Dualzahlen mit Vorzeichen

Additionen:

101	111
+110	+1111
1011	10110

Wir beschränken uns im Folgenden auf **acht**stellige Dualzahlen. Alle Aussagen gelten jedoch sinngemäß auch für beliebig große Dualzahlen. Man kann, muss aber nicht, das linke, höchstwertige Bit (wird oft auch als **MSB**, *most significant bit*, bezeichnet) zur Kennzeichnung des Vorzeichens dieser Zahl verwenden. In vielen Programmiersprachen gibt es zwei unterschiedliche 8-bit-Zahlentypen. Der oftmals *Byte* genannte Typ interpretiert das MSB nicht als Vorzeichen und kann somit die Zahlen von 0 bis 255 darstellen. Der oftmals *Shortint* genannte Typ hingegen interpretiert das MSB als Vorzeichen. Dieser Typ stellt dann, wie wir noch sehen werden, die ganzen Zahlen von –128 bis +127 dar (das sind interessanterweise genauso viele Zahlen).

Vorzeichenbehaftete Zahlen (Typ *Shortint*):
Besitzt das MSB den Wert 0, so ist die Zahl positiv. Besitzt es hingegen den Wert 1, so ist die Zahl negativ. Beispiel: 0101 1110 ist positiv und 1000 0101 ist eine negative Zahl.

Leider ist es nicht möglich, bei einer negativen Zahl einzig und allein das MSB auf 1 zu setzen und die restlichen Bits unverändert zu belassen. Dies würde bei der Addition zu Fehlern führen. Beispiel: Würde man die Dezimalzahl –3 als 1000 0011 darstellen, so ergäbe sich bei der Berechnung von 4 + (–3) :

0000 0100
+**1**000 0011
1000 0111

Das Ergebnis müsste demnach als (–7) interpretiert werden.

Die Lösung dieses Problems ist die Einführung des sog. *Zweierkomplements* für negative Dualzahlen.

> Zum besseren Verständnis betrachten wir vorher im Dezimalsystem das Zehnerkomplement. Das Zehnerkomplement ist die Differenz von einer Ziffer bis zur Zahl 10.

8 + (–3) = 8 + (–[10 – 7]) = 8 + 7 – 10 = **1**5 – 10 = 5

Statt die Zahl (–3) zu addieren, kann man also auch das Zehnerkomplement der Zahl 3 (also die Zahl 7) addieren; anschließend muss man den dabei entstehenden Übertrag ignorieren.
Das Zehnerkomplement von 3 kann man auch durch Bildung des Neunerkomplementes der Zahl 3 und anschließender Addition von 1 erhalten.

Bemerkung: Die Zahl 7 ist das Zehnerkomplement der Zahl 3. Die Zahl 3 ist wiederum das Zehnerkomplement der Zahl 7.

Dasselbe Verfahren werden wir nun im Dualsystem anwenden: Anstatt die Zahl %11 zu subtrahieren werden wir das entsprechende Zweierkomplement addieren und den dabei entstehenden Übertrag ignorieren.

Übrigens: die Zahl dez 3 zu subtrahieren, bedeutet mathematisch, die Zahl dez (–3) zu addieren. Die Zahl dez (–3) wäre demnach im Dualsystem das Zweierkomplement der Zahl dez 3.

Nun zurück zur Lösung unseres Problems der Darstellung von negativen Zahlen. Die Zahl (–3) wird als Zweierkomplement der Zahl 3 dargestellt. Das Zweierkomplement erhält man durch Bildung des Einerkomplementes (was besonders einfach ist: Einfach die Nullen und Einsen invertieren!) und anschließender Addition von 1.

Beispiel:	dez 3:	0000 0011
	Einerkomplement:	1111 1100
	Addition von 1:	0000 0001
	Zweierkomplement:	1111 1101

Also die Zahl (–3) wird durch 1111 1101 dargestellt.

Kontrollieren wir nun noch einmal die Summe 4 + (–3) :

```
  0000 0100
+ 1111 1101
-----------
1 0000 0001
```

Auch hier muss der Übertrag, das **neunte** Bit, ignoriert werden. Dann erhält man das richtige Ergebnis.

Ein schnelleres Verfahren zur Berechnung des Zweierkomplementes geht folgendermaßen (ohne Beweis):
Kopiere von rechts nach links alle Stellen bis einschließlich der erstmalig auftretenden Ziffer 1. Alle weiteren Stellen werden invertiert.
Beispiel: Zahl = 1110 1100
Zweierkomplement = 0001 0100

Eine negative 8-Bit-Dualzahl erhält man, indem man von der vorzeichenlosen 8-Bit-Darstellung des Betrages dieser Zahl das Zweierkomplement bildet.

Beispiel: dez 68: 0100 0100
dez (–68) = Zweierkomplement: 1011 1100

Problem: Wie erkennt man den Dezimalwert einer negativen Dualzahl?
Lösung: **Das Zweierkomplement einer negativen Dualzahl ergibt den Betrag dieser Zahl.**

Beispiel: Die Dualzahl 1011 1100 ist offensichtlich negativ, was am MSB zu erkennen ist.
Zweierkomplement: 0100 0100 = dez 68

Es gilt also: % 1011 1100 = dez –68

weitere Beispielsaufgaben:
a) dez 4 – dez 5 = dez 4 + dez (–5)

dez 5: 0000 0101
dez (–5): 1111 1011

Kontrolle: 0000 0100
+1111 1011
1111 1111

Dies ist offensichtlich eine negative Zahl. Ihren Absolutbetrag erhält man wieder durch Zweierkomplementbildung. Offensichtlich hat das Ergebnis hier den Wert dez (–1).

b) dez 4 – dez 4 = dez 4 + dez (–4)

dez 4: 0000 0100
dez (–4): 1111 1100

Kontrolle: 0000 0100
+1111 1100
1 0000 0000

Auch hier muss der Übertrag, das **neunte** Bit, ignoriert werden. Dann erhält man das Ergebnis 0.

c) dez(–4) + dez(–3) : 1111 1100 (nachprüfen!)
+1111 1101
1 1111 1001

Das **neunte** Übertragsbit wird ignoriert !
Das Ergebnis ist hier also % 1111 1001 = dez –7 (nachprüfen!)

d) dez 2 + dez (–3) : 0000 0010
+1111 1101
1111 1111 = dez (–1) (nachprüfen!)

Bemerkung: die negative Dualzahl 1000 0000 hat den Absolutwert 1000 0000.
Also gilt % 1000 0000 = dez (–128)

wichtig: Die größte vorzeichenbehaftete 8-Bit-Zahl ist offensichtlich
% 0111 1111 = dez 127 = 2^7–1
Die kleinste vorzeichenbehaftete 8-Bit-Zahl ist
% 1000 0000 = dez –128 = –2^7

Der Zahlenbereich für eine vorzeichenbehaftete 8-Bit-Zahl reicht also von –128 bis +127

Eine *ShortInt*-Zahl, also eine vorzeichenbehaftete 8-Bit-Zahl liegt zwischen −128 und +127. Bei der Addition zweier positiver *ShortInt*-Zahlen kann also kein **neuntes** Übertragsbit entstehen. Was passiert aber, wenn man z.B. die beiden *ShortInt*-Zahlen dez 64 und dez 66 addiert? :

```
  0100 0000
+ 0100 0010
  1000 0010
```

Dieses Ergebnis ist offensichtlich negativ und hat den Wert −126 (nachprüfen!).

Man spricht hier von einem sog. *Bereichsüberlauf*. Der Zahlenbereich für *ShortInt* wurde überschritten. Das richtige Ergebnis hätte (64 + 66 =) 130 sein müssen. Dies ist um **3** größer als die maximale *ShortInt*-Zahl.
Mathematisch interessant ist auch, dass −126 die **dritte** mögliche (angefangen bei −128) *ShortInt*-Zahl ist.

Derartige Bereichsüberläufe führen wesentlich öfter als man annimmt zu fehlerhaften Computerberechnungen. Wenn dabei völlig unsinnige Ergebnisse herauskommen (Beispiel: 64 + 66 = −126), kann man noch froh sein, weil man dann relativ schnell den Programmierfehler findet. Meistens ist es jedoch leider so, dass man etwa mit zwei unbekannten *ShortInt*-Variablen x und y rechnet, die jeden Wert zwischen −128 und +127 annehmen können, und das Endergebnis einer längeren Rechnung hängt dann etwa von der Summe von x und y ab. Dieses Endergebnis ist dann oft falsch, aber leider nicht offensichtlich unsinnig!

In neueren Programmiersprachen kann man die Compilierung eines Programmtextes so einstellen, dass der Computer mögliche Bereichsüberschreitungen überprüft und meldet (dann wird aber das eigentliche Programm umständlicher und langsamer).

Rechnet man z.B. mit *ShortInt*-Zahlen, so gelten einige mathematische Regeln nicht:

$(64 + 66) - 60 \neq 64 + (66 - 60)$ und $(5 * 30)\ \mathrm{DIV}\ 6 \neq 5 * (30\ \mathrm{DIV}\ 6)$

Derartige Fehlermöglichkeiten treten nicht nur bei *ShortInt*-Zahlen, sondern grundsätzlich bei jedem Zahlentyp auf.