Subtraktion von binären Zahlen
Drei Schritte zu Subtraktion:
Das Einerkomplement Das Zweierkomplement Die Subtraktion von Dualzahlen

Einerkomplement

Was ist das Komplement von Dualzahlen? Man bildet das sogenannte Einerkomplement, indem man jede Zahl durch ihr Gegenteil ersetzt, also die 0 durch die 1 und die 1 durch die 0.

01011010 wird zu 10100101
11101101 wird zu 00010010

Das Zweierkomplement

Das Zweierkomplement entspricht dem Einerkomplement, nur wird zusätzlich noch 1 addiert.
01011010 wird im Einerkomplement zu 10100101 im Zweierkomplement zu 10100110
11101101 wird im Einerkomplement zu 00010010 im Zweierkomplement zu 00010011

Konvertierung von Festkommazahlen Dez zu Bin 10,2

Vorkommastelle 10 = 1010
Nachkommastelle
0,2 * 2 = 0,4 + 0 *MSB*
0,4 * 2 = 0,8 + 0
0,8 * 2 = 0,6 + 1
0,6 * 2 = *0,2* + 1 *LSB*

Sobald es sich wiederholt kann aufgehört werden.
0, 2 = 0,0011
10,2 = 1010,00110011 ≈ 0,19921875
$\Longrightarrow$ Eine Abweichung von -0,00078125

Konvertierung von Fließkommazahlen Dez zu Bin $18{,}4_{10}$

$8_{10} = 10010_2$
$0{,}4_{10} = 0{,}011_2$

mov vs. ldr

ldr	mov	Funktion
r1, [r2]	r1, r2	speichere Wert von r2 in r1
r1, =255	r1, # 255	speichere 255 in r1
Bewegt Speicher/Register	Bewegt Register	-
32-Bit	8-Bit	-

Die Flags

Es existieren 3 Arten von Flags:

C Carry-Flag

V Overflow-Flag

Z Zero-Flag

Carry-Flag oder Borrow-flag

Definition: Zeigt an, dass die Rechenoperation den verwendeten Zahlenbereich überschreitet. Bei Subtraktion ist das negierte Carry-Flag das **Borrow-Flag**), dieses zeigt an, dass der Zahlenbereich unterschritten wurde.

Wichtig bei Addition: Carry = 1 → Bereichsüberschreitung → Fehler
Subtraktion (oder Addition e. negativen Wertes): Borrow = 1 → Bereichsüberschreitung → Fehler

```
 1111 1110
+1011 1110
 ----------
11111 110
==========
 0000 0100
```

C 1 letzter Übertrag = 1

V 0 xor letzter und vorletzter Übertrag = 1 xor 1 = 0

Overflow-Flag

Definition: Zeigt an, dass das Vorzeichenbit durch Überlauf verändert wurde (obwohl es eigentlich nicht hätte verändert werden sollen).

Wichtig bei signed: Wenn gesetzt → falsches Ergebnis.

```
 0010 1111
+0101 1001
 ----------
01111 111
 ==========
 0000 0100
```

C 0 letzter Übertrag = 0

V 1 xor letzter und vorletzter Übertrag = 0 xor 1 = 1

Zero-Flag

Definition

Wichtig bei

Die Subtraktion von Dualzahlen

Der Satz lautet: Die Subtraktion von 2 Zahlen erfolgt durch die Addition des Zweierkomplementes. Als konkretes Beispiel nehmen wir dazu die Rechnung 14-9=5.
9 ist im Dualsystem 00001001. Das Einerkomplement zu 00001001 ist 11110110. Das Zweierkomplement 11110111. Dies addieren wir nun zu 14 also 00001110.

```
 00001110
+11110111
 ========
 00000101
```

Auch hier wäre die richtige Zahl eigentlich 00000101 Übertrag 1, da wir den Übertrag jedoch nicht speichern können, bleiben wir bei 00000101 was ja der Dezimalzahl 5 entspricht.

