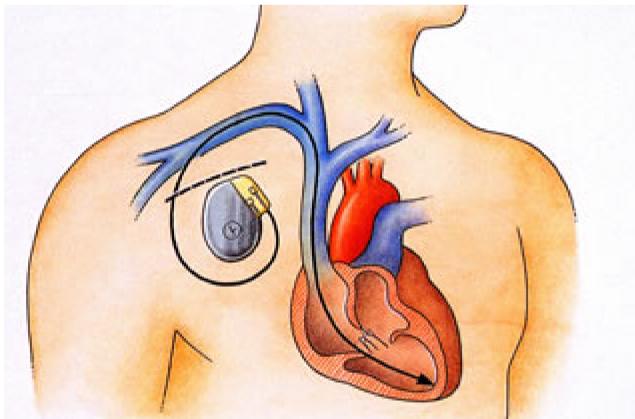
- Haushaltsgeräte (Schaltuhren, Fernbedienungen, Haushaltsgeräte, ...)
- Spielzeuge (Playstations, LEGO-Mindstorm,)
- Autos, Digitalkameras, Smartphones, Reader, Navigationssysteme,
- Maschinensteuerungen





Sicherheitsrelevante Programme

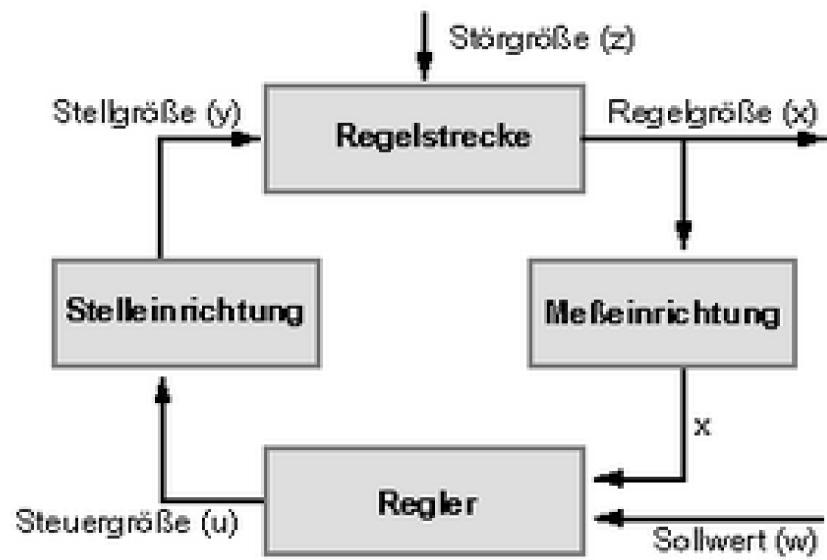
- Medizintechnik (z.B. Herzschrittmacher, Herz-Lungen-Maschine)
- Kraftwerkstechnik, Flugzeugtechnik
- Bremsregelung, ABS





Geschwindigkeitskritische Anwendungen

- Audio- und Videoanwendungen, Musikanwendungen
- Echtzeitsysteme und Regelungen

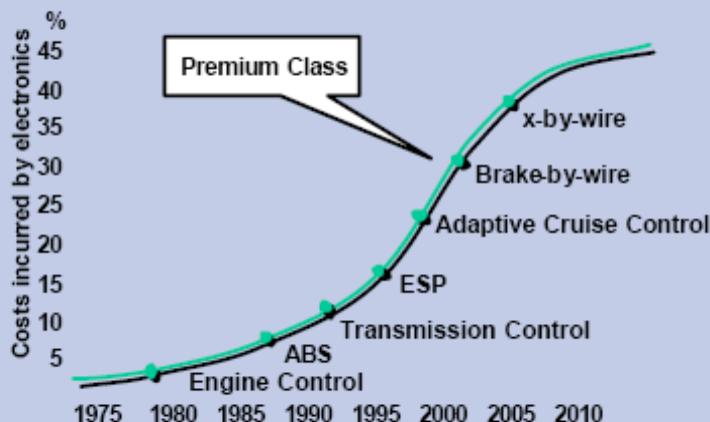




Entwicklungen eingebetteter Systeme in Fahrzeugen

Electronic Content

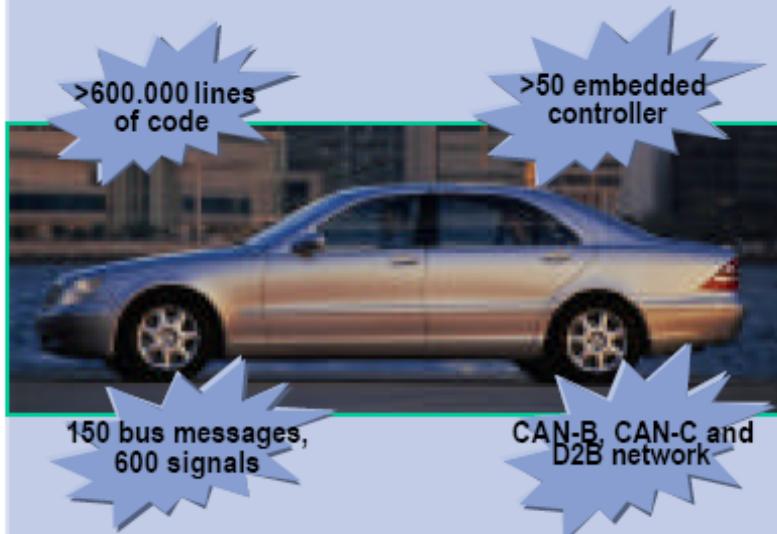
The continuous growth of vehicle electronics leads to a significant increase in software complexity



- more than 80% of functions driven by software
- continuously increasing

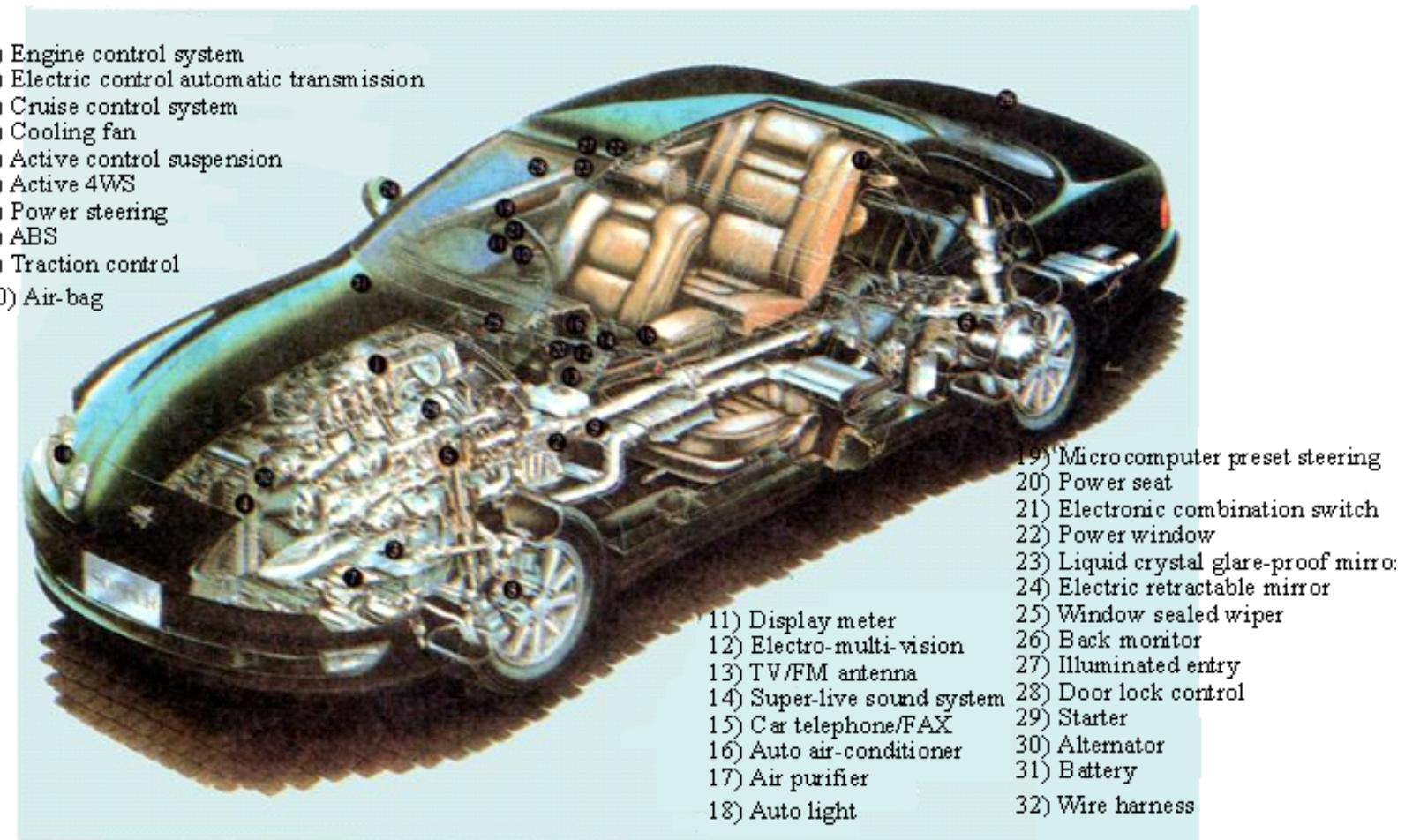
Premium Class, 2000

State of the Art



... not even accounting for telematics

Premium Class, 2000



Provided by Toyota Motor Corporation

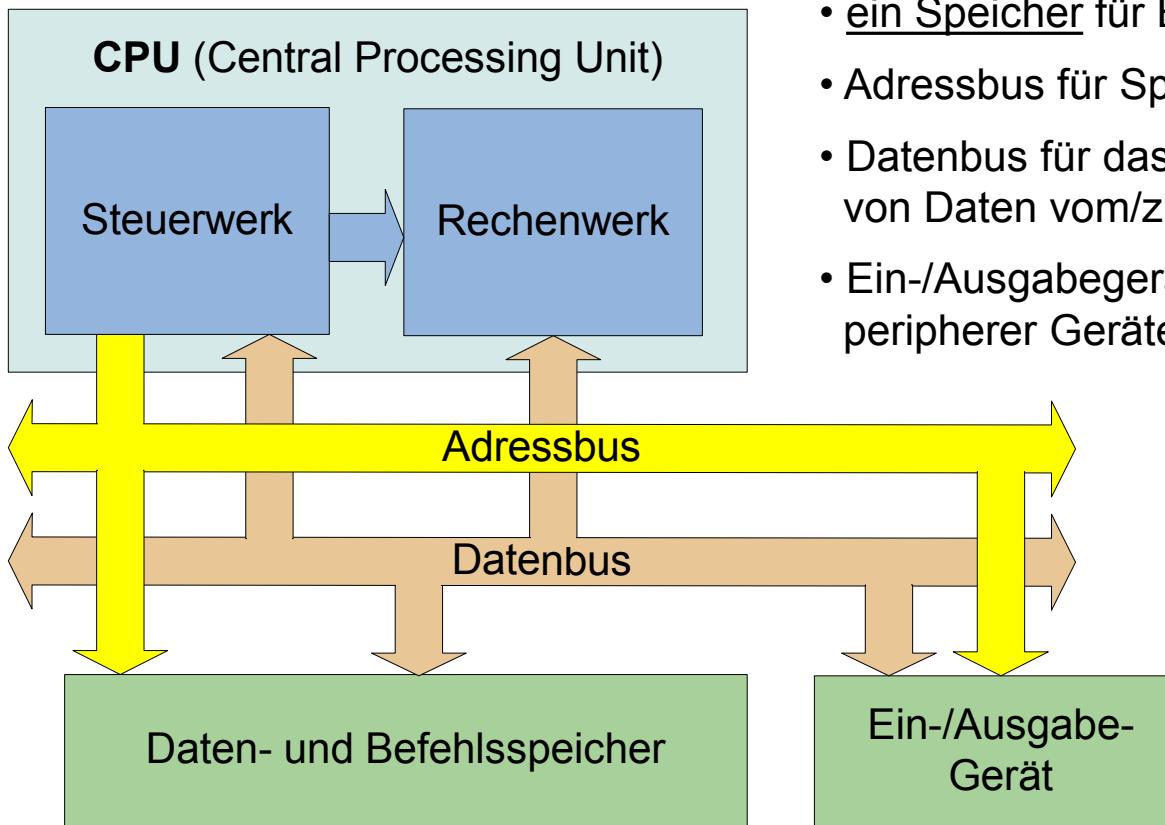


2. Grundsätzliches zum Aufbau von Rechnern

2.1 Von-Neumann-Rechnermodell

2.1.1 Übersicht

Von-Neumann-Rechnermodell



- ein Speicher für Befehle und Daten
- Adressbus für Speicheradressierung
- Datenbus für das Lesen/Schreiben von Daten vom/zum Speicher
- Ein-/Ausgabegerät zur Anbindung peripherer Geräte.



