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 !

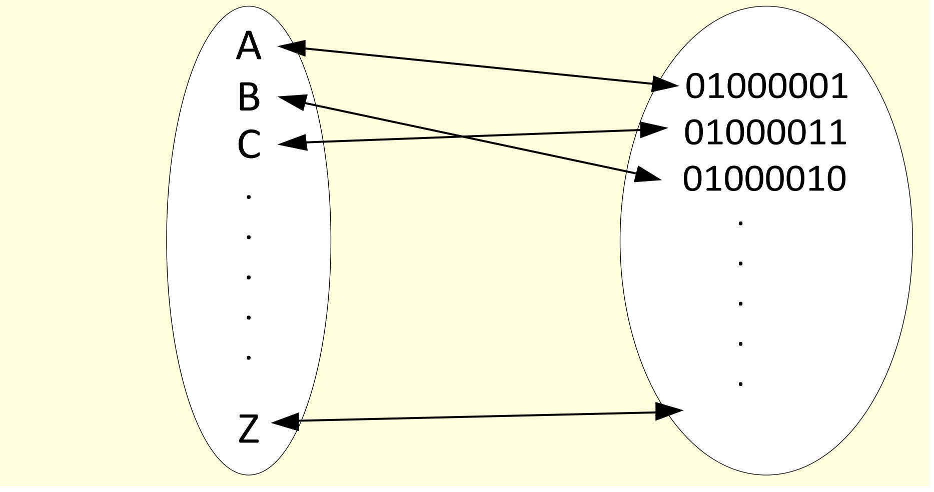
3.2 Codierung

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 S*tellenwertsysteme, 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

Einer
Zehner
Hunderter
Tausender

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: $1010_B = \mathbf{1} * 2^3 + \mathbf{0} * 2^2 + \mathbf{1} * 2^1 + \mathbf{0} * 2^0 = 10_D$

Wertigkeit 1
Wertigkeit 2
Wertigkeit 4
Wertigkeit 8

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

$$\sum_{i=0}^{n-1} a_i \cdot 16^i$$

mit

n	:	Stellenzahl der Binärziffer
a_i	:	Mantisse der Stelle i
16^i	:	Wertigkeit der Stelle i

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

Wertigkeit 1

Wertigkeit 16

Wertigkeit 256

Wertigkeit 4096

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

Umrechnung zwischen Zahlensystemen

Für die Umrechnung zwischen Zahlensystemen werden im Folgenden drei Verfahren angewendet.

1. Addition von Potenzen

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

mit

n	:	Stellenzahl der Binärziffer
a_i	:	Mantisse der Stelle i
b^i	:	Wertigkeit der Stelle i

2. Horner-Schema

$$(\ldots((a_{n-1} \cdot b \ + a_{n-2}) \cdot b + a_{n-3}) \cdot b \ldots + a_1) \cdot b + a_0$$

Beispiel: 0x23F → Dezimalschreibweise

3. Modulo-Division

Beispiel: 23 → binär

ÜBUNG: Zahlenumwandlung Dezimal ⇔ Binär (1)

Wandeln Sie folgende Dezimalzahlen in Binärzahlen um:

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

41, 221

ÜBUNG: Zahlenumwandlung Hexadezimal ⬄ 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)