Little-/Bigendian

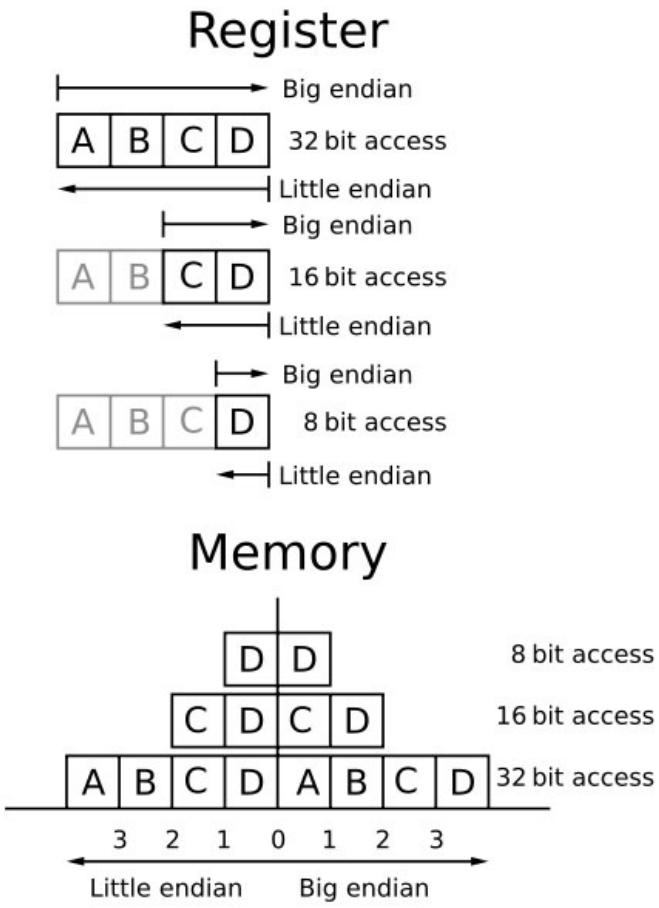

Figure 1: A simple caption

Assemblerbefehle

```
AREA MyCommonBlock, COMMON, ALIGN = 10 ; Read-Write-Data
MyCommonBlock bezeichnet die Anfangsadresse des Speicherblocks
COMMON: vom Linker mit Nullen initialisierter Speicherbereich
Alignment mit 2^10 erzeugt eine Blockgrenze bzw.   anfang   mit n * 1024

mov r0, #0x21
Lade #0x21 in Register R0:
R0
00000021

mov r1, #-10



DCB  8 Bit
DCW 16 Bit
DCD 32 Bit
```

Lösungen für Tests und ihre Vorbereitungsaufgaben

1 Lösung für Test 1:

1.1

Wie lautet die Hexadezimalzahl zur Binärzahl?

```
1001 1101 1010
   9    D   10
```

1.2

Wie lautet die Binärzahl zur Dezimalzahl 97

Lösung 0110 0001

Erklärung 1*64 + 1*32 + 1*1

1.3

Addieren Die nebenstehende 8-Bit Binärzahlen:

Ist das Carry-Flag gesetzt:
Ist das Overflow-Flag gesetzt:

```
 0111 1111
+1011 0011
```

```
   0111 1111
  +1011 0011
-------------
U 11111 1110
-------------
   0011 0010
```

Carry-Flag: ja
Overflow-Flag: nein

Erklärung Das Carry-Flag ist gesetzt dar der hinterste Übertrag auf 1 gesetzt ist. Das Overflow-Flag ist nicht gesetzt dar der hinterste und der vorhinterste Übertrag in Xor-Verbindung 0 ergibt.

1.4

Geben Sie den Dezimalwert zur Hexadezimalzahl 7D an.