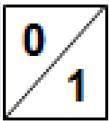
2.1.2 Daten- und Befehlsspeicher

2.1.2.1 Bit = kleinstmögliche Datenmenge

1 Bit =

- ein Schaltelement (*Flip-Flop*), welches 2 Zustände einnehmen kann (0/1),
- stellt die kleinstmögliche *Datenmenge* dar,
- kann genau eine Ja/Nein-Entscheidung speichern.

n Bit können 2^n Zustände speichern:

1 Bit		0	1						
2 Bit		00	01	10	11				
3 Bit		000	001	010	011	100	101	110	111



Fragen: Speicherung von Daten und Befehlen

Angenommen ein Zeichencode umfasst 80 unterschiedliche Zeichen.
(26 Kleinbuchstaben, 26 Großbuchstaben, 10 Zahlzeichen, 18 Sonderzeichen)

Wieviele Bits sind zur Codierung eines Zeichens mindestens notwendig?

Wie viele Zustände lassen sich mit 12Bit codieren?

Wieviele Bits sind notwendig, um die 256 Graustufen eines Bildpunktes unterscheiden zu können?



2.1.2.2 Byte = kleinste Zugriffseinheit

1 Byte =

- eine Gruppe aus 8 Bits,
- kann 256 unterschiedliche Zustände einnehmen,
- Ist die kleinste schreib- und lesbare Datenmenge in den meisten Computern (\rightarrow Bytemaschinen).
- Jedes Byte eines Speichers ist durch eine eigene Adresse ansprechbar (in Bytemaschinen).

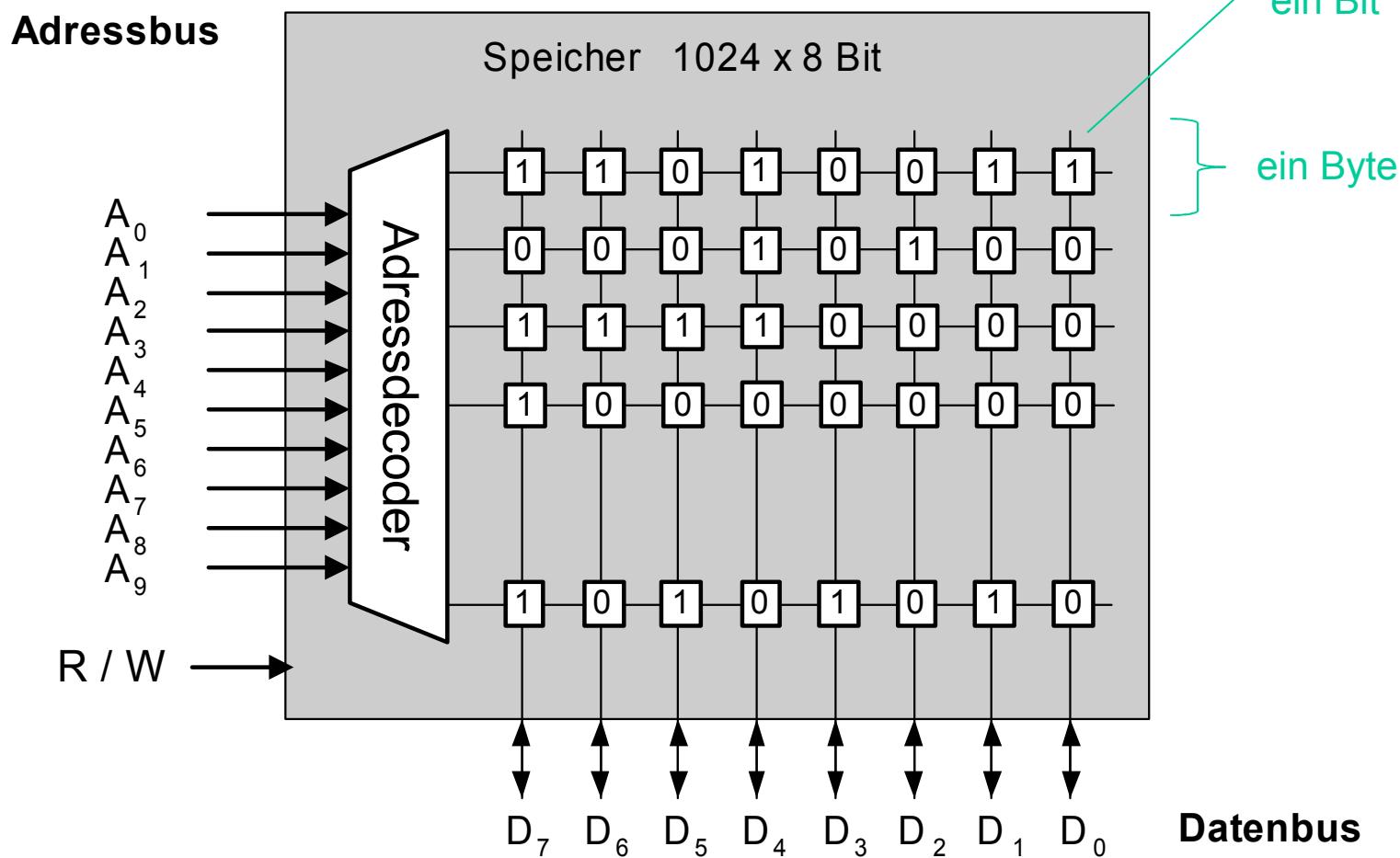
Daten bestehen meist aus einer ganzzahligen Anzahl von Bytes.

Beispiele:	Musiksignale	\rightarrow 2, 3 oder 4 Byte
	Farbecodierung eines Farbbildpunktes	\rightarrow 3 Byte



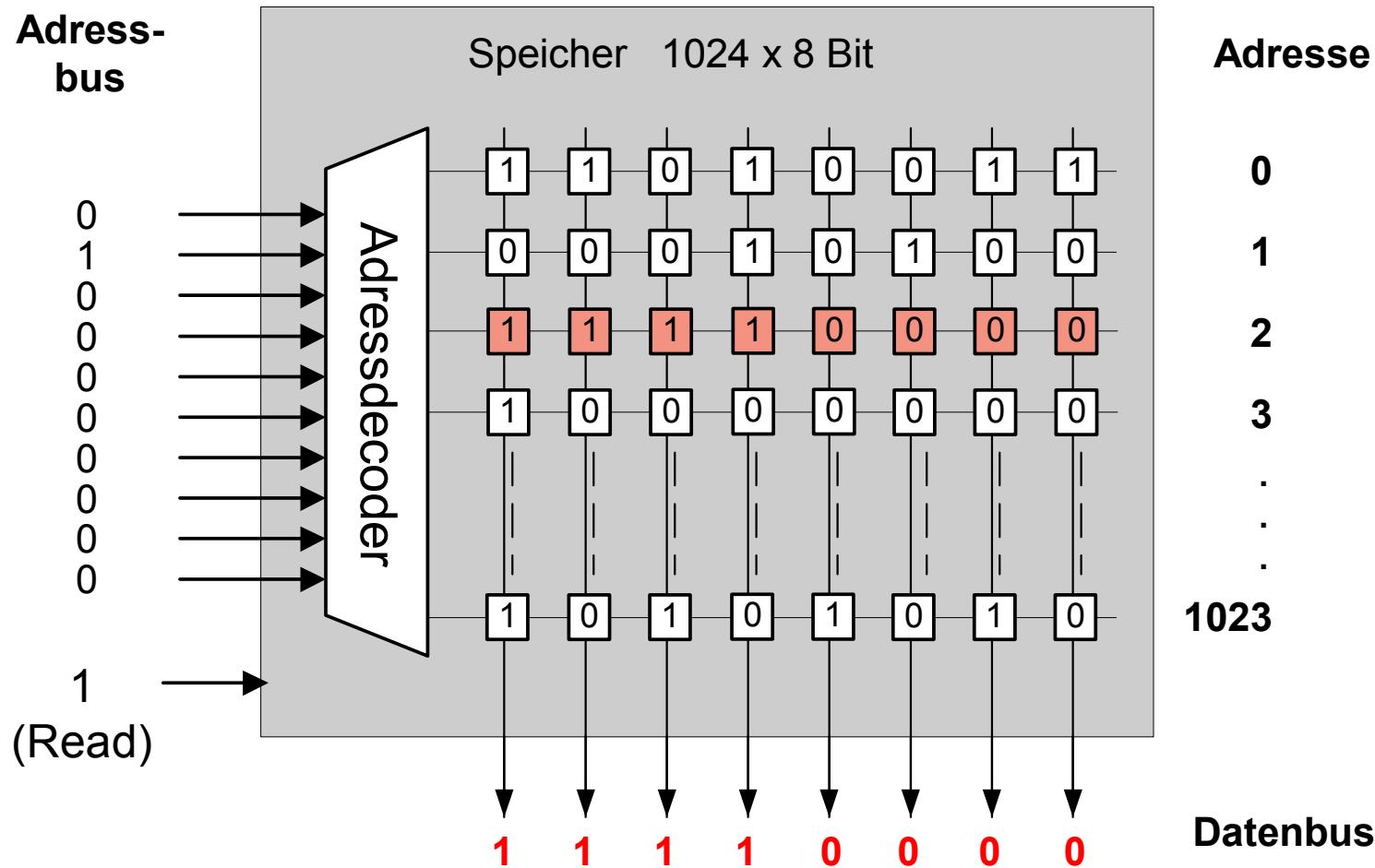
2.1.2.3 Realisierung des Daten- und Befehlsspeichers (einf. Beispiel)

Zweck: Speichern und Lesen von Daten und Maschinenbefehlen unter einer Adresse (hier Bytemaschine).



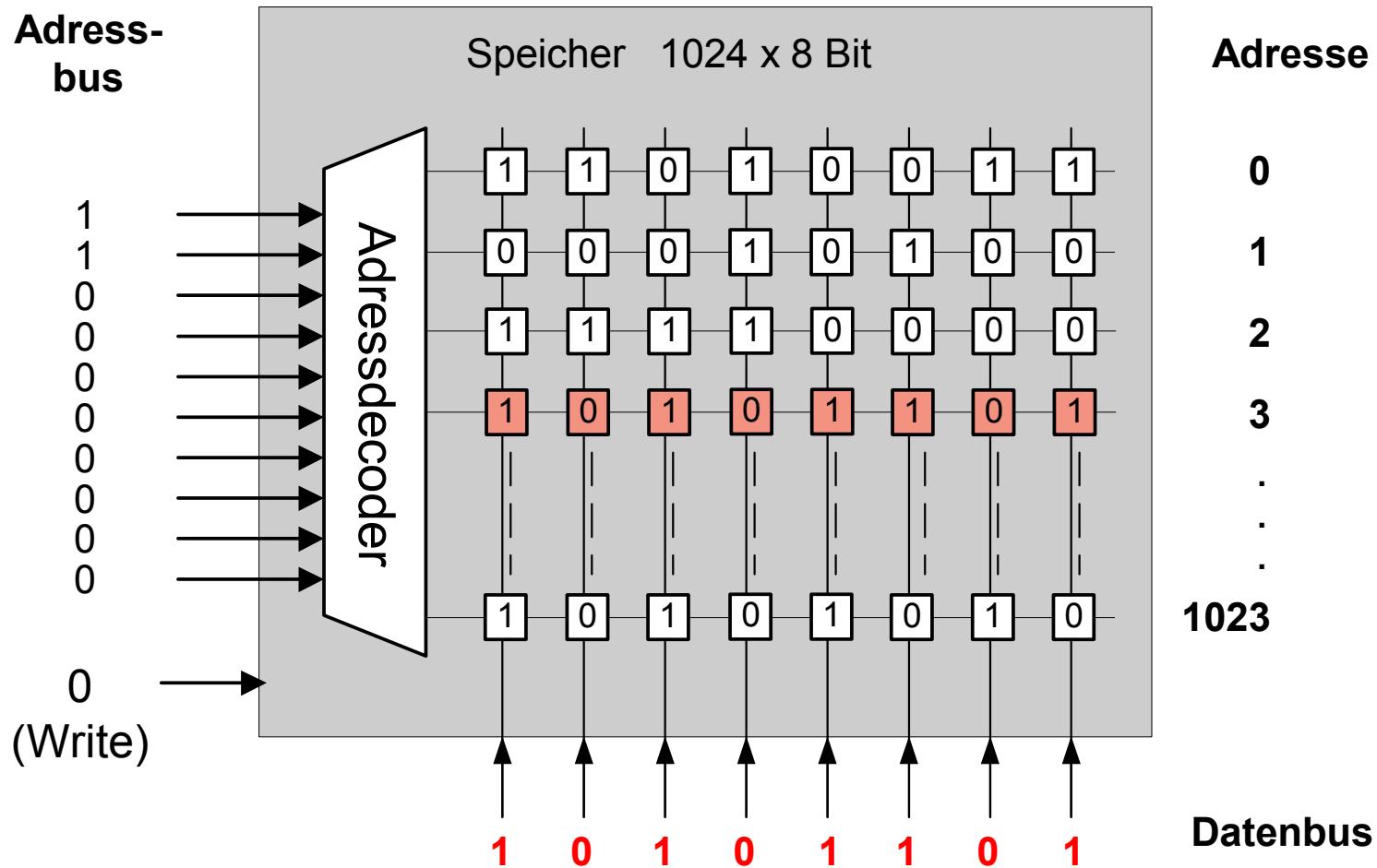


Lesen eines Speicherwortes





Schreiben eines Speicherwortes





2.1.3 Adress-/Datenbus verschiedener Rechner

- typ. Adressbusbreiten:

- 16 bit (z.B. 6502)	→ Adressraum 2^{16}	= 65.536
- 24 bit (z.B. 68000)	→ Adressraum 2^{24}	= 6.777.216
- 32 bit (Cortex M4)	→ Adressraum 2^{32}	= 4.294.967.296
- 36 bit (Pentium II, .. IV)	→ Adressraum 2^{36}	= 68.719.476.735

- typ. Datenbusbreiten:

- 8 bit (z.B. 6502)
- 16 bit (z.B. 68000)
- 32 bit (z.B. 68030, **Cortex M4**)
- 64 bit (z.B. Pentium)

• übliche kleinste adressierbare Einheit ist **1 Byte** → **Byte-Maschine**

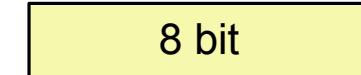


2.1.4 Maschinennahe Datenformate (Anm.: Bezeichnungen maschinenabh.)

Byte-Format (1 Byte)

unsigned 0 ... 255 (2^8-1)

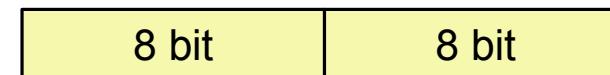
signed -128 ... 127



Halbwort-Format (2 Byte)

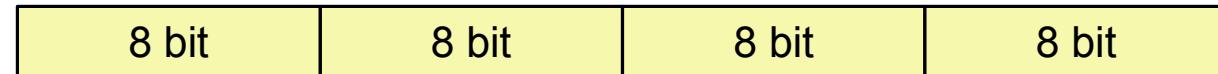
unsigned 0 ... 65 535 ($2^{16}-1$)

signed -32 768 ... 32 767

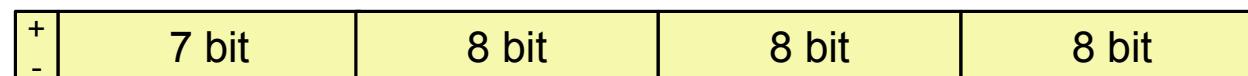


Wort-Format (4 Byte)

unsigned 0 ... 4 294 967 295 ($2^{32}-1$)



signed -2 147 483 648 ... 2 147 483 648





Hinweis zur Darstellung von Speicherauszügen (Dumps)

Eine größere Bitgruppe ist für den Menschen kaum lesbar/merkbar.

Beispiel : 4 Byte

10100011 11101011 10000010 00100011

Bit-muster	Hex-wert
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Eine kompaktere Darstellung erhält man wie folgt:

1. Jedes Byte wird in zwei Hälften (*Nibble* = 4 bit) unterteilt.
2. Der Wert eines Nibbles wird dann kurz durch seinen Hexadezimalwert dargestellt.

10100011 11101011 10000010 00100011
A 3 E B 8 2 2 3



Fragen: Darstellung von Speicherauszügen (Hexdump)

Gegeben ist der folgende Hexdump einer Bytemaschine ab Adresse 1000:

A4 33 72 F9 B0

1. Wie viele Bits und Bytes sind dargestellt ?
2. Was ist die Adresse des letzten Bytes?
3. Geben Sie das gespeicherte Bitmuster an.
4. Handelt es sich um codierte Buchstaben, Zahlen oder Befehle?



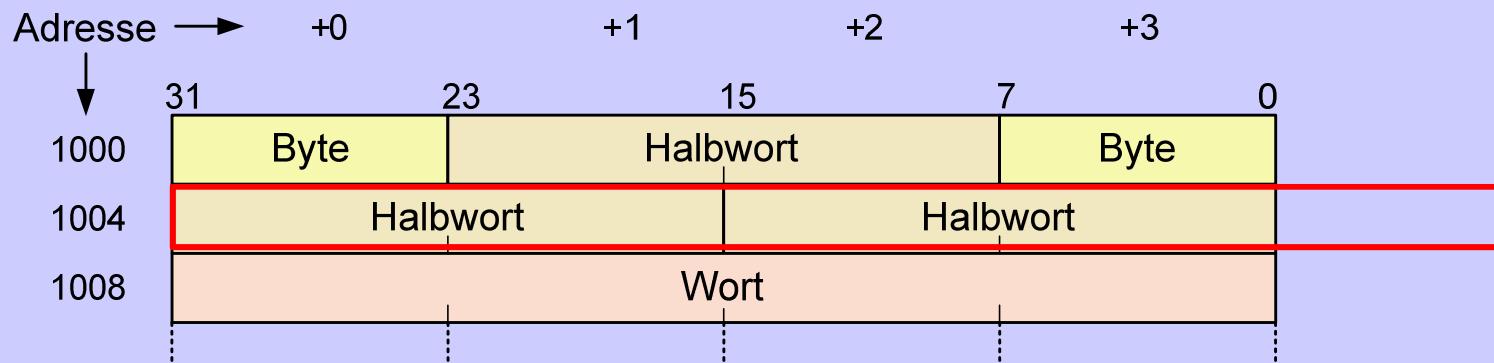
2.1.5 Speicherorganisation

Die Daten können auf unterschiedliche Art im Speicher abgelegt werden:

2.1.5.1 n-Bit-Speicher (= n-Bit-Zugriffsbreite = n-Bit-Datenbusbreite)

→ d.h. es können n Bit in einem Zugriff gelesen/geschrieben werden
typ.: 16-Bit-Speicher, 32-bit-Speicher, 64-bit-Speicher

Beispiel: 32-bit-Speicher ab Adresse 1000



Anm.: Nicht immer braucht man 32 bit (z.B. Bildpunkte, ASCII-Zeichen,).
Um keinen Speicher zu verschwenden, müssen auch 16 bit oder 8
Bit adressierbar sein. → deswegen „Byte-Maschine“



2.1.5.2 Byte-Reihenfolge (byte ordering) bei Halbworten (2 Byte) u. Worten (4 Byte)

big endian: „höchstwertiges“ Byte auf niedrigster Adresse

littleEndian: „höchstwertiges“ Byte auf höchster Adresse (sog. *byte swapping*)

Beispiel: Folgende Daten sind im Speicher ab Adresse 1000 abgelegt (Hex.):

Word:	(4-Byte)	12 34 56 AB
Halfword:	(2-Byte)	41 44
Byte:		78
Byte:		89
String:		„ABCD“

big-endian

1000	12	34	56	AB
1004	41	44	78	89
1008	41	42	43	44

little-endian

byte swapping (nur bei Worten und Halbworten im Speicher)

1000	AB	56	34	12
1004	44	41	78	89
1008	41	42	43	44



2.1.5.3 Speicherausrichtung (data alignment)

aligned: Worte und Langworte werden so abgelegt, dass sie in einem Zugriff gelesen werden können.

→ Worte beginnen auf Adr., die durch 4 teilbar sind (Wortgrenzen).

→ Halbworte beginnen auf Adr., die durch 2 teilbar sind (Halbwortgrenzen).

Beispiel: 32-Bit-Zugriffsbreite, big-endian,
folgende Daten sind im Speicher abgelegt (Hex.):

Halfword: (2-Byte)

Word: (4-Byte)

Byte:

Halfword:

Halfword:

12 34

56 78 9A BC

CD CE CF

1F A1

B5 D5

1000	12	34	xx	xx
1004	56	78	9A	BC
1008	CD	CE	CF	xx
	1F	A1	B5	D5

1000	12	34	56	78	
1004	9A	BC	CD	CE	
1008	CF	1F	A1	B5	
	D5				



BEISPIEL: Cortex M4 im Praktikumssystem

Alignment: Worte immer aligned an Wortgrenzen
Halbwörter immer aligned an Halbwortgrenzen
Byte-ordering: little endian (d.h. byte swapping bei Halbworten und Worten)

Folgende Daten sind aligned im Speicher abgelegt
(Anm.: Hexadezimaldarstellung):

Halfword:	12 34	→ alignment erzwingen (ALIGN 4)
Word:	56 78 9A BC	
Byte:	CD CE CF	→ alignment erzwingen (ALIGN 2)
Halfword:	1F A1	

Im Debugger sieht der Speicherinhalt wie folgt aus:

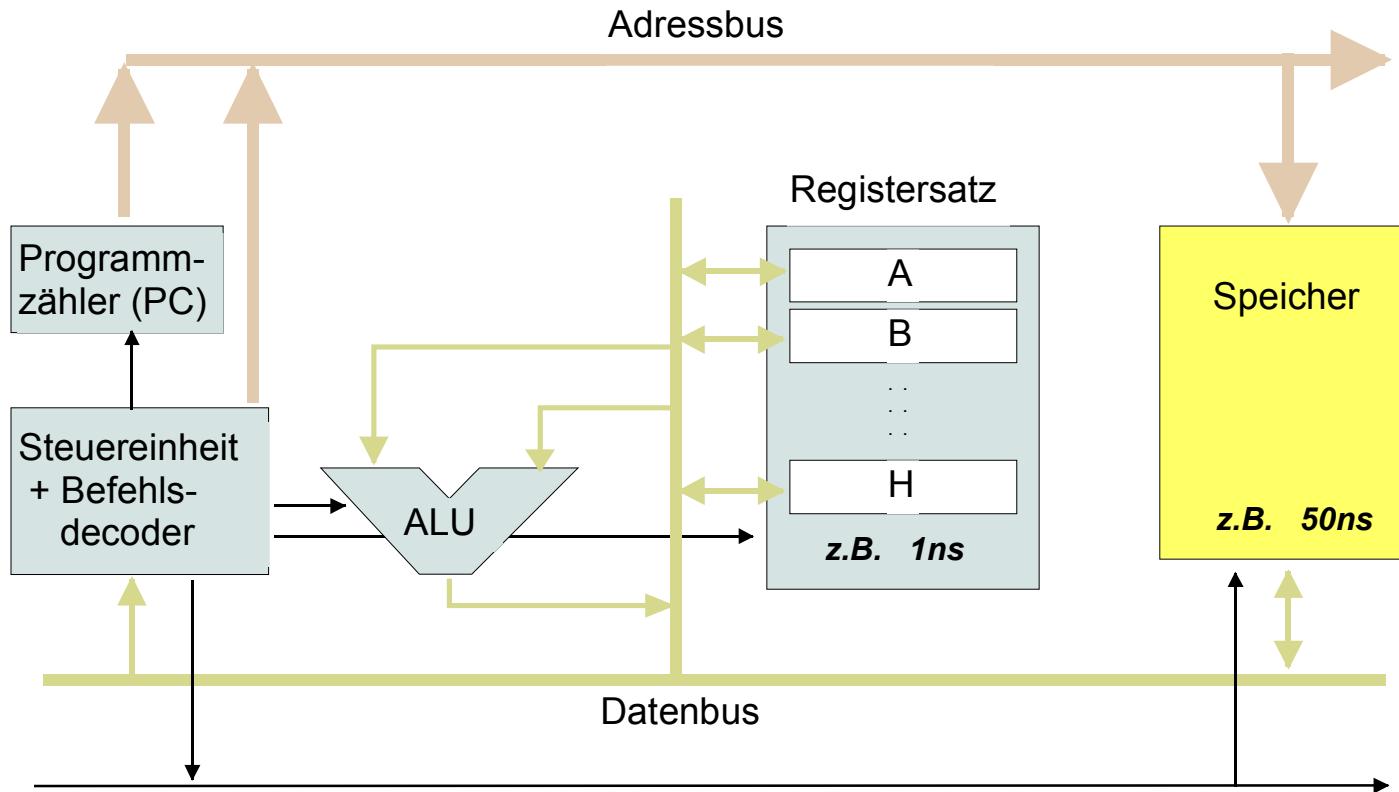
34 12 **00 00** BC 9A 78 56 CD CE CF **00** A1 1F

Würde man beim Ablegen der Daten auf das Alignment verzichten (was ohne weiteres geht), würden die Daten beim Zugriff falsch gelesen.



2.1.6 Zentraleinheit (**CPU = central processing unit**)

..... eines einfachen Von-Neumann-Rechners
→ gemeinsamer Speicher für Befehle und Daten





Zentraleinheit (CPU)

Programmzähler :

(PC = Program counter)

Enthält die Adresse des nächsten auszuführenden Befehls.

Steuereinheit bzw. Befehlsdecoder :

(CU = Control Unit)

Dekodiert den Instruktionsteil (Instr.) und steuert abhängig davon das Zusammenspiel der anderen Einheiten (Fetch- und Execute Phase).