s. Tafel

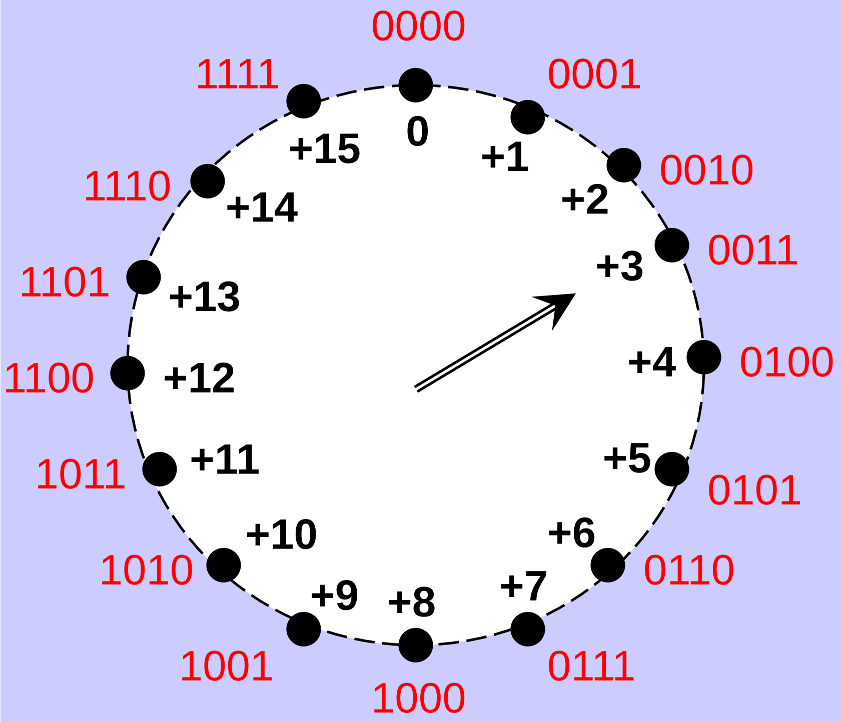
Innerhalb von Rechnern werden Binärzahlen (i.Allg.) mit einer vorgegebenen Zahl von Stellen n codiert (z.B. 8 Bit, 16 Bit, 32 Bit).

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



Beispiel:
Darstellung vorzeichenloser Zahlen im 4-Bit-Format.

3.3.3 Vorzeichenbehaftete Binärzahlen mit begrenzter Stellenzahl (signed)

s. Tafel

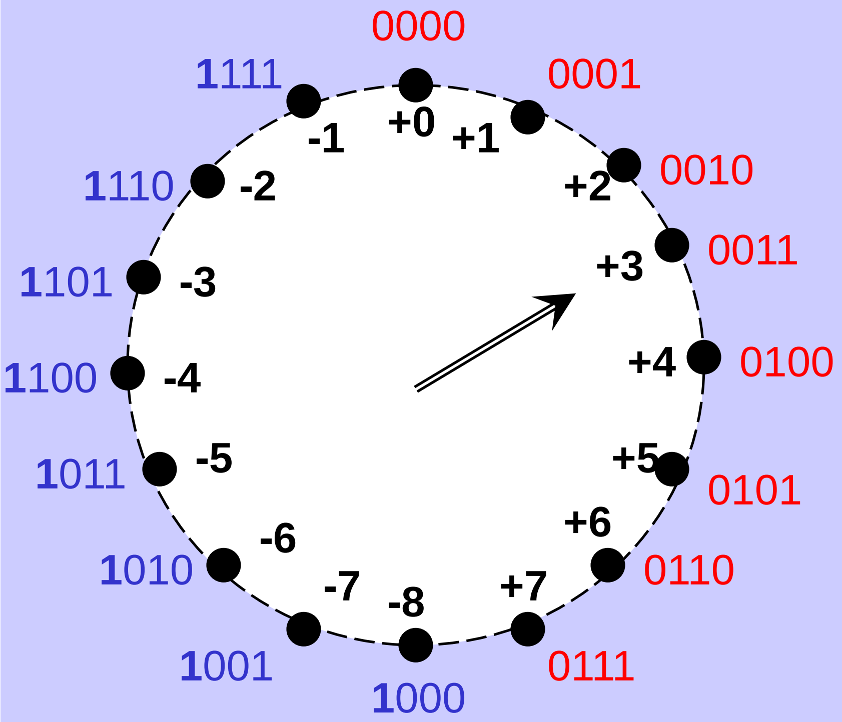
Das **Einerkomplement (Komp$_1$(z))** einer Binärzahl erhält man durch bitweises invertieren der Binärzahl z.

Das **Zweierkomplement (Komp$_2$(z))** erhält man durch: Komp$_1$(z) + 1

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

Der darstellbare Zahlenbereich ist dann:

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



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

verschoben!