Lösung 125

```
Hex $\rightarrow$           7 |       D\\
Bin $\rightarrow$ 0  1  1  1 | 1 1 0 1\\
Dez $\rightarrow$ / 64 32 16 | 8 4 / 1 = 125\\
```

1.5

Wie lautet die 8-Bit-Zweierkompliment-Darstellung zur Dezimal -97?

```
                 −97 = 1001 1111
Zweierkompliment: 97 = 0110 0001
```

Erklärung -128 + 16 + 8 + 4 + 2 + 1 = -97
→ Zweierkompliment = Binär invertieren + 0000 0001

1.6

Woran erkennt man bei der Subtraktion zweier Vorzeichenbehafteter Zahlen, ob das berechnete Ergebnis falsch ist?

Lösung Overflow-Flag:
=1 → flasch
=0 → richtig

Erklärung s.o.

1.7

Das Datenfeld...

1.8

Ab Adresse 0x1004 steht folgendes im Speciher (hex.) little endian:

```
14 21 32 A3 A7 F3 FA
```

Was steht in r1 nach folgender Sequenz?

```
mov     r0, #0x1006
ldrh    r1, [r0]
```

Lösung

2 Lösung für Test 2:

2.1

```
Bytefeld        DCB     11,'B', 0xB, 0b01000010
```

a)

```
Bytefeld        0B, 42, 11, 42
```

b)

```
ldr r1, =Bytefeld
ldrb r0, r1
```

c)

```
0x11, B
```

2.2

```
ALIGN 4
mov r0, 0xAB
```

2.3

```
ldr r0, 0x1256ABCD
```

2.4

```
bgt
```

2.5

```
bgt
```

2.6

Aufgabe:
Die **vorzeichenlose** Zahl in r0 soll durch 4 geteilt werden. Das Ergebnis soll in r1 stehen.
Geben Sie den Befehl an:

```
MOV r0,r1,ASR #2
```

Erklärung: Eine Verschiebeoperation nach **links** um 1 Bit entspricht der **Multiplikation** mit 2
und eine Verschiebeoperation nach **rechts** um 1 Bit entspricht der **Division** mit 2.
Warum # 2 statt 4? # 1 $\rightarrow$ x $\div$ 2; # 2 $\rightarrow$ x $\div$ 2 $\div$ 2 $\rightarrow$ x $\div$ 4

Nachschlagen: Kapitel 8.5.5

2.7

Aufgabe:
Das Datenfeld Var1 beginne bei Adresse 0x2000. Welcher Wert (hex.) vsteht nach Ausführung des Befehls in r0?

```
Var1    DCB     10, 'A', 0xA, '1'

                ldr r0, =Var1
```

Lösung r0 = 0x2000

Erklärung: Lade die Adresse von Var1 in r0.

Nachschlagen: Kapitel 7.5.3

2.8

Das Datenfeld Tab beginne bei Adresse 0x2000. Geben Sie die Speicherinhalte (hex.) von Adresse 0x2000 - 0x2003 an?

```
Var1    DCB     0x10, 'A', 10, '1'
```

Lösung 0x2000: 41 0A 31 10

Erklärung: 0x10 → Hexadezimal → 10
'A' → ASCII → 41
10 → Hexadezimal → A
'1' → ASCII → 41

Nachschlagen: Kapitel 7.4.3 Folie 18 → Wie werden die Sachen gespeichert?
Kapitel 6.4 → Reihenfolge im Speicher

2.9

Folgendes Datenfeld sei gegeben:

```
Var1    DCD     0x10, 0xAA12
```

Geben Sie die Assemblerbefehle an, um das erste Datenwort des Feldes Var1 nach r1 zu kopieren

```
ldr r0, =Var1 ; Arraystartadresse laden
ldr r1, [r0] ; Erstes Element des Arrays
```

Erklärung: Warum nicht mov?
mov kopiert nur ein Datenwort Syntax: MOV <wohin>, <woher,was> → Daten >
Nachschlagen von MOV: Kapitel 6.9.3