Rechenwerk :

(ALU = Arithmetic and logic unit)

Führt Berechnungen in Byte-, Word- oder Longword-Format durch z.B. ADD, ADC, SUB, bitweises AND/OR,

Statusregister :

(CCR = Condition code register)

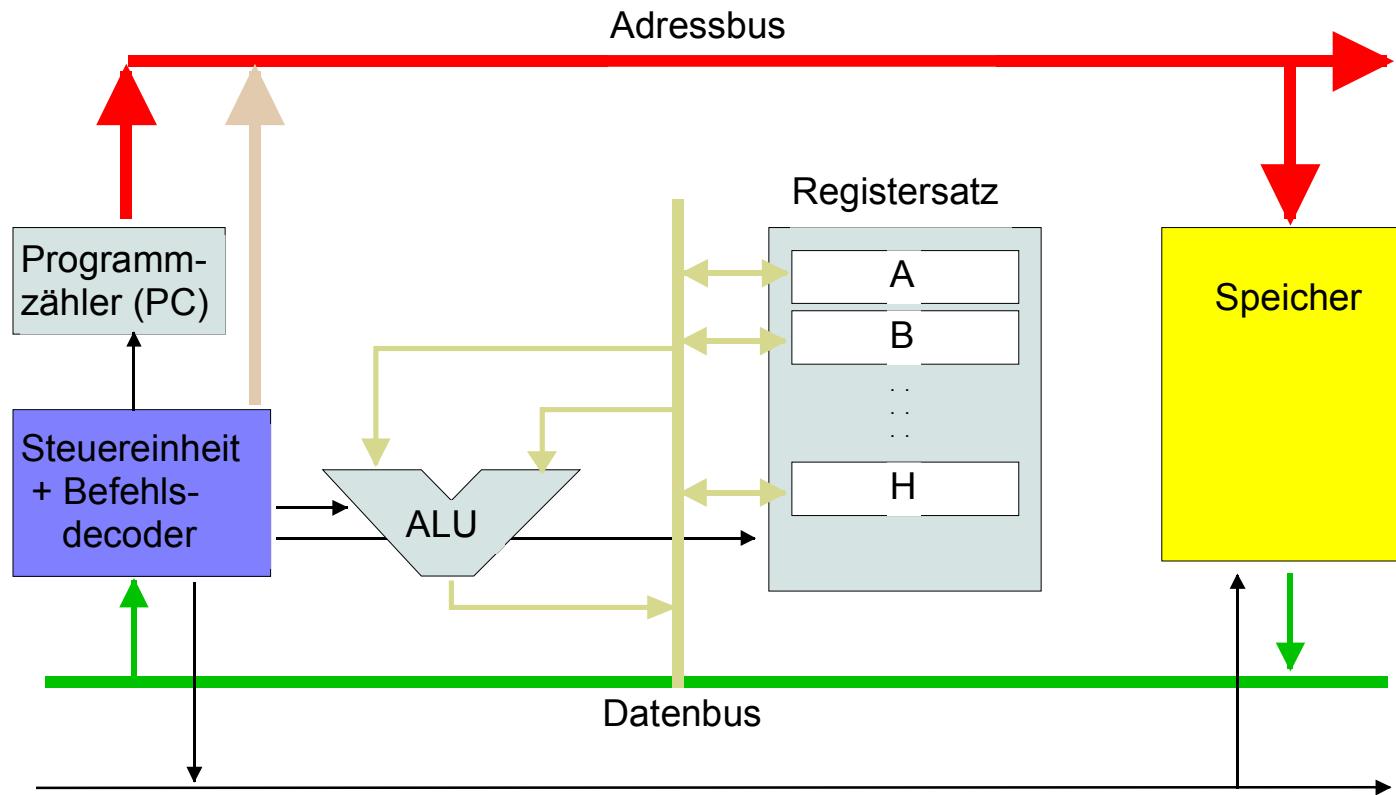
Mehrere 1-bit-Flags, die der Prozessor bei der Befehlsausführung setzt (z.B. Carry-Flag, Zero-Flag, Overflow-Flag,).



2.1.7 Befehlausführung - "Fetch-Decode-Cycle" (Feststellen, was zu tun ist)

Beispiel: "Lade Inhalt von Speicher 1000 in Register A"

1. Der Programmzähler legt die aktuelle Programmadresse an den Adressbus.
2. Der Befehlsdecoder analysiert das Datenwort (Maschinenbefehl), um welche Instruktion es sich handelt.
--> ab jetzt ist bekannt: "Inhalt von Speicher 1000 in Register A kopieren"

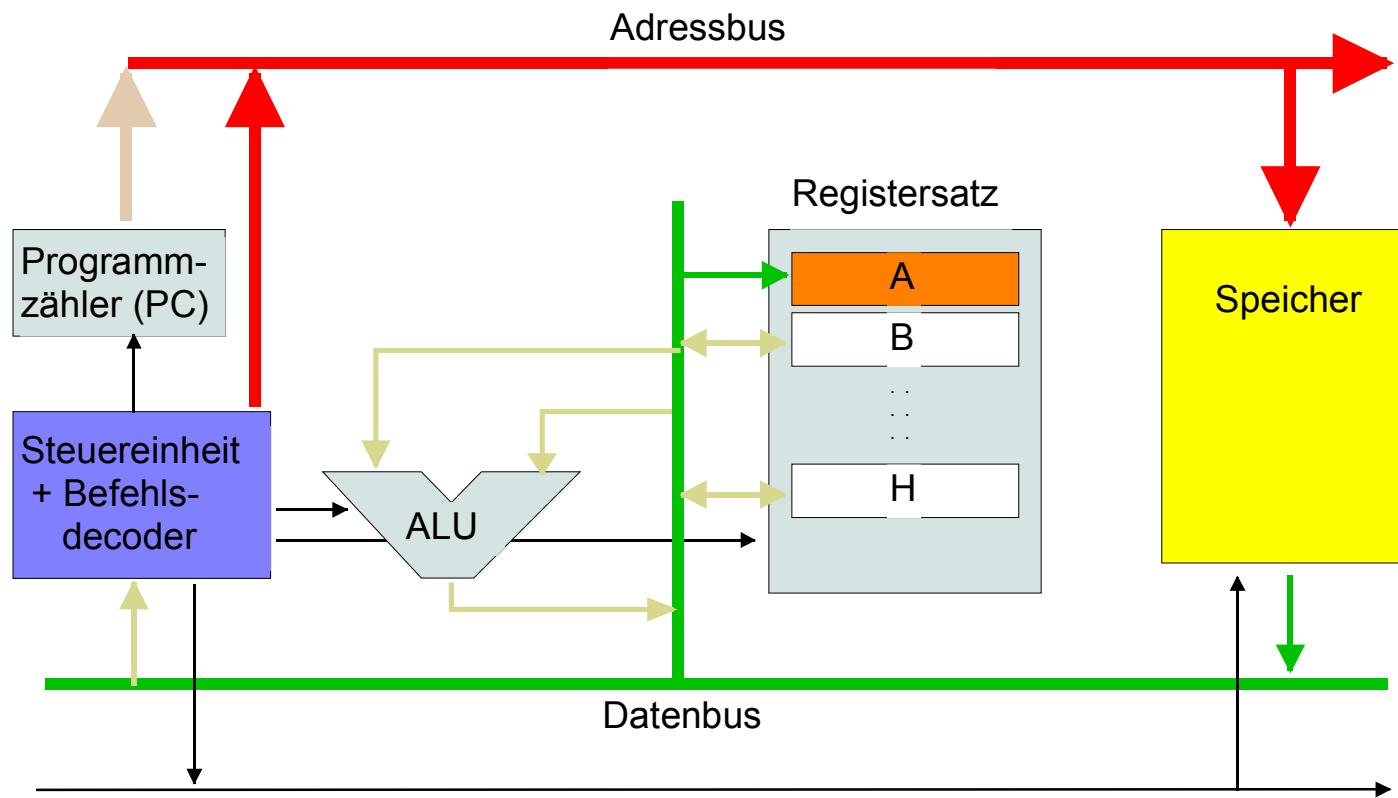




Befehlsausführung - "Execute-Cycle" (den Befehl Ausführen)

Beispiel: "Lade Inhalt von Speicher 1000 in Register A"

3. Steuereinheit legt Adresse 1000 an den Adressbus.
4. Das ausgegebene Datenwort wird nach Register A kopiert.





3. Daten

3.1 Einführende Bemerkungen

Daten = Informationen, die in Form von Zeichen (Symbolen) abgelegt sind.

- Zweck : - Verarbeitung
- Speicherung
- Übertragung

Einfache Datentypen:

- ganze Zahlen (z.B. 5, 233, -17, -2659)
- reelle Zahlen (z.B. 12.3, $-17.5 \cdot 10^{12}$, $0.235 \cdot 10^{-6}$)
- Zeichen (z.B. a, b, c, x, z, F, G, P, Q, 1, 2, 3)

Codierung : Daten (Zahlen, Zeichen, Bilddaten, Amplitudenwerte,)
in eine zweckgerechte Form bringen.

Beim Computer : Um Informationen verarbeiten und speichern zu können ist eine Abbildung dieser Zeichen auf Binärzeichen {0,1} notwendig !



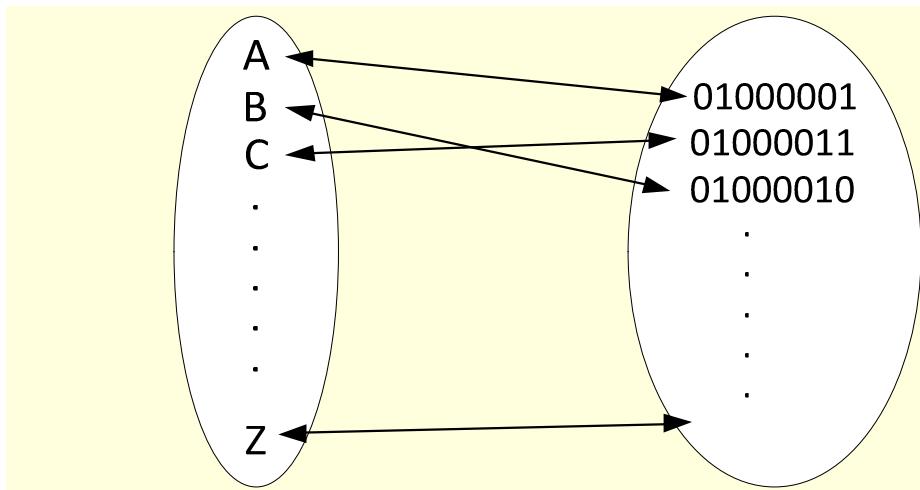
3.2 Codierung

Allgemein gilt:

Codierung = Inhalt einer Information oder Nachricht in einer verarbeitbaren, speicherbaren oder übermittelbaren Form darstellen (Zweck).

Achtung: Nicht verwechseln mit **Verschlüsselung**
 (= gegen unbefugten Zugriff schützen).

Code = Abbildungsvorschrift, mit der die Zeichen eines Quellalphabets A auf Zeichen (oder auch Zeichenketten, d.h. Worte) eines Zielalphabets B umkehrbar eindeutig abgebildet werden.





3.3 Wichtige Zifferncodes

3.3.1 Stellenwertsysteme

Eine vielfach zweckmäßige Zahlencodierung bieten *Stellenwertsysteme*, d.h. , bei einem Zahlensystem zur Basis b hat Stelle i (rechts beginnend) die Wertigkeit b^i .

Dezimalsystem

Beispiel : $1243 = 1 * 10^3 + 2 * 10^2 + 4 * 10^1 + 3 * 10^0$



Basis: 10

Ziffern der Mantisse: 0,1,2,3,4,5,6,7,8,9

$$\sum_{i=0}^{n-1} a_i \cdot 10^i$$

mit

n :

Stellenzahl der Binärziffer

a_i :

Mantisse der Stelle i

10^i :

Wertigkeit der Stelle i



Binäre Zahlendarstellung

In der Digitalrechnertechnik wird für die Zahlencodierung häufig das *Binärsystem eingesetzt*.

Basis: 2

Ziffern der Mantisse: 0,1

$$\sum_{i=0}^{n-1} a_i \cdot 2^i$$

mit

n :

Stellenzahl der Binärziffer

a_i :

Mantisse der Stelle i

2^i :

Wertigkeit der Stelle i

Beispiel: $1010_B = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 10_D$



Anm.: In vielen Programmiersprachen wird dem Rechner durch ein vorangestelltes **0b** mitgeteilt, dass die eingegebene Zahl binär zu interpretieren ist.

Bsp.: $a = 0b11$ ist gleichbedeutend mit $a = 3$



Hexadezimale Zahlendarstellung

Für die kompakte Beschreibung von Binärzahlen ist das *Hexadezimalsystem* sehr gut geeignet.

Basis: 16

Ziffern der Mantisse: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

$$\sum_{i=0}^{n-1} a_i \cdot 16^i \quad \text{mit} \quad \begin{aligned} n &: \quad \text{Stellenzahl der Binärziffer} \\ a_i &: \quad \text{Mantisse der Stelle } i \\ 16^i &: \quad \text{Wertigkeit der Stelle } i \end{aligned}$$

Beispiel: $E2A1_H = E * 16^3 + 2 * 16^2 + A * 16^1 + 1 * 16^0 = 58017_D$



Anm.: In vielen Programmiersprachen wird dem Rechner durch ein vorangestelltes **0x** mitgeteilt, dass die eingegebene Zahl hexadezimal zu interpretieren ist.

Bsp.: $a = 0x11$ ist gleichbedeutend mit $a = 17$



ÜBUNG: Zahlenaumwandlung Dezimal \leftrightarrow Binär⁽¹⁾

Wandeln Sie folgende Dezimalzahlen in Binärzahlen um:

- a) durch „Addition von Zweierpotenzen“,
- b) durch „Modulo-Division“.

41, 221



ÜBUNG: Zahlenumwandlung Hexadezimal \leftrightarrow Binär/Dezimal

Wandeln Sie folgende Hexadezimalzahlen in Dezimalzahlen um:

- a) durch „Addition von Potenzen“,
- b) mit Hilfe des „Horner-Schemas“.

$A35_H$, $AC2F_H$, $12CF_H$

Wandeln Sie folgende Hexadezimalzahlen in Binärzahlen um.

$AB73_H$, $12BC_H$

Wandeln Sie folgende Binärzahlen durch „Gruppieren“ in Hexadezimalzahlen um.

11110110_B , 1100111_B



3.3.2 Vorzeichenlose Binärzahlen mit begrenzter Stellenzahl (unsigned)

Innerhalb von Rechnern werden Binärzahlen (i.Allg.) mit einer vorgegebenen Zahl von Stellen n codiert (z.B. 8 Bit, 16 Bit, 32 Bit).

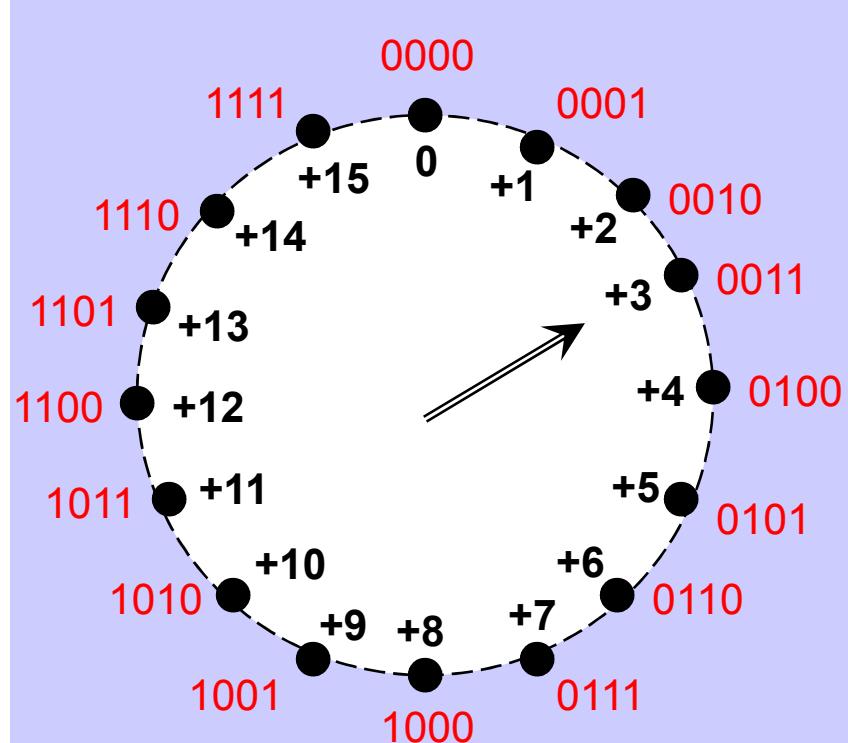
Der darstellbare Zahlenbereich vorzeichenloser Zahlen ist daher begrenzt auf 0 $2^n - 1$.

Beispiele:

8 Bit-Zahlen: 0 255_D

16 Bit-Zahlen : 0 . . . 65 535_D

s. Tafel



Beispiel:

Darstellung vorzeichenloser Zahlen im 4-Bit-Format.



3.3.3 Vorzeichenbehaftete Binärzahlen mit begrenzter Stellenzahl (signed)

Das **Einerkomplement (Komp₁(z))** einer Binärzahl erhält man durch bitweises invertieren der Binärzahl z.

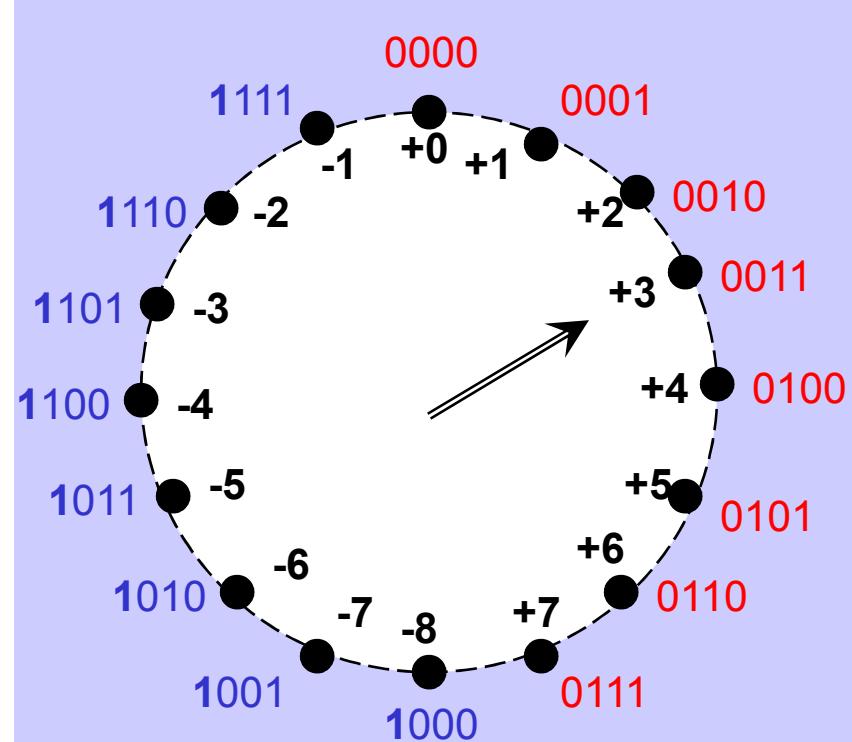
Das **Zweierkomplement (Komp₂(z))** erhält man durch: Komp₁(z) + 1

Negative Zahlen werden üblicherweise in der Zweierkomplement-Codierung dargestellt. Die höchstwertige Bit zeigt dabei das Vorzeichen an (0=pos., 1=neg.).

Der darstellbare Zahlenbereich ist dann:

$$-2^{(n-1)} \dots 0 \dots +2^{(n-1)} - 1$$

s. Tafel



Beispiel:

Darstellung negativer Zahlen in der 4-Bit-Zweierkomplement-Codierung



ÜBUNG: Zweierkomplement (7,8)

s. **Tafel** : anschauliche Erläuterung vorab

Geben Sie zu den nachfolgenden negativen Dezimalzahlen die Binärzahlen im **8-bit 2-er-Komplement** an

-1_D , -7_D , -32_D

Geben Sie zu den folgenden vorzeichenbehafteten Binärzahlen (**8-bit 2-er-Komplement-Codierung**) die Dezimalwerte an:

$1001\ 0001_B$, $0100\ 0001_B$, 1000001_B



3.3.4 Festkommazahlen

Festkommazahlen lassen sich ebenfalls durch das Stellenwertsystem darstellen. Die Nachkommastellen haben dann die Wertigkeit 2^{-1} , 2^{-2} , 2^{-3} , u.s.w..

Für eine Binärzahl mit n Vorkommastellen und m Nachkommastellen gilt somit:

$$\sum_{i=-m}^{n-1} a_i \cdot 2^i$$

Beispiel: $9.75_D = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1001.11_B$

In Rechnern ist die Stellenzahl meist begrenzt (z.B. 8 bit vor und nach dem Komma). Daraus resultiert eine größte und kleinste darstellbare Zahl.

Beispiel:

Vorzeichenlose 16 bit Festkommazahl mit 8 bit Vor-/Nachkommaanteil

größte Zahl: $1111\ 1111.1111\ 1111_B = 255.99609375_D$

kleinste Zahl (ungl. 0): $0000\ 0000.0000\ 0001_B = 0.00390625$



.... Festkommazahlen: Diskussion

Vorteile:

- ermöglicht schnelle Berechnungen auch auf einfachen Prozessoren ohne Floatingpoint-Unit (FPU)
- einfache Arithmetik

Nachteile:

- trotz begrenzter Dezimalstellenzahl kann die Zahl der notwendigen Binärstellen für eine fehlerlose Darstellung unendlich sein.

Beispiel: $0.8_D = 0.\overline{1100}_B$

- Einsetzbarkeit und notwendige Stellenzahl sind für den jeweiligen Einsatzfall sehr genau zu prüfen (Rundungsfehler).

Beispiel: aus 0.8_D wird bei 8 Nachkommastellen 0.796875_D
(err: 0.003125)

- Für hohe Genauigkeiten sind sehr viele Stellen notwendig.



ÜBUNG: Festkommazahlen

Geben Sie zu folgenden Dezimalzahlen die Festkomma-Binärzahlen
(8 Vorkomma- und 8 Nachkommastellen) an.

15.6875_D , 37.1875_D , 10.2_D

Verwandeln Sie die zu 10.2_D korrespondierenden Binärzahlen wieder zurück in Dezimalzahlen. Wie groß ist der Fehler?



3.3.5 Gleitkommazahlen

Gleitkommazahlen (Floatingpoint) dienen zur Codierung (einer Teilmenge) der reellen Zahlen.

Prinzip: Jede Zahl, unabhängig von ihrer Größe, wird durch eine feste Anzahl *signifikanter Ziffern* dargestellt.

Beispiele:

$$0.31415 * 10^1$$
$$0.12345 * 10^{-23}$$
$$0.43434 * 10^{16}$$

In Rechnern gilt: $\text{Zahl} \approx m * b^e$ mit

- m: Mantisse (Stellenzahl begrenzt)
- b: Basis (= 2 bei Binärzahlen)
- e: Exponent (Stellenzahl begrenzt)

Weltweit akzeptierter Standard wurde definiert vom IEEE (*Institut of Electrical and Electronics Engineers*) → **Standard 754**



..... Gleitkommazahlen : interne Darstellung

Darstellung von 4-Byte-Gleitkommazahlen (einfache Genauigkeit):

31 sign s	30 exzeß e (= 8 bit)	23	22 fraction f (= 23 bit)	0
--------------	-------------------------	----	-----------------------------	---

Normalisierte Zahlen werden dann wie folgt dargestellt:

$$\text{Zahl} = (-1)^s \cdot 2^{e-127} \cdot 1.f$$

sign (s) : Vorzeichen der Zahl (0: positiv, 1: negativ)

exzeß (e) : Vorzeichen des Exponenten wird durch Addition des sog. *Exzeß* (=127) berücksichtigt. ($e \in [1, 254]$)

Beispiel:

$2^0 = 2^{(0+127)-127}$	$\rightarrow e = 127$
$2^{-7} = 2^{(-7+127)-127}$	$\rightarrow e = 120$
$2^{15} = 2^{(15+127)-127}$	$\rightarrow e = 142$

fraction (f) : Nachkommaanteil der Mantisse, wobei die Zahl so normiert wird, daß genau eine 1 links neben dem Komma steht.

Beispiel:

$1101011.101 = 1.101011101 * 2^6$
$0.001011011 = 1.011011 * 2^{-3}$



ÜBUNG: Gleitkommazahlen

Geben Sie zu folgenden dezimalen Gleitkommazahlen die entsprechende binäre Darstellung (im Floatingpointformat nach IEEE Standard 754) an:

12.75_D , -128.8_D

Geben Sie zu folgenden Gleitkommazahlen (im Floatingpointformat nach IEEE Standard 754) die entsprechenden dezimalen Gleitkommazahlen an:

1 01111011 000000000000000000000000_B

0 10000100 001011000000000000000000_B



..... Gleitkommazahlen : Zahlenbereich

Zahlenbereich:

größte normalisierte Zahl (Anm.: e darf nicht 0 oder 255 sein !)

$$\begin{aligned}0 \text{ } 11111110 \text{ } 11111111111111111111111111 &= 2^{127} * 1,999999881 \\&= 3.40282347 * 10^{38}\end{aligned}$$

kleinste pos. normalisierte Zahl (Anm.: $e_{\text{Min}} = 1 = (-126 + \text{Exzeß})$)

$$\begin{aligned}0 \text{ } 00000001 \text{ } 00000000000000000000000000 &= 2^{-126} * 1,00000000 \\&= 1.17549435 * 10^{-38}\end{aligned}$$

Sonderfälle:

e = 0 und f = 0

→ Zahl ist +/-0

e = 0 und f ≠ 0

→ Zahl ist denormalisiert

e = 255 und f = 0

→ Zahl ist +/- unendlich

e = 255 und f ≠ 0

→ Zahl ist „not a number“ (-nan)



3.4 Zeichencodes

3.4.1 ASCII (American Standard Code for Information Interchange)

- Standardcode für Datenübertragung und Zeichendarstellung im PC-Bereich
- ISO-Code 646 (*International Organization for Standardization*)
- 8-bit-Code, davon 7-bit genutzt (128 Zeichen)

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

s. Wikipedia



oder kurz

0x00 - 0x1F: Steuerzeichen

Wichtige Steuerzeichen:

nul	null
bel	bell
bs	backspace
ht	horizontal tab
nl	newline
cr	carriage return
esc	escape

0x20 : Leerzeichen

ab 0x30: Ziffern

0, 1, 2, 3,

ab 0x41: Großbuchstaben

A, B, C, D,

ab 0x61: Kleinbuchstaben

a, b, c, d,



ÜBUNG: ASCII-Code (15)

Im Speicher steht folgendes Binärmuster (in Hexadezimaldarstellung angezeigt). Wie lautet der entsprechende ASCII-Text?

48 41 4C 4C 4F 20 31 32 33



4. Befehle und Befehlscodierung

4.1 Einführung

- Maschinenbefehl:**
- Ein im Speicher abgelegtes Datenwort, dessen Bedeutung der Befehlsdecodierer der CPU ermittelt (\rightarrow *Fetch-Decode-Cycle*).
 - Entsprechend der ermittelten Bedeutung steuert das Steuerwerk die an der Befehlsausführung beteiligten Komponenten (Speicher, Register, Alu) an (\rightarrow *Execute-Cycle*).

Maschinenbefehle werden unmittelbar von der Prozessorhardware ausgeführt.

Da Hardware sehr teuer ist, stellen Maschinenbefehle lediglich Grundfunktionen für die Verarbeitung einfacher Datentypen (z.B. Zeichen, ganze Zahlen) bereit.

Komplexere Funktionen müssen aus diesen Grundfunktionen zusammengesetzt werden (\rightarrow Software).



4.2 Minimal notwendiger Befehlssatz

Typ	Befehl	Quelle	(Quelle)/Ziel
Kopierfunktionen	Verschiebe Inhalt	Konstante Register Speicher	Register Speicher
Arithmetik	Addiere Subtrahiere	Konstante Register Speicher	Register
Bitoperationen	AND OR XOR NOT	Konstante Register Speicher	Register
	Schiebe Rotiere		Register
unbedingte /bedingte Sprungbefehle	Springe nach .. Springe nach .. wenn ..		



4.3 Adressierungsarten

4.3.1 Warum sind verschiedene Adressierungsarten notwendig?

Angenommen jemand möchte Daten von einer Datenquelle zu einem Datenziel übermitteln, so könnten folgende Fälle vorkommen

- die Daten sind zur Programmierzeit bekannt

Beispiel: Konstanten

- die Daten ändern sich im Programmverlauf aber der Datenort ist fest und zur Programmierzeit bekannt

Beispiel: Variablen

- der Datenort ändert sich zur Laufzeit

Beispiel: zur Laufzeit gelesene/angelegte Datenfelder



4.3.2 Basisadressierungsarten

Konstant (beim ARM : *immediate*) → Zahl zur Programmierzeit bekannt.

Beispiel: "Lade eine 25 ins Register A,"

Anm.: beim ARM-Prozessor nur eingeschränkt verfügbar

Register

Beispiel: "Lade Inhalt von Register A in das Register B

Direkt → nur die Adresse ist fest und ist zur Programmierzeit bekannt.

Beispiel: "Lade Inhalt von Speicherzelle 1000 in das Register A."

Anm.: beim ARM-Prozessor nicht verfügbar

Indirekt → die Adresse ändert sich zur Laufzeit

Beispiel: "Lade Inhalt des Speichers, auf den Register A zeigt (d.h. dessen Adresse in Reg. A steht), in das Register B"



4.3.3 RTL – Notation für maschinennahe Operationen

RTL (*Register transfer language*) beschreibt Computer-Operationen in einer Algebraähnlichen Form. RTL wird im folgenden als Notation verwendet.

[A]	bedeutet	Inhalt des Registers A
[M(1020)]	bedeutet	Inhalt der Speicherstelle mit der Adresse 1020
←	bedeutet	„wird kopiert nach“ (von rechts nach links)

Beispiele:

- [A] ← 1 Der konstante Wert 1 wird in das Register A kopiert.
- [A] ← [M(1000)] Der Inhalt der Speicherstelle 1000 wird in das Register A kopiert (direkte Adressierung).
- [A] ← [M([C])] Der Inhalt derjenigen Speicherstelle, deren Adresse in Register C steht, wird in das Register A kopiert (indirekte Adressierung).
- [B] ← [B] + 1 Der Inhalt des Registers B wird um eins erhöht.



ÜBUNG: Register Transfer Language

1. Für den gegebenen Speicherauszug sind die Werte der folgenden RTL-Ausdrücke zu bestimmen.

Alle Speicheradressen und –werte sind Dezimalzahlen.

- a. $[M(2)]$
- b. $[M([M(1)])]$

Adr.	Wert
0	12
1	3
2	7
3	4
4	8
5	2

2. Angenommen die direkte Adressierung steht nicht zur Verfügung. Wie könnte man den folgenden Befehl ersetzen (z.B. beim ARM)?

$[A] \leftarrow [M(1000)]$

Anm.: Direkte Adressierung



ÜBUNG: Register Transfer Language 2

Gegeben ist der nebenstehende Speicherauszug.

Es soll die Summe über alle Elemente einer Tabelle berechnet werden.

Die Startadresse der Tabelle steht unter Adr. 1000.

Die Endadresse der Tabelle steht unter Adr. 1001.

Das Ergebnis soll unter Adr. 1002 abgelegt werden.

Schreiben Sie ein entsprechendes RTL-Programm

- mit direkter Adressierung,
- ohne direkte Adressierung.

Es stehen die Register A ... H zur Verfügung.

	Adr.	Wert
<i>StartAdr</i>	1000	5000
<i>EndAdr</i>	1001	5999
<i>Erg</i>	1002	0
	1004	...

...
5000	-87
5001	128
5002	-12
5998	366
5999	-97
6000	...



4.4 Modell eines sehr einfachen Digitalrechners

4.4.1 Ziel

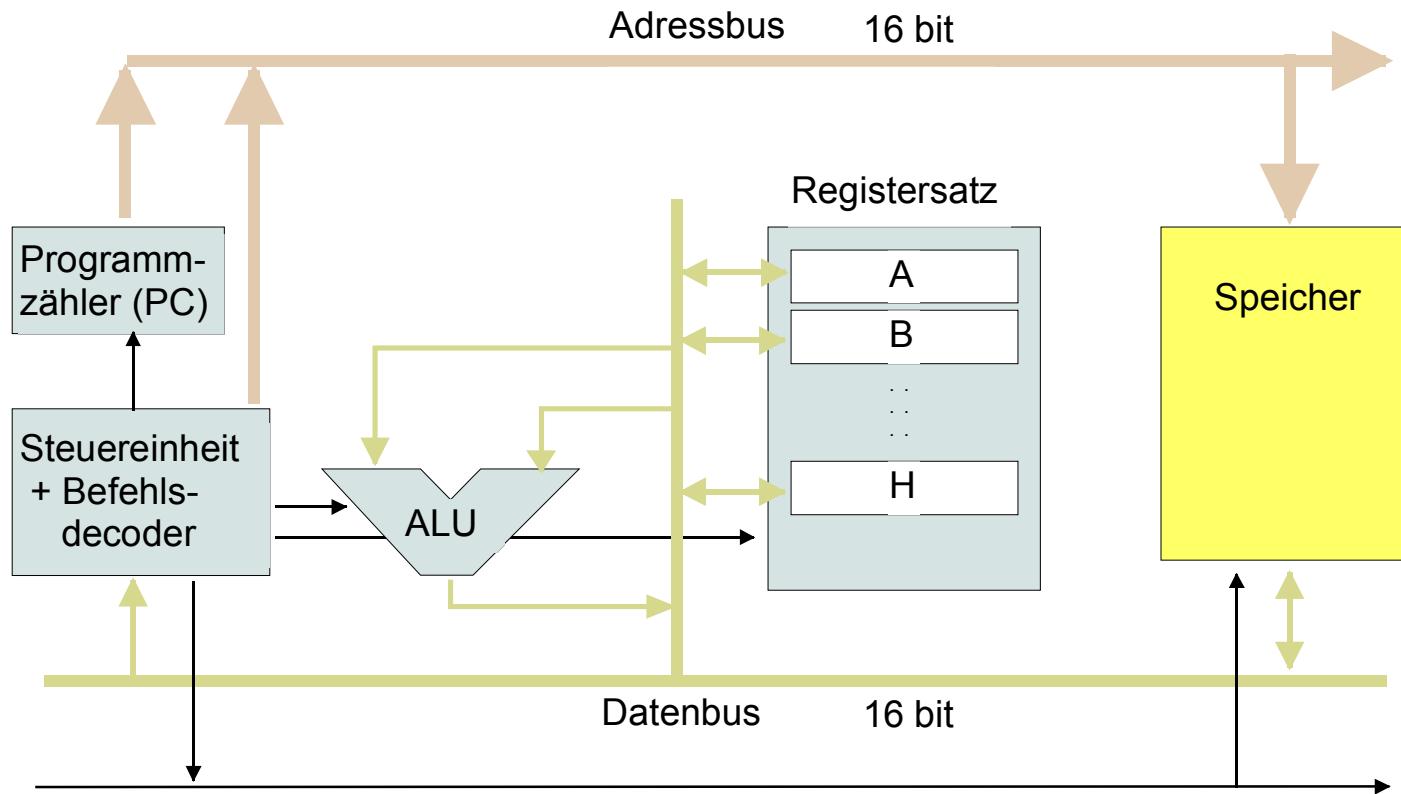
- Prinzip der Befehlscodierung
- einfache Adressierungsarten
- Zusammenhang zwischen Maschinensprache und Assembler

4.4.2 Eigenschaften des einfachen Modellrechners

- Zwei-Adress-Maschine (*Operation Ziel, Quelle*)
- "Big-Endian"
- 16-bit Wortbreite für Daten und Adressen
- Acht 16-bit-Register (A, B,H)
- 2 x 1-bit-Flagregister (Carry-Flag, Zero-Flag)



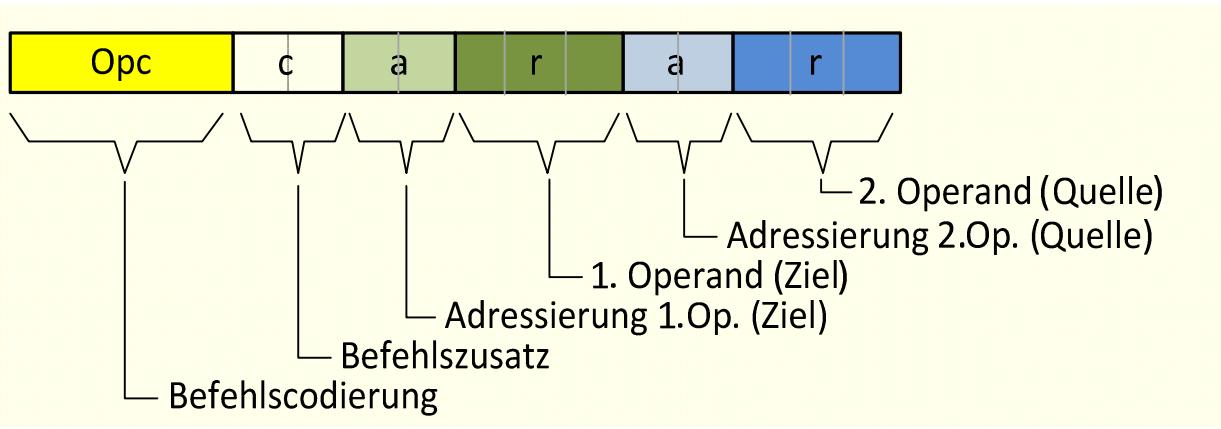
Interner Aufbau (von-Neumann-Architektur)





4.4.3 Aufbau des Operationscodes

- Befehlscodierung in ein oder zwei 16-bit-Worten
- Das zweite Wort enthält bei Bedarf eine Konstante oder eine Adresse.

**Befehlszusatz**

c	Mn	Bed.
00	Z	= 0
01	NZ	= 0
10	C	carry
11	NC	not carry

Mnemo	Opcode	Bedeutung
LD	0001	Lade
ST	0010	Speichere
CP	0011	Vergleiche
ADD	0100	Addiere
SUB	0110	Subtrahiere
...	...	
JP	1110	Springe

Adressierungsarten

a	Bedeutung
00	Konstante (2 Worte)
01	Register (1 Wort)
10	Dir. Adress. (2 Worte)
11	Ind. Adress. (1 Wort)

Register

r	Register
000	A
001	B
....
111	H



zur Notation (einfacher Assembler)

Beschreibung von Befehlen: *Befehlsmnemo Ziel, Quelle*

Beispiel: LD B, A [B] \leftarrow [A]

Beschreibung von Konstanten: #*Zahl*

Beispiel: LD A, #25 [A] \leftarrow 25

Beschreibung des Zahlenformates:

in welchem die Konstante oder
Adresse angegeben werden soll

dezimal:

Dezimalzahl

hexadezimal:

0xHexadezimalzahl

binär:

0bBinärzahl

ASCII:

'Zeichen'

Beschreibung der Quelladressierungsart (Anm.: hier Ziel immer Register B):

Register: LD B, A @ [B] \leftarrow [A]

Konstant: LD B, #0x25 @ [B] \leftarrow 37 (=Hex: 25)

Direkt: LD B, 0x1000 @ [B] \leftarrow [M(0x1000)]

Indirekt: LD B, (A) @ [B] \leftarrow [M([A])]



Beispiel : Assemblieren eines kleinen Programms

s. Tafel

LD A, #0x100	[A] \leftarrow 0x100	Lade Register A mit Konstante 0x100 Opcode 0001 00 01 000 00 000 (0x1100) 0000 0001 0000 0000 (0x0100)
SUB A, B	[A] \leftarrow [A] - [B]	Subtr. von Register A den Inhalt von Reg. B Opcode 0110 00 01 000 01 001 (0x6109)
LD B, 0x200	[B] \leftarrow [M(0x200)]	Lade Reg. B mit Inhalt des Speichers 0x200 Opcode 0001 00 01 001 10 000 (0x1130) 0000 0010 0000 0000 (0x0200)
ADD B, (A)	[B] \leftarrow [B] + [M([A])]	Addiere zu Reg. B den Inhalt des Speichers, dessen Adresse in Reg. A steht Opcode 0100 00 01 001 11 000 (0x4138)



ÜBUNG: Disassemblieren eines Programms

Gegeben ist folgender Speicherauszug (Memorydump).

Geben Sie hierzu das Assemblerprogramm an. Was sind Befehle, was Daten?

Adresse (Hex)	Inhalt (Hex)
1000	6100
1002	FF00
1004	1120
1006	0008
1008	4109
100A	6120
100C	0001
100E	E400
1010	1008
	...
	...
	...



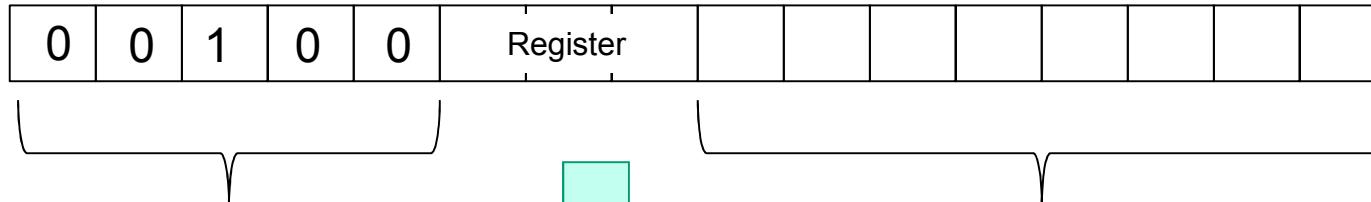
4.4.4 Erster Blick auf die Befehlscodierung beim Cortex-Prozessor

- RISC-Architektur (**reduced instruction set**), d.h.
 - wenige einfache Basisbefehle,
 - die in einem Taktschritt ausgeführt werden (einige wenige Ausnahmen).
- 2- und 3-Adressbefehle (z.B. add r1, r2 oder add r1, r2, r3)
- Befehle werden in 16 **oder** 32 bit codiert (*Thumb-2-Befehlssatz*)
 - daher: direkte Adressierung nicht möglich
 - daher: Konstanten werden nur mit 12 bit (8 + 4) codiert → 0 255 und Vielfache (4, 16, 64, 256,) davon
- alle Operationen werden in Registern ausgeführt (Load-store-Architektur)
 - Vorteil: Register haben erheblich kürzere Zugriffszeiten als ext. Speicher
- viele Register (17), davon etwa 13 Register frei nutzbar



Beispiel: 16-bit-Befehlscodierung bei Immediate-Adressierung

..... wenn Register r0-r7 **und** 8-bit-Konstante **und** keine Conditioncodes

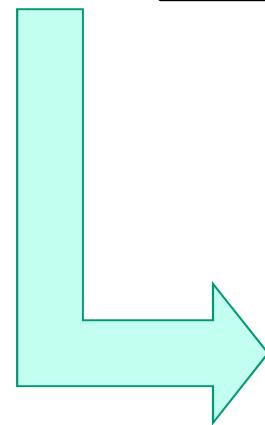


**Befehls-
code (5 bit)**

Konstante 0...255 (8 bit)

Analog auch

SUBS	00111
ADDS	00110
CMPS	00101
MOVS	00100



3 bit eins der Register r0 - r7

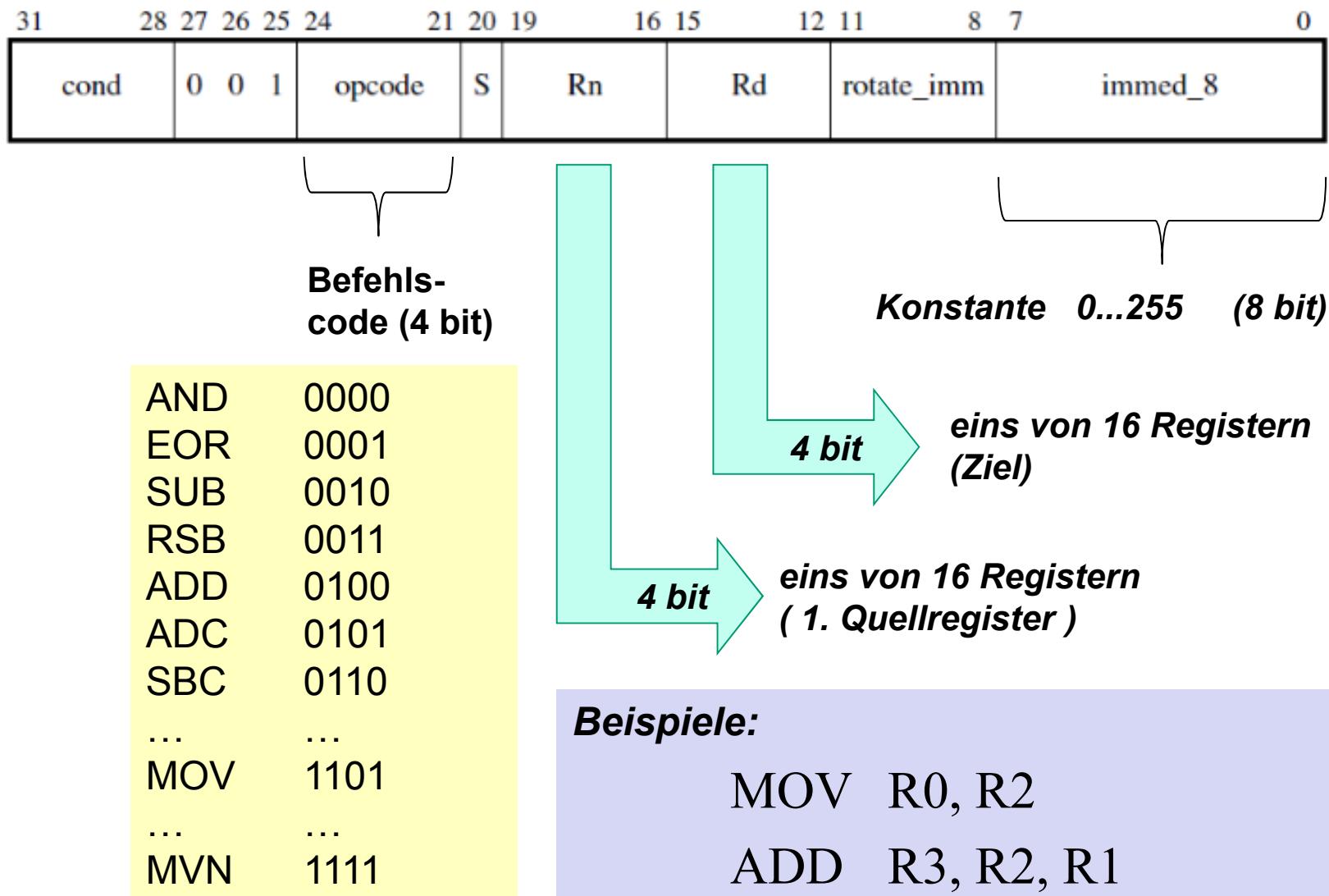
Beispiele:

MOVS R0, #0

ADDS R3, #12



Beispiel: 32-bit-Befehlscodierung bei Register-Adressierung





5. Maschinennahes Rechnen

Wie bereits gezeigt werden natürliche (*unsigned integer*) und ganze (*signed integer*) Zahlen nur auf Datentypen mit begrenzter Stellenzahl abgebildet (z.B. Byte, Word, Longword).

Daher kann beim Rechnen das Problem auftreten, dass das Ergebnis mit dem verwendeten Datentyp nicht nicht mehr darstellbar ist.

Um Folgefehler vermeiden zu können, müssen Rechenfehler erkannt und signalisiert werden.

Dabei ist sind die Fälle „vorzeichenlose“ und „vorzeichenbehaftete“ Rechnung zu unterscheiden.



5.1 Rechnerinterne Realisierung der Addition und Subtraktion

5.1.1 Binäraddition

Bitweise Addition mit Berücksichtigung des Übertrags (engl. **carry-Bit**, Abk.: c).

$$\begin{array}{r}
 x \quad 0 \ 1 \ 1 \ 0 \\
 +y \quad 1 \ 1 \ 0 \ 0 \\
 +c \quad \textcolor{red}{1} \ 1 \ 0 \ 0 \\
 \hline
 1 \ 0 \ 0 \ 1 \ 0
 \end{array}$$

5.1.2 Subtraktion = Addition des negativen Wertes (4-Bit 2-er Komplement)

Bitweise Addition des Zweierkomplements mit Berücksichtigung des carry.

Anm.: Bei der direkten Subtraktion spricht man vom „Borger“ (engl. borrow, Abk.: b).

direkte Subtraktion

$$\begin{array}{r}
 x \quad 1 \ 0 \ 1 \ 0 \\
 -y \quad 0 \ 1 \ 0 \ 0 \\
 -b \quad \textcolor{red}{0} \ 1 \ 0 \ 0 \\
 \hline
 0 \ 1 \ 1 \ 0
 \end{array}$$

oder

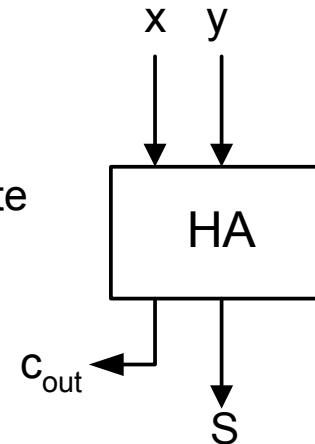
Subtraktion durch Add. des Zweierkomplements

$$\begin{array}{r}
 x \quad 1 \ 0 \ 1 \ 0 \\
 +\text{Kom}_2(y) \quad 1 \ 1 \ 0 \ 0 \\
 c \quad \textcolor{red}{1} \ 0 \ 0 \ 0 \\
 \hline
 0 \ 1 \ 1 \ 0
 \end{array}$$



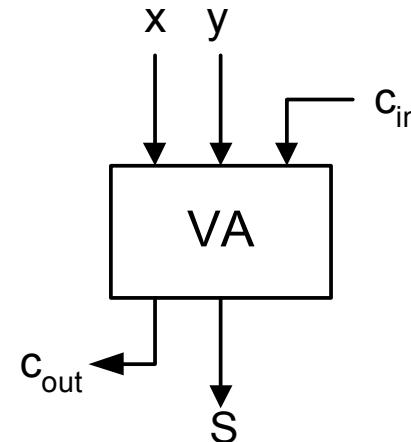
5.1.3 Elementaroperationen bei der Addition (1-bit-Addition)

- 1-bit-Addierer für die niederwertigste Stelle (= *Halbaddierer* HA)
- Übertrag (engl. carry-Bit, Abk.: c)



x	y	S	c _{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

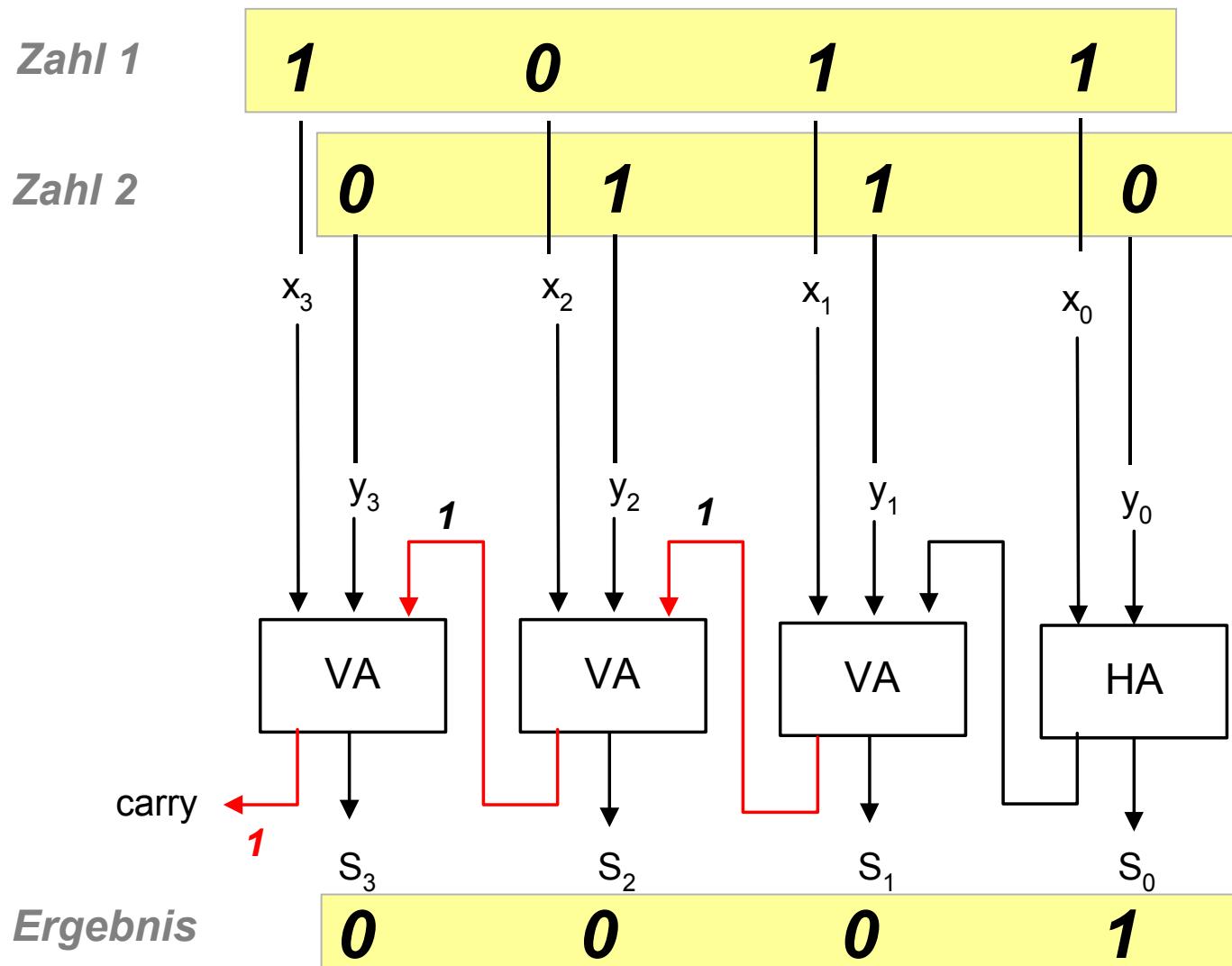
- 1-bit-Addierer für alle anderen Stellen (= *Volladdierer* VA)



c _{in}	x	y	S	c _{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



5.1.4 Beispiel: Einfaches 4-bit-Addierwerk





5.2 Addition und Subtraktion mit begrenzter Stellenzahl

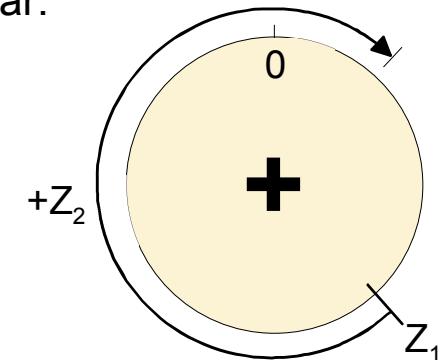
5.2.1 Vorzeichenlose Zahlen (unsigned)

Fehler 1: Bei der Addition ist das Ergebnis größer als darstellbar.

Beispiel: (4-bit-Zahlen)

$$\begin{array}{r} 0110 + 1100 \\ 6 \quad + 12 \end{array}$$

$$\begin{array}{r} x \quad 0110 \\ +y \quad 1100 \\ +c \quad 1100 \\ \hline 10010 \end{array} \quad \begin{array}{l} 6 \\ +12 \\ \hline 2 \end{array}$$

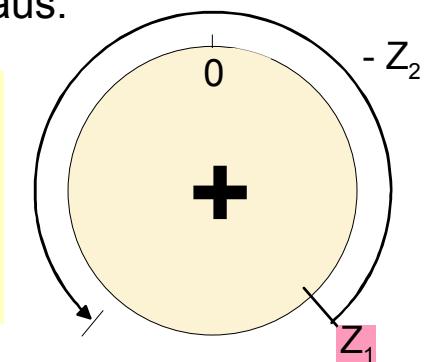


Fehler 2: Bei der Subtraktion kommt ein negatives Ergebnis heraus.

Beispiel: (4-bit-Zahlen)

$$\begin{array}{r} 0110 - 1100 \\ 6 \quad - 12 \end{array}$$

$$\begin{array}{r} x \quad 0110 \\ +(-y) \quad 0100 \\ c \quad 0100 \\ \hline 01010 \end{array} \quad \begin{array}{l} 6 \\ +(-12) \\ \hline 10 \end{array}$$



Für vorzeichenlose Zahlen gilt: Man erkennt einen **Rechenfehler** daran, dass nach der Addition **carry=1** ist, bzw. nach der Subtraktion **carry=0** ist.

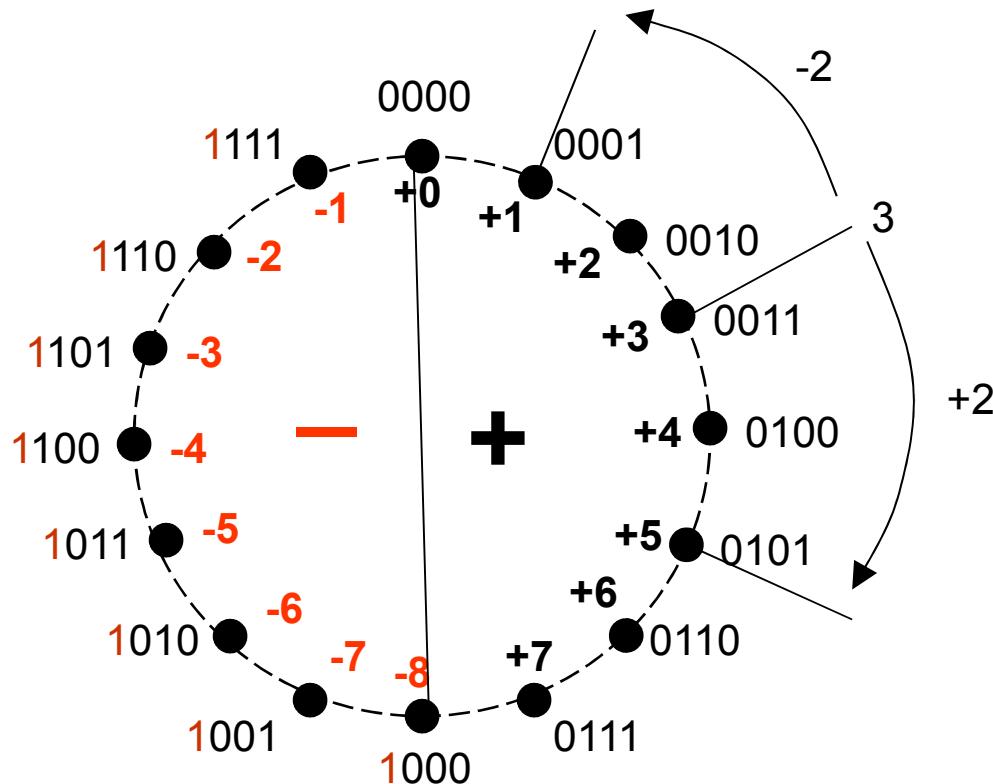


5.2.2 Vorzeichenbehaftete Zahlen (signed)

5.2.2.1 Grundgedanke (z.B. hier 4-bit-Zweierkomplement)

Fortschreiten im Uhrzeigersinn: → Addieren (+ pos. Zahl bzw. – neg.Zahl)

Fortschreiten geg. Uhrzeigersinn: → Subtrahieren (– pos. Zahl bzw. + neg.Zahl)





5.2.2.2 Überlegungen zur Addition/Subtraktion im (4-bit-)Zweierkomplement

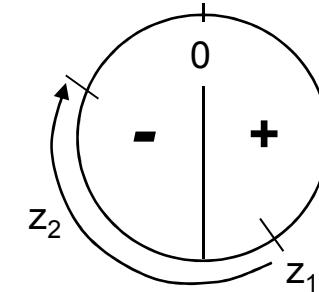
Fehler 1: pos.Zahl + pos.Zahl = neg.Zahl bzw. pos.Zahl - neg.Zahl = neg.Zahl

Beispiel:

$$0110 + 0111 \rightarrow \\ 6 + 7$$

x	0	1	1	0		6
+y	0	1	1	1		+ 7
+c	0	1	1			
	0	1	1	0	1	→ -3

Vz



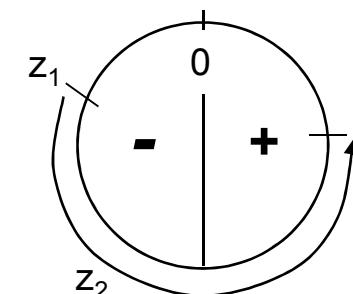
Fehler 2: neg.Zahl + neg.Zahl = pos.Zahl bzw. neg.Zahl - pos.Zahl = pos.Zahl

Beispiel:

$$1010 - 0111 \rightarrow \\ -6 - 7$$

x	1	0	1	0		- 6
+ (-y)	1	0	0	1		- (+7)
c	1	0	0	0		
	1	0	0	1	1	→ +3

Vz



Für vorbehaftete Zahlen gilt: Das carry signalisiert hier die Fehler nicht !
Der Fehler wird hier mit **Overflow-Flag** (v) signalisiert !

$$v = c_n \text{ XOR } c_{n-1}$$



5.2.2.4 Zusammenfassung

- Bei Addition und Subtraktion vorzeichenbehafteter Zahlen können Überläufe auftreten. Das Ergebnis ist dann falsch (Zahlenbereichsüberschreitung) !
- Überläufe lassen sich jedoch feststellen.
- Es gibt keinen Unterschied zwischen der vorzeichenlosen/-behafteten Rechnung !

Operation	Überlauf möglich ?	Auswirkung	Beispiel
Pos. + Pos.	Ja	Vorzeichenwechsel Neg. Ergebnis	$5 + 5$
Neg. + Neg.	Ja	Vorzeichenwechsel Pos. Ergebnis	$-5 + (-5)$
Pos. + Neg.	Nein	immer richtig	$5 + (-5)$
Neg. – Pos.	Ja	Pos. Ergebnis	$-5 - (+5)$
Pos. – Neg.	Ja	Neg. Ergebnis	$5 - (-5)$
Neg. – Neg.	Nein	immer richtig	$-5 - (-5)$
Pos. – Pos.	Nein	immer richtig	$5 - (+5)$



ÜBUNG: Addition und Subtraktion von Zahlen mit begrenzter Stellenzahl

Berechnen Sie die folgenden 8-stelligen Binärzahlen:

		vorzeichenlos	vorzeichenbehaftet
a)	00100010 + 00111110	34 + 62	34 + 62
b)	00110100 + 01100010	52 + 98	52 + 98
c)	00100111 - 01000101	39 – 69	39 – 69
d)	01001001 - 00010110	73 – 22	73 – 22
e)	11001100 - 01001110	204 – 78	-52 – 78
f)	11110100 - 00100010	244 – 34	-12 – 34

Fall A) Angenommen die Zahlen werden als vorzeichenlos betrachtet.
Ist das Ergebnis richtig?

Fall B) Angenommen die Zahlen werden als vorzeichenbehaftet betrachtet
(8-bit-Zweierkomp.). Ist das Ergebnis richtig?



5.3 Binäre Multiplikation

5.3.1 *Multiplikation mit 2*

Für Binärzahlen gilt: Multiplikation mit 2 = um eine Stelle nach links schieben

verdeutlichendes Beispiel: $45 * 2 = 90$

45

Wertigkeit	128	64	32	16	8	4	2	1
Binärzahl	0	0	1	0	1	1	0	1

$45 * 2$



Wertigkeit	128	64	32	16	8	4	2	1
Binärzahl	0	1	0	1	1	0	1	0



5.3.2 Multiplikation von beliebigen Binärzahlen durch Schieben und Addieren