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 <u>n Vorkommastellen</u> und <u>m Nachkommastellen</u> gilt somit:

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

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

In Rechnern ist die <u>Stellenzahl</u> meist <u>begrenzt</u> (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

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 (→ *Fetch-Decode-Cycle*).
- Entsprechend der ermittelten Bedeutung steuert das Steuerwerk die an der Befehlsausführung beteiligten Komponenten (Speicher, Register, Alu) an (→ *Execute-Cycle*).

Maschinenbefehle werden unmittelbar von der Prozessorhardware ausgeführt.

Maschinenbefehle enthalten meist: ein **Befehl** und **Daten** (Operator u. Operand(en))

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 (→ Software).

4.2 Minimal notwendiger Befehlssatz

Typ	***Befehl***	***Quelle***	***(Quelle)/Ziel***
Kopierfunktionen	**Verschiebe Inhalt** **MOV, LDR, STR,…**	Konstante Register Speicher	Register Speicher
Arithmetik	**Addiere ADD** **Subtrahiere SUB**	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 <u>Datenquelle</u> zu einem <u>Datenziel</u> übermitteln, so könnten folgende Fälle vorkommen

- die <u>Daten sind zur Programmierzeit</u> bekannt
  <u>Beispiel</u>: Konstanten

- die Daten ändern sich im Programmverlauf aber
  der <u>Datenort ist fest und zur Programmierzeit</u> bekannt
  <u>Beispiel</u>: Variablen

- der Datenort ändert sich zur Laufzeit
  <u>Beispiel</u>: 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

Einschub nur zur Info

RTL (*Register transfer language*) beschreibt Computer-Operationen in einer Algebra-ähnlichen 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] ← [M(1000)] # Anm.: Direkte Adressierung

Einschub nur zur Info

ÜBUNG: Register Transfer Language 2

Einschub nur zur Info

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
a) mit direkter Adressierung,
b) ohne direkte Adressierung.

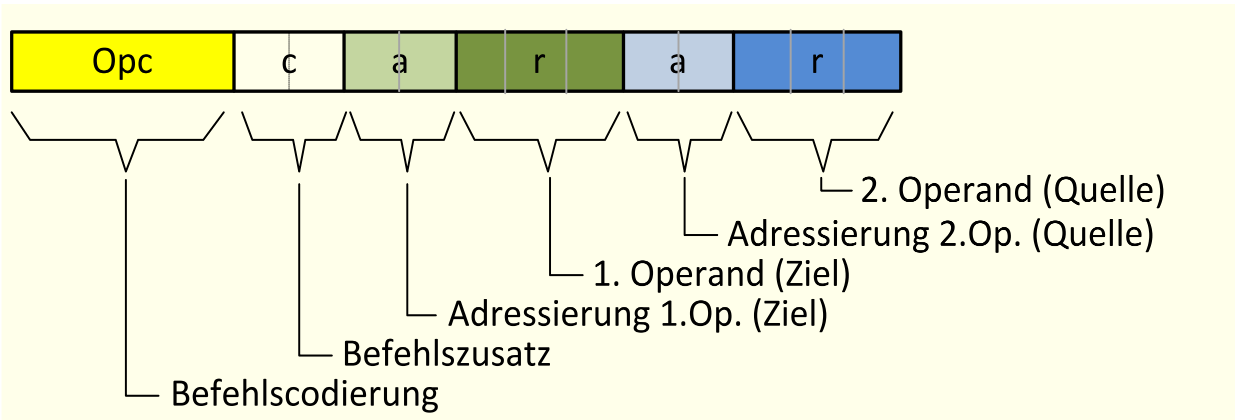
Es stehen die Register A …. H zur Verfügung.