Nachschlagen: Kapitel 7.5.3 Kapitel 7.7.3

2.10

Was steht in r0 nach folgendem Befehl (hex.)?

```
ldr             r0, =0x1234ABCD
```

Lösung: r0 = 0x1234ABCD

Erklärung: Wenn nach '=' ein Hexwert kommt dann speichere den Wert. Wenn Variable, dann speichere die Adresse.
Auch hier würde mov nicht funktionieren, da 0x1234ABCD > 8 Bit

Nachschlagen: `http://www.keil.com/support/man/docs/armasm/armasm_dom1361289875065.htm`
`https://www.raspberrypi.org/forums/viewtopic.php?&t=16528`

2.11

In welchem Wertebereich muss r0 liegen, damit ein Sprung nach LOOP erfolgt? (dezimal oder hex.)

```
mov             r1, #-15
cmp             r0, r1
bge             LOOP    ;if greater or equal
```

Größer oder gleich:
Kleiner oder gleich:

Lösung: Größer oder gleich: -15
Kleiner oder gleich: 255

Erklärung: greater or equal → r1 muss >= -15 mov r1 → 8-Bit → 1 muss <= 255

Nachschlagen: bge → Kapitel 8.3.5

2.12

Was steht in r0 nach folgender Befehlssequenz (hex.)?

```
ldr                 r1 , =0xFFFFFF87
mov                 r0 , #0x78
and                 r0 , r1
```

```
r1  1111  1111  1111  1111  1111  1111  1000  0111
r0  0000  0000  0000  0000  0000  0000  0111  1000
-------------------------------------------------
r0  0000  0000  0000  0000  0000  0000  0000  0000
```

Erklärung: logisches UND nur wenn gleiche Werte in r1 und r0 stehen → 1

Nachschlagen: Kapitel 8.4.3

3 Lösung für die Vorbereitung für Test 3:

3.1

```
ldr             r0, =V1
ldr             r1, =V2
```

3.1.1 Nachschlagen:

Kapitel 10.2.2.3

3.2

```
ldr             r0, 12355
ldr             r1, 12
```

3.2.1 Nachschlagen:

Kapitel 10.2.2.3

3.3

```
Aufrufendes Programm
ldr             r0, =V1
ldr             r1, =V2
push    {r0, r1}                ; PUSH V1 und V2
bl      Binom1
add     sp, #8                  ; Stack korrigieren

Unterprogramm Binom1
push    {fp, lr}                                ; PUSH fp , fp_alt retten
mov     fp , sp                                 ; aktuelle Stackpos. merken
push    {r1-r4}                         ; PUSH,  Register retten
ldr             r0, =V1
ldr             r1, =V2
pop     {r1-r4}                         ; POP,  Register restaurieren
pop     {fp,lr}                                 ; POP  fp u. lr restaurieren
bx              lr
```

3.3.1 Nachschlagen:

Kapitel 10.2.2.3

3.4

Das Linkregister speichert die Rücksprungsadresse.

3.5

```
TabAdd
ldr             r0, Tab1
ldr             r1, 4
```

3.5.1 Nachschlagen:

Kapitel 10.2.2.3

3.6

```
Aufrufendes Programm
ldr             r0, =V1
ldr             r1, =V2
push    {r0, r1}                ; PUSH V1 und V2
bl      Binom1
add     sp, #8                  ; Stack korrigieren

Unterprogramm Binom1
push    {fp, lr}                                ; PUSH fp , fp_alt retten
mov     fp , sp                                 ; aktuelle Stackpos. merken
push    {r1-r4}                         ; PUSH,  Register retten
ldr     r2, [fp, #8]             ; [r2] V2
ldr     r1, [fp, #12]            ; [r1] V1
...
...
...
pop     {r1-r4}                         ; POP,  Register restaurieren
pop     {fp,lr}                                 ; POP  fp u. lr restaurieren
bx              lr
```

3.6.1 Nachschlagen:

Kapitel 10.2.2.3

3.7

den Stackpointer zu korrigieren.