	Adr.	Wert
StartAdr	**1000**	5000
EndAdr	**1001**	5999
Erg	**1002**	0
	1004	…

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

4.4.3 Aufbau von Operationscodes

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



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

Einschub nur zur Info

zur Notation (einfacher Assembler)

Beschreibung von Befehlen: *Befehlsmnemo Ziel, Quelle*

Beispiel: LD B, A [B] ← [A]

Beschreibung von Konstanten: *#Zahl*

Beispiel: LD A, #25 [A] ← 25

Beschreibung des Zahlenformates:
in welchem die Konstante oder Adresse angegeben werden soll

dezimal:	*Dezimalzahl*
hexadezimal:	0x*Hexadezimalzahl*
binär:	0b*Binärzahl* / 2_*Binärzahl*
ASCII:	‘*Zeichen*‘

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

Register:	LD B, A	@ [B] ← [A]	
Konstant:	LD B, #0x25	@ [B] ← 37	(=Hex: 25)
Direkt:	LD B, 0x1000	@ [B] ← [M(0x1000)]	
Indirekt:	LD B, (A)	@ [B] ← [M([A])]	

Einschub nur zur Info

Beispiel : *Assemblieren eines kleinen Programms*

s. Tafel

Einschub nur zur Info

LD A, #0x100	[A] ← 0x100	Lade Register A mit Konstante 0x100
		Opcode 0001 00 01 000 00 000 (0x1100)
		0000 0001 0000 0000 (0x0100)
SUB A, B	[A] ← [A] - [B]	Subtr. von Register A den Inhalt von Reg. B
		Opcode 0110 00 01 000 01 001 (0x6109)
LD B, 0x200	[B] ← [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] ← [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
	...
	...
	...

Einschub nur zur Info

4.4.4 Erster Blick auf die Befehlscodierung beim Cortex-Prozessor

- RISC-Architektur (**r**educed **i**nstruction **s**et **c**omputer), 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

0	0	1	0	0	Register								

Befehls-code (5 bit)

Konstante 0...255 (8 bit)

3 bit eins der Register r0 - r7

Analog auch

SUBS 00111
ADDS 00110
CMPS 00101
MOVS 00100

Beispiele:

MOVS R0, #0

ADDS R3, #12

S : Flags setzen
: Konstante, direkt

Beispiel: 32-bit-Befehlscodierung bei Register-Adressierung

31–28	27	26	25	24–21	20	19–16	15–12	11–8	7–0
cond	0	0	1	opcode	S	Rn	Rd	rotate_imm	immed_8

Befehls-code (4 bit)

Konstante 0...255 (8 bit)

4 bit ***eins von 16 Registern (Ziel)***

4 bit ***eins von 16 Registern (1. Quellregister)***

AND	0000
EOR	0001
SUB	0010
RSB	0011
ADD	0100
ADC	0101
SBC	0110
…	…
MOV	1101
…	…
MVN	1111

Beispiele:

MOV R0, R2

ADD R3, R2, R1

ERGÄNZUNG / EINSCHUB

6.4 Speicherformate des ARM-Cortex-M4

Byte-Maschine : ***Jedes Byte hat eine eigene Adresse.***

Datentypen : ***Byte, Halbwort (2 Byte), Wort (4 Byte).***

Little Endian : ***Beide Modi (big endian / little endian) sind möglich.***

<u>Üblicherweise</u> wird der ARM im <u>little endian</u> mode betrieben.

→ Das LSB (least significant byte) eines Halbwortes / Wortes liegt auf der kleinsten Adresse.

Aligned : ***Halbworte (2 Byte) müssen auf Halbwortgrenzen (= durch 2 teilbare Adressen) beginnen.***

Worte (4 Byte) müssen auf Wortgrenzen (= durch 4 teilbare Adressen) beginnen.

Außnahmen möglich (LDR, STR), aber meist langsamer.

6.5 Register des ARM-Cortex-M4

6.5.1 Die Register r0 – r15

ERGÄNZUNG / EINSCHUB

16 x ***Register r0….r15 (alle 32 bit)***
im Standardmodus für Anwendungsprogramme (User mode)

r0…r12 : ***frei verwendbar (bedingt r11, r12)***

r13 : ***Stackpointer (SP)***

- ***→ per Konvention festgelegt***
- ***→ z.B. für die Parameterübergabe bei Unterprogrammaufrufen***

r14 : ***Link Register (LR)***

- ***→ speichert die Rücksprungadresse bei Unterprogrammaufrufen***

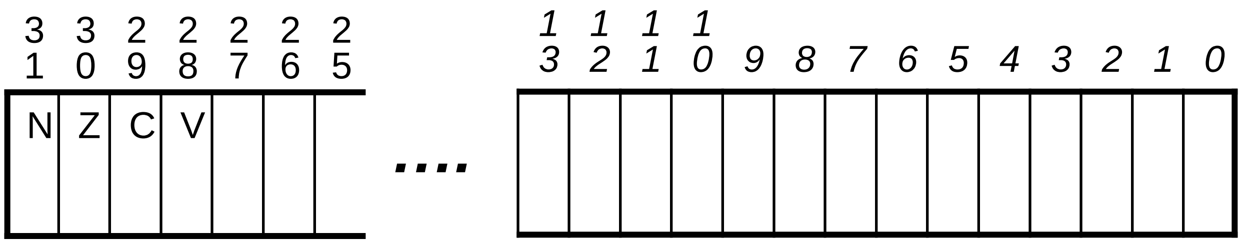
r15 : ***Program Counter (PC)***

- ***→ Bits 0 und 1 undefiniert (wg. 32-bit-Alignment)***
- ***→ enthält die Adresse des aktuell ausgeführten Befehls***
- ***→ wird nach jedem Befehl um 4 erhöht***

+condition code register, xPSR

ERGÄNZUNG / EINSCHUB

6.5.2 Statusregister



N (Negative-Flag) : enthält das MSB eines Ergebnisses, d.h. 0=pos. 1=neg.

Z (Zero-Flag) : wird gesetzt, wenn das Ergebnis 0 ist.

C (Carry-Flag) : Übertrag arith. Operationen

V (Overflow-Flag) : zeigt einen Overflow bei signed-integer-Operationen an.

Alle anderen Flags spielen in dieser Vorlesung/Praktikum keine Rolle.

6.7 Grundsätzliches zum Befehlsaufbau und zur Notation

ERGÄNZUNG / EINSCHUB

6.7.1 Anmerkung zur verwendeten Assemblernotation

Für die mnemonische Beschreibung von ARM-Maschinenprogrammen gibt es verschiedene Notationen, die sich zum Teil erheblich voneinander unterscheiden.

Im Rahmen dieser Vorlesung wird die Notation der Keil-Entwicklungsumgebung (μVision) verwendet !!

ERGÄNZUNG / EINSCHUB

6.7.2 Befehlsaufbau bei einfachen Befehlen

Befehlsaufbau bei 2-Operanden-Befehlen:

Syntax: ***Operator*** *Ziel, Quelle*

Beispiel: mov r5, r4 `; [r5] ← [r4]`

Befehlsaufbau bei 3-Operanden-Befehlen:

Syntax: ***Operator*** *Ziel, Operator_1, Operator_2*

Beispiel: add r5, r4, r3 `; [r5] ← [r4]+[r3]`

6.7.3 Angabe von Konstanten

ERGÄNZUNG / EINSCHUB

Binäre numerische Konstanten

Syntax	2_ { binäre Ziffer }		
Beipiel:	mov r0, #2_01000001	[r0] ← 65 (=0x41)	

Dezimale numerische Konstanten

Syntax	DezimalzifferOhneNull { Dezimalziffer }	
Beipiel:	mov r0, #65	[r0] ← 65 (=0x41)

Hexadezimale numerische Konstanten

Syntax	0x { hexadezimale Ziffer }	
Beipiel:	mov r0, #0x41	[r0] ← 65 (=0x41)

Zeichenkonstanten

Syntax	‘Zeichen ‘	
Beipiel:	mov r0, #‘A‘	[r0] ← 65 (=0x41)

5. Maschinennahes Rechnen

Prequel: Schriftliche dezimale Addition z.B. 1950431 + 2961742

[3.3-2] Einschub: Rechnen in Zifferncodes

[3.3.1-2] Rechnen in Stellenwertsystemen

In *Stellenwertsystemen* können Rechenoperationen (z.B. Addition, Subtraktion) einfach manuell durchgeführt werden:

Rechnen im Dezimalsystem und im Binärsystem

Beispiel :

$$1243 = \mathbf{1} * 10^3 + \mathbf{2} * 10^2 + \mathbf{4} * 10^1 + \mathbf{3} * 10^0$$

$$100 = \mathbf{0} * 10^3 + \mathbf{1} * 10^2 + \mathbf{0} * 10^1 + \mathbf{0} * 10^0$$

$$1343 = \mathbf{1} * 10^3 + \mathbf{3} * 10^2 + \mathbf{4} * 10^1 + \mathbf{3} * 10^0$$

Einer
Zehner
Hunderter
Tausender

Beispiel:

$$1010_B = \mathbf{1} * 2^3 + \mathbf{0} * 2^2 + \mathbf{1} * 2^1 + \mathbf{0} * 2^0 = 10_D$$

$$10_B = \mathbf{0} * 2^3 + \mathbf{0} * 2^2 + \mathbf{1} * 2^1 + \mathbf{0} * 2^0 = 10_D$$

Wertigkeit 1
Wertigkeit 2
Wertigkeit 4
Wertigkeit 8

5. Maschinennahes Rechnen

Prequel: Schriftliche dezimale Addition z.B. 1950431 + 2961742

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).

```
  x       0 1 0 1 1 0    22          x      0 1 1 0     6
 +y       0 1 1 1 0 0    28         +y      1 1 0 0    12
 +c     0 1 1 1 0 0                 +c    1 1 0 0
 -------------------------          ----------------
          1 1 0 0 1 0                     1 0 0 1 0
```

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).

<u>direkte Subtraktion</u>

```
 x        1 0 1 0    10
-y        0 1 0 0    -4
-b      0 1 0 0
-----------------
          0 1 1 0
```

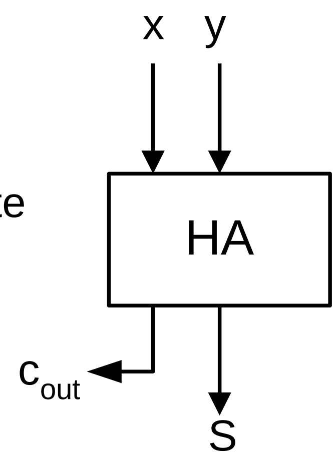
oder

<u>Subtraktion durch Add. des Zweierkomplements</u>

```
 x             1 0 1 0       10
+Kom2(y)       1 1 0 0    + (-4)
 c           1 0 0 0
--------------------------
               0 1 1 0
```

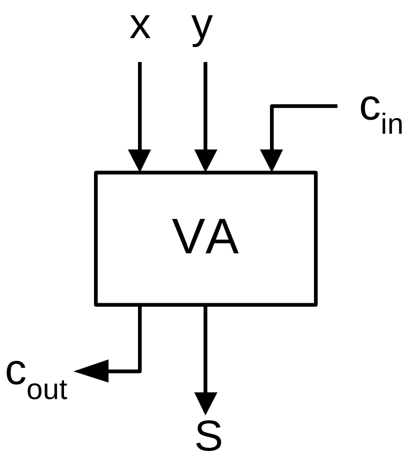
5.1.3 Elementaroperationen bei der Addition (1-bit-Addition)

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



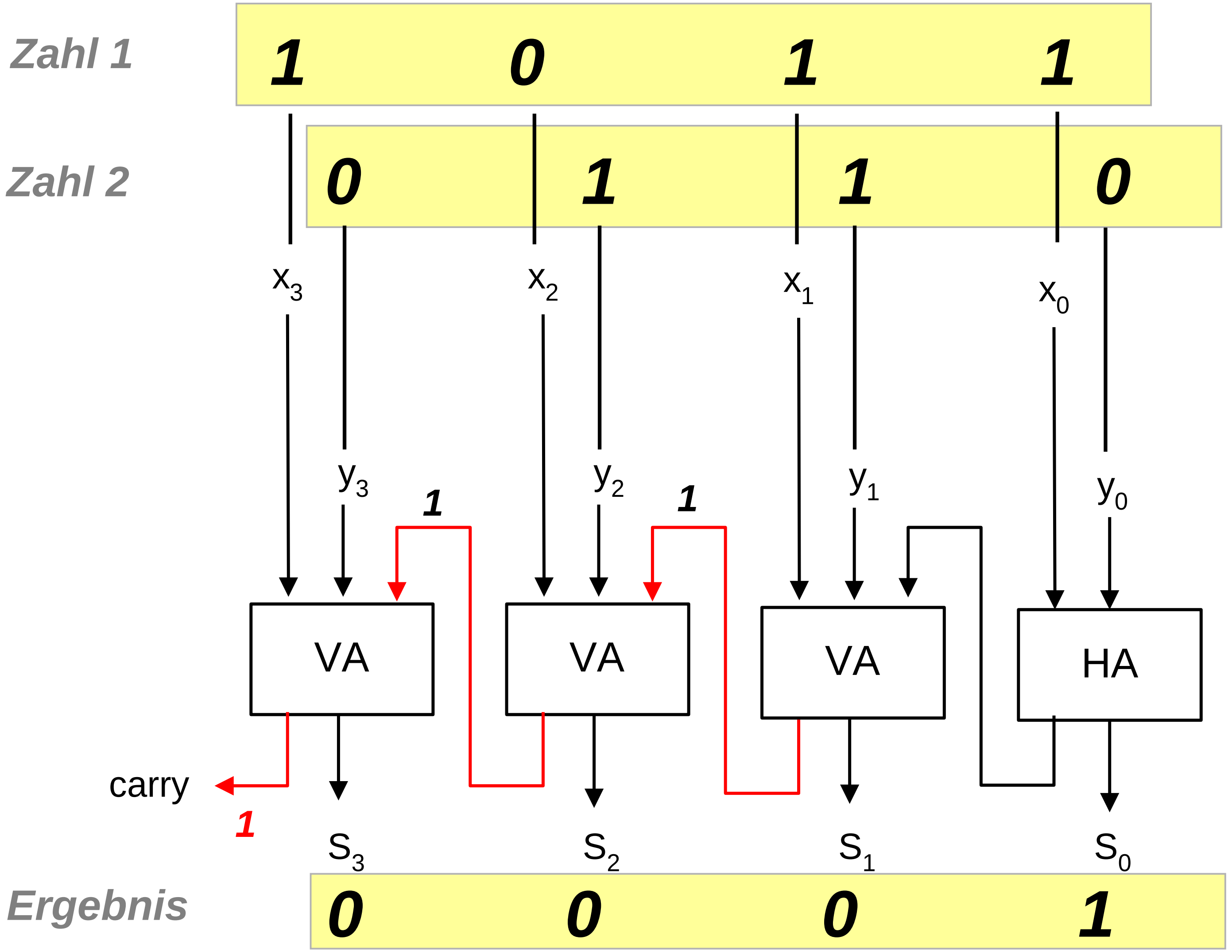
x	y	S	c_{out}
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- 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)

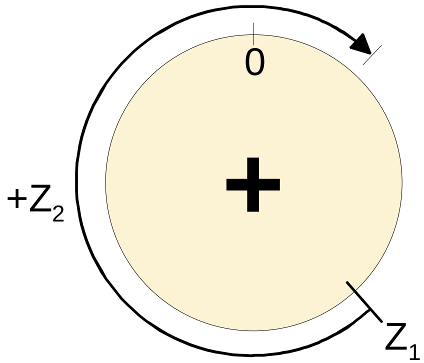
Fehler 1: Bei der Addition ist das Ergebnis größer als darstellbar.

Beispiel: (**4-bit**-Zahlen)

0110 + 1100 →

6 + 12

```
  x    0 1 1 0        6
 +y    1 1 0 0      +12
 +c  1 1 0 0
 ---------------     -----
     1 0 0 1 0   →    2
```



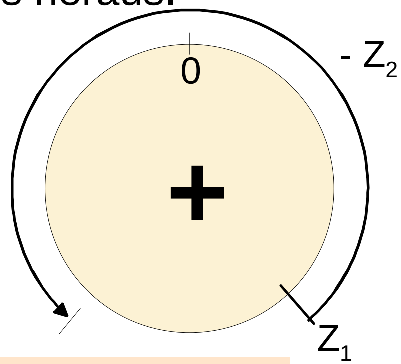
Fehler 2: Bei der Subtraktion kommt ein negatives (zu großes) Ergebnis heraus.

Beispiel: (**4-bit**-Zahlen)

0110 – 1100 →

6 – 12

```
     x     0 1 1 0         6
 +(- y)    0 1 0 0      +( -12)
     c   0 1 0 0
 -------------------    ---------
         0 1 0 1 0   →     10
```

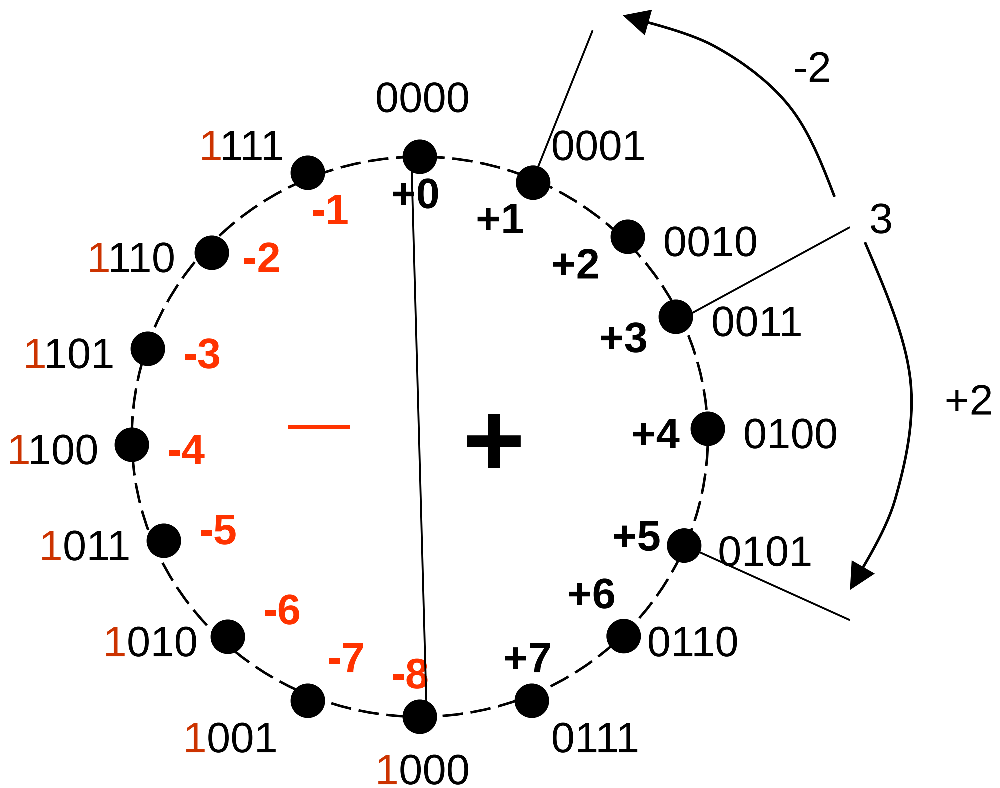


Für vorzeichenlose Zahlen gilt: Man erkennt einen Rechenfehler daran, dass nach der Addition carry=1 ist, bzw. nach der Subtraktion carry=0 ist.

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

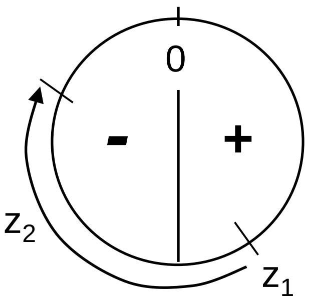
Beispiel:

0110 + 0111 →

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

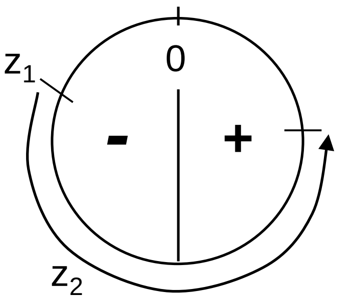
Beispiel:

1010 – 0111 →

-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 vorz.behaftete 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.
- Dies gilt für vorzeichenlose und vorzeichenbehaftete Rechnung gleichermaßen !

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?