3.7.1 Nachschlagen:

Kapitel 10.2.2.3

4 Lösung für Test 3:

Gegeben ist folgendes Programmfragment (für die Aufgaben 1-3):

```
                AREA MyData, DATA, align = 8
VarA                    DCD             0x17, 17, 0xA, 'A'
VarB                    DCB             33, -1, 0x10, 0x20, 0x30, 'a', '1'
```

4.1

```
ldr     r0,= VarA       ; Lade den Anfang von VarA in r0
ldr     r1, [r0]        ; Lade das erste Elemente von VarA in r1
ldr     r2, [r0 #4]     ; Lade das zweite Element von VarA in r2 (pre increment)
adds    r1, r1, r2      ; Addiere r1 und r2 und speichere das Ergebnis in r1
```

4.2

```
VarB = 0x2010
VarA = 0x2000
VarB - VarA = 0x2010 - 0x2000 = 0x0010  ; Anfangsadressen werden subtrahiert
```

4.3

0x2000	0x2001	0x2002	0x2003	0x2004	0x2005
0x17	00	00	00	0x11	00

Zu Beachten: align = 8 → Es wird auf 8 Byte aufgefüllt.

4.4

```
bl MyProg
```

bl (branch link) springt bedingungslos.

4.5

Speichert die Rücksprungsstelle.

```
r14
```

.

Nachschlagen: Kapitel 10.1.3.2

4.6

Der Stackpointer zeigt uns an welcher Stelle wir uns im Stack befinden.

```
r13
```

4.7

push fügt etwas dem Stack hinzu. sp (Stackpointer) wird ums eins verringert.

Nachschlagen: Kapitel 9.2

4.8

```
push    {fp, lr}
mov     fp, sp
push    {r1 - r4}        ; 8-align -> gerade Anzahl Registern
pop     {r1 - r4}        ; 8-align -> gerade Anzahl Registern
pop             {fp, lr}
bx              lr
```

Nachschlagen: Kapitel 9.3.1

4.9

```
push    {r1 - r4}
bl
...
add             sp, #16
```

4.10

Vereinfacht Zugriff auf lokale Daten.

```
r11
```

Nachschlagen: Kapitel 10.2.3.1

5 Lösung für Test 4:

5.1

```
b       StrCmp
```

Warum nicht bl? Braucht man das lr nicht?

Nachschlagen:

5.2

```
pa      = &a;
int *pa = &a;
```

Nachschlagen: Kapitel 13.13

5.3

```
pVek = Vek;
int *pVek = Vek[0];
```

Nachschlagen: Kapitel 13.13
Kapitel 13.18

5.4

```
int var = Vek[3];
int var = *pVek[3];
```

Nachschlagen: Kapitel 13.13

5.5

```
printf("%d %d %d", x, &x, px);

-> schreibe 3 Dezimalzahlen:
        x               -> Wert von x
        &x              -> Adresse von x
        px              -> Adresse auf die px zeigt -> Adresse von x

10, 2400, 2400

printf("%d %d %d", &px, *px, *ppx);

-> schreibe 3 Dezimalzahlen:
        &px     -> Adresse von px
        *px             -> Wert dessen worauf px zeigt -> Wert von x
```

```
*ppx    -> Wert dessen worauf ppx zeigt -> Wert von px -> Adresse von x
```

2000, 10, 2400

Nachschlagen: Kapitel 13.13

5.6

```
int* api[12];
```

Array mit der Laenge 12 von Pointern, die auf Integer zeigen

Nachschlagen: Kapitel 13.20

5.7

```
Laenge1 = strlen(sP1);
```

```
Laenge2 = strlen(sP2[2]);
```

```
Laenge3 = strlen(&*(sP2 + 2);
```

Nachschlagen: