

# Automatisierungstechnik AP1

-- Elementare Programmier Techniken --

## Verarbeiten breiter Operanden

Transportieren  
Logische Verknüpfungen und Tests  
Addieren  
Subtrahieren  
Erhöhen/Vermindern  
Vergleichen  
Verschieben

## Rechnen mit natürlichen und ganzen Binärzahlen

Addition  
Subtraktion  
Multiplikation  
Division  
Multiplizieren und Dividieren durch Verschieben  
Runden  
Sättigungsarithmetik  
Binär codierte Dezimalzahlen

## Einzelbitoperationen

Wodurch wird ein einzelnes Bit bestimmt?  
Bitadresse und Bitmaske  
Feste und berechnete Bitpositionen  
Bits setzen  
Bits löschen  
Bits transportieren  
Bits abfragen

## Bitfeldoperationen

Wodurch wird ein Bitfeld bestimmt?  
Bitfeldtransporte

## Bitvektoroperationen

Position der niedrigstwertigen Eins  
Position der höchstwertigen Eins  
Anzahl der Einsen

## Rechnen mit Booleschen Funktionen

Darstellung Boolescher Funktionen

## Speicheradressierung und Adreßrechnung

Zeichenketten, Vektoren, Matrizen, Records

## Verzweigungen

## Schleifen

## **Unterprogramme**

### **Der Stackmechanismus**

**Operationen mit drei Operanden**Schema:  $\langle r_c \rangle := \langle r_a \rangle \text{ op } \langle r_b \rangle$ 

1.  $\langle r_c \rangle := \langle r_a \rangle$
2. **op** rc, rb

Beispiel:

```
MOV rc, ra
ADD rc, rb
```

**Testoperationen (Verknüpfungen ohne Überschreiben des ersten Operanden)**

1. Prinzip: ein drittes Register als Hilfsregister verwenden. Dann weiter gemäß Operationen mit drei Operanden.
2. Prinzip: Zielregister (PUSH) retten und wiederherstellen (POP)

Beispiel:

```
PUSH rd
SUB rd, rs
POP rd
```

**Verarbeiten breiter Operanden**

Gelegentlich sind Operand zu verarbeiten, deren Bitanzahl ein Mehrfaches der Verarbeitungsbreite beträgt.

*Transportieren*

Die Operanden werden abschnittsweise gemäß der Verarbeitungsbreite transportiert.

*Transportieren zwischen direkt adressierten Registern (MOVE)*

Wenn es keine Möglichkeiten zur Adreßrechnung gibt, bleibt nur eine Folge von Transportbefehlen, wovon jeder einen Abschnitt (z. B. ein Byte) bewegt.

```
MOV rd_lo, rs_lo
MOV rd_hi, rs_hi
```

*Transportieren zwischen Registern und Speicher (LOAD, STORE)*

Bei direkter Adressierung (keine Adreßrechnung zur Laufzeit) bleibt nur eine Folge von Transportbefehlen, wovon jeder einen Abschnitt (z. B. ein Byte) bewegt. Dabei ist vom 2. Befehl an eine jeweils um 1 erhöhte Speicheradresse einzutragen (Adreßrechnung kann dem Assembler überlassen werden).

```
LDS rd_lo, mem_adrs
LDS rd_hi, mem_adrs+1
```

Bei indirekter Adressierung muß das Adreßregister jeweils um 1 erhöht werden (Postincrement).

```
LD rd_lo, Z+
LD rd_hi, Z+
```

*Transportieren zwischen Speicheradressen*

Das muß auf Transporte Speicher => Register, Register => Speicher zurückgeführt werden (LOAD-STORE-Folgen).

Direkte Adressierung:

```
LDS r, s_adrs
STS d_adrs, r
LDS r, s_adrs+1
STS d_adrs+1, r
```

Indirekte Adressierung (Beispiel: Quelladresse in Y, Zieladresse in Z):

```
LD r, Y+
ST Z+, r
LD r, Y+
ST Z+, r
```

*Laden von Direktwerten:*

```
LDI rd_lo, low(immediate)
LDI rd_hi, high(immediate)
```

*Holen von Direktwerten aus dem Programmspeicher:*

```
LPM rd_lo, Z+
LPM rd_hi, Z+
```

Wenn diese LPM-Version nicht unterstützt wird:

```
LPM
MOV rd_lo, r0
ADIW Z, 1
LPM
MOV rd_hi, r0
ADIW Z, 1
```

*Nullerweiterung*

Zu füllende Stellen mit Festwert Null belegen bzw. löschen.

```
LDS rd_0, mem_adrs
LDS rd_1, mem_adrs+1
EOR rd_2, rd_2
MOV rd_3, rd_2
```

*Vorzeichenerweiterung*

Zu füllende Stellen je nach Vorzeichen mit Festwert Null oder Eins belegen bzw. löschen oder setzen.

```
LDS rd_0, mem_adrs
LDS rd_1, mem_adrs+1
EOR rd_2, rd_2
SBRC rd_1, 7
COM rd_2
MOV rd_3, rd_2
```

*Logische Verknüpfungen und Tests*

Der typische Test betrifft lediglich die Nullbedingung (Ergebnis = 0 oder  $\lt 0$  (Zero-Flag)).

a) Verknüpfungsergebnis, kein Test

Abschnittsweise Verknüpfung:

```
AND rd_0, rs_0
AND rd_1, rs_1
AND rd_2, rs_2
AND rd_3, rs_3
```

b) nur Test, Verknüpfungsergebnis bedeutungslos

Man könnte nach jedem Abschnitt testen und bei Ergebnis  $\lt 0$  ans Ende verzweigen.

```
AND rd_0, rs_0
BRNE ready
AND rd_1, rs_1
BRNE ready
AND rd_2, rs_2
BRNE ready
AND rd_3, rs_3
ready:
```

c) Verknüpfungsergebnis und Test

Herausspringen ist nicht möglich.

1. Vorschlag: Testen mit zusätzlichem Hilfsregister *rt*. *rt* wird mit dem ersten Teilergebnis geladen. Alle weiteren Teilergebnisse werden disjunktiv verknüpft. Jede Eins im Ergebnis bleibt somit gleichsam in *rt* hängen ( $rt \lt 0$ , wenn Ergebnis  $\lt 0$ ).

```
AND rd_0, rs_0
MOV rt, rd_0
AND rd_1, rs_1
OR rt, rd_1
AND rd_2, rs_2
OR rt, rd_2
AND rd_3, rs_3
OR rt, rd_3
```

2. Vorschlag: Ergebnis wird wie üblich gebildet und dann insgesamt getestet. Hierzu wird die besondere Unterstützung der Z-Flag im SBCI-Befehl ausgenutzt. Ist das (Teil-) Ergebnis ungleich B Null, wird die Z-Flag gelöscht, ist es gleich Null, wird die Belegung nicht geändert. Eine von einem vorausgegangenen Befehl mit Ergebnis  $\lt 0$  gelöschte Z-Flag wird also nicht durch ein nachfolgendes Nullergebnis wieder gesetzt, sondern bleibt gleichsam hängen (Sticky Flag).

```
AND rd_0, rs_0
AND rd_1, rs_1
AND rd_2, rs_2
AND rd_3, rs_3
SUBI rd_0, 0
SBCI rd_1, 0
SBCI rd_2, 0
SBCI rd_3, 0
```

*Addieren*

Die Addition beginnt mit einem gewöhnlichen Additionsbefehl und wird dann mit Additionsbefehlen fortgesetzt, die den Eingangsübertrag berücksichtigen. Achtung: ADC hat keine Sonderunterstützung für die Nullbedingung. Das Ergebnis müßte also ggf. gesondert auf Null getestet werden (wie vorstehend beschrieben).

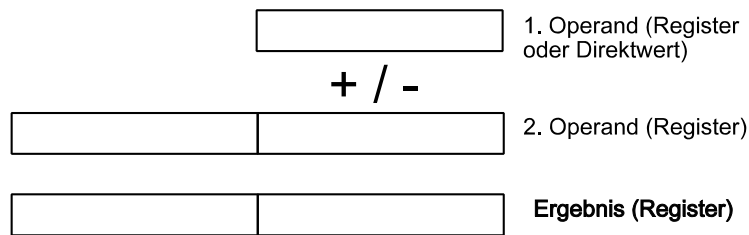
```
ADD rd_0, rs_0
ADC rd_1, rs_1
ADC rd_2, rs_2
ADC rd_3, rs_3
```

*Subtrahieren*

Die Subtraktion beginnt mit einem gewöhnlichen Subtraktionsbefehl und wird dann mit Subtraktionsbefehlen fortgesetzt, die den Eingangsübertrag berücksichtigen. Die Z-Flag gibt am Ende der Rechnung die Nullbedingung des gesamten Ergebnisses wieder (Sticky Flag).

```
SUB rd_0, rs_0
SBC rd_1, rs_1
SBC rd_2, rs_2
SBC rd_3, rs_3
```

*Addieren und Subtrahieren von Operanden unterschiedlicher Länge*

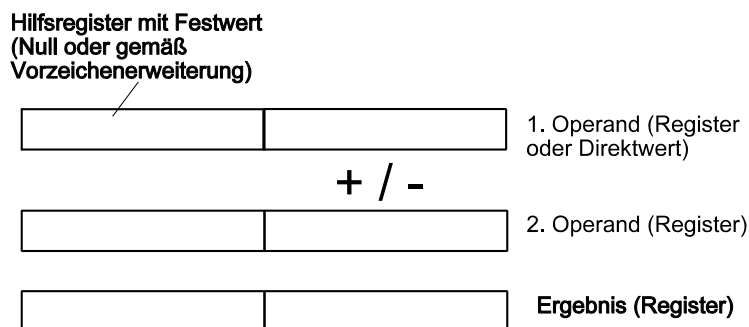


1. Lösung: Hilfsregister

Der erste Operand wird durch ein Hilfsregister verlängert, das mit einem passenden Festwert (0 oder FFH (Vorzeichenerweiterung)) geladen wird. (Es gibt Prozessoren, die spezielle Registeradressen haben, die passende Festwerte liefern.)

```
ADD rd_lo, rs
ADC rd_hi, r_aux
```

```
SUB rd_lo, rs
SBC rd_hi, r_aux
```



## 2. Lösung

Übertrag abfragen und höhere Stellen um 1 erhöhen oder vermindern

```
ADD rd_low, rs
BRCC ready
INC rd_hi
ready:
```

Wenn das Carry-Flag gestellt werden muß:

```
ADD rd_low, rs
BRCC ready
SUBI rd_hi, -1
ready:
```

```
SUB rd_low, rs
BRCC ready
DEC rd_hi
ready:
```

## 3. Lösung

Festwertaddition oder -subtraktion mit Eingangsübertrag.

```
ADD rd_low, rs
ADIC rd_hi, 0
```

Der AVR hat keine Festwertaddition (ADIC fehlt). Subtraktion von 0 führt aber zu Fehler, weil das Carry-Flag beim Addieren anders gebildet wird (müßte zwecks Subtraktion invertiert werden). Deshalb Lösung nur bei Subtraktion sinnvoll.

```
SUB rd_low, rs
SBCI rd_hi, 0
```

### *Addieren und Subtrahieren von Direktwerten*

Dies wird durch die Befehle SUBI und SBCI unterstützt. Das Addieren ist auf das Subtrahieren des Zweierkomplements zurückzuführen. Zweierkomplementbildung erledigt der Assembler. Längere Direktwerte (z. B. 32 Bits) müßten ggf. anderweitig umgerechnet und byteweise hexadezimal angegeben werden. Die Z-Flag gibt am Ende der Rechnung die Nullbedingung des gesamten Ergebnisses wieder (Sticky Flag).

Addieren:

```
SUBI rd_lo, low(-immediate)
SBCI rd_hi, high(-immediate)
```

Subtrahieren:

```
SUBI rd_lo, low(immediate)
SBCI rd_hi, high(immediate)
```

Alternative:

Hilfsregister verwenden, die mit den Direktwerten geladen werden.

*Erhöhen, Vermindern (Increment, Decrement)*

Beim Erhöhen wird eine Eins addiert, beim Vermindern eine Eins subtrahiert. Hierzu können die Befehle SUBI und SBCI angegeben werden (wie vorstehend beschrieben). +1 = 00..01H, -1 = FF...FFH. Die Z-Flag gibt am Ende der Rechnung die Nullbedingung des gesamten Ergebnisses wieder (Sticky Flag).

Erhöhen (Increment):

```
SUBI rd_0, 0xFF
SBCI rd_1, 0xFF
SBCI rd_2, 0xFF
SBCI rd_3, 0xFF
```

Vermindern (Decrement):

```
SUBI rd_0, 1
SBCI rd_1, 0
SBCI rd_2, 0
SBCI rd_3, 0
```

Erhöhen ohne Beeinflussung der C-Flag (mit INC-Befehlen):

```
INC rd_0
BRNE ready
INC rd_1
BRNE ready
INC rd_2
BRNE ready
INC rd_3
ready:
```

Vermindern ohne Beeinflussung der C-Flag (mit DEC-Befehlen):

```
TST rd_0
BREQ dec_1
DEC rd_0
RJMP ready
dec_1:
DEC rd_0
TST rd_1
BREQ dec_2
DEC rd_1
RJMP ready
dec_2:
DEC rd_1
TST rd_2
BREQ dec_3
DEC rd_2
RJMP ready
dec_3:
DEC rd_2
DEC rd_3
ready:
```

### *Zweierkomplementbildung*

Das Zweierkomplement kann durch Negieren und Erhöhen um 1 gebildet werden:

```
COM rd_0
COM rd_1
COM rd_2
COM rd_3
SUBI rd_0, 0xFF
SBCI rd_1, 0xFF
SBCI rd_2, 0xFF
SBCI rd_3, 0xFF
```

### *Vergleichen*

Das Vergleichen ist typischerweise ein Subtrahieren ohne Ergebnisspeicherung. Das Vergleichen von zwei Variablen beginnt mit einem gewöhnlichen Vergleichsbefehl und wird dann mit Vergleichsbefehlen fortgesetzt, die den Eingangsübertrag berücksichtigen. Die Z-Flag gibt am Ende der Rechnung die Nullbedingung des gesamten Ergebnisses wieder (Sticky Flag).

```
CP rd_0, rs_0
CPC rd_1, rs_1
CPC rd_2, rs_2
CPC rd_3, rs_3
```

### *Vergleichen mit Direktwert*

Da es keinen Direktwert-Vergleichsbefehl gibt, der den Eingangsübertrag unterstützt, muß das Vergleichen auf ein Subtrahieren (mit den Befehlen SUBI und SBCI) zurückgeführt werden. Ggf. Registerinhalte in den Stack retten und am Ende wiederherstellen. Das niedrigstwertige Byte könnte mit CPI verglichen werden (erspart Retten eines Registers).

```
PUSH rd_1
CPI rd_0, low (immediate)
SBCI rd_1, high (immediate)
POP rd_1
```

### *Verschieben*

Längere Operanden können byteweise verschoben und rotiert werden, wobei das Carry-Flag den jeweils weiterzugebenden Wert aufnimmt.

Linksverschieben:

```
LSL rd_0
ROL rd_1
ROL rd_2
ROL rd_3
```

Linksrotieren:

```
ROL rd_0
ROL rd_1
ROL rd_2
ROL rd_3
```

Rechtsverschieben:

```
LSR rd_3
ROR rd_2
ROR rd_1
ROR rd_0
```

Rechtsrotieren:

```
ROR rd_3
ROR rd_2
ROR rd_1
ROR rd_0
```

Rechtsverschieben arithmetisch:

```
ASR rd_3
ROR rd_2
ROR rd_1
ROR rd_0
```

*Betragsbildung (Absolutwert)*

Es ist das Vorzeichen zu prüfen. Ist die Zahl positiv, so ist nichts weiter zu tun. Ansonsten ist das Zweierkomplement zu bilden.

1 Byte:

```
SBRC rd,7
NEG rd
```

4 Bytes:

```
SBRC rd_3,7
RCALL Zweierkomplement
```

*Löschen*

Verfahren: Laden mit Direktwert. Antivalenzverknüpfung mit sich selbst, Subtraktion von sich selbst (setzt auch C-Flag zurück).

```
LDI rd, 0
```

```
EOR rd, rd
```

```
SUB rd, rd
```

```
SUB rd_0, rd_0
SBC rd_1, rd_1
SBC rd_2, rd_2
SBC rd_3, rd_3
```

Mehrere Register mit gleichem Wert füllen

Das erste mit dem Wert füllen und dann weiterkopieren (Wirtschaftlichkeit bei beliebigen Verarbeitungsbreiten).

```
LDI rd_0, beispielwert
MOV rd_1, rd_0
MOV rd_2, rd_0
MOV rd_3, rd_0
```

### *Sättigungsarithmetik*

Nach Addition, vorzeichenlos

Wenn C-Flag gesetzt, dann Ergebnis = FF...FFH (größte positive Zahl).

```
BRCC ready
LDI rd_0, 0xff
MOV rd_1, rd_0
MOV rd_2, rd_0
MOV rd_3, rd_0
ready:
```

Nach Subtraktion, vorzeichenlos

Wenn C-Flag gesetzt, dann Ergebnis = 0 (kleinste positive Zahl).

```
BRCC ready
EOR rd_0, rd_0
MOV rd_1, rd_0
MOV rd_2, rd_0
MOV rd_3, rd_0
ready:
```

Nach Addition oder Subtraktion, ganzzahlig

Wenn V-Flag gesetzt (Overflow), dann Ergebnisauf einen der Endwerte setzen. Maßgeblich ist dann der Zweierkomplement-Ausgangsübertrag (C-Flag bei Addition, invertierte C-Flag bei Subtraktion).

Hinweis: Overflow = Ausgangsübertrag XOR Übertrag in höchstwertige Bitstelle (gemäß Zweierkomplementarithmetik).

Wenn Zweierkomplementübertrag = 1, dann Bereichsüberschreitung, dann Ergebnis = 7H...FFH (größte positive Zahl).

Wenn Zweierkomplementübertrag = 0, dann Bereichsunterschreitung, dann Ergebnis = 80...00H (kleinste negative Zahl).

Addition:

```
BRVC ready
BRCC underrun
LDI rd_0, 0xff
MOV rd_1, rd_0
MOV rd_2, rd_0
LDI rd_3, 0x7f
rjmp ready
underrun:
EOR rd_0, rd_1
MOV rd_1, rd_0
MOV rd_2, rd_0
```

```
LDI rd_3, 0x80
ready:
```

Subtraktion:

```
BRVC ready
BRCS underrun
LDI rd_0, 0xff
MOV rd_1, rd_0
MOV rd_2, rd_0
LDI rd_3, 0x7f
rjmp ready
underrun:
EOR rd_0, rd_1
MOV rd_1, rd_0
MOV rd_2, rd_0
LDI rd_3, 0x80
ready:
```

*Multiplizieren*

Es wird zunächst vorzeichenlos multipliziert.

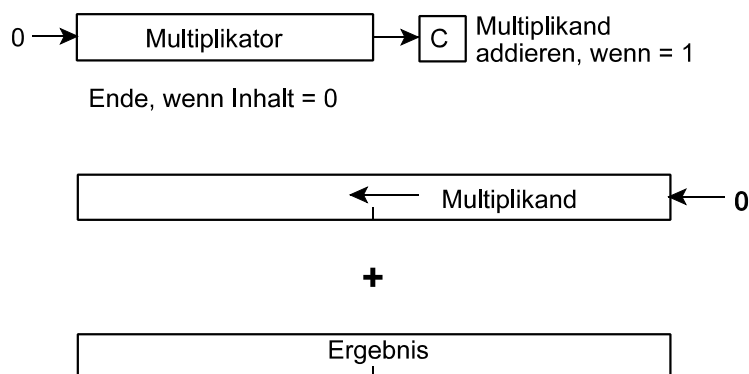
Ein naiver Ablauf:

Multiplikatorregister: einfache Länge. Multiplikator wird bitweise nach rechts herausgeschoben (ins C-Flag). Multiplikation ist zu Ende, wenn Multiplikatorregister nur Nullen enthält.

Multiplikandenregister: doppelte Länge. Anfänglich wird die höherwertige Hälfte gelöscht. Multiplikand wird in jedem Schritt um 1 Bit nach links verschoben.

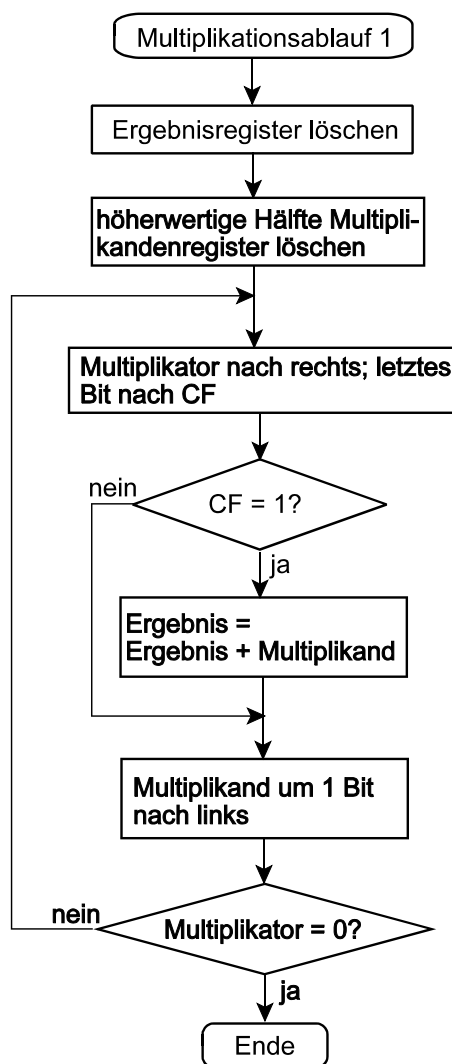
Ergebnisregister: doppelte Länge. Wird anfänglich gelöscht. Ist das Multiplikatorbit im C-Flag gleich 1, wird der aktuelle Inhalt des Multiplikandenregisters zum Ergebnisregister addiert (Akkumulatorwirkung).

- Multiplikand wird verschoben, Ergebnis bleibt stehen.
- Ablauf kommt zu Ende, wenn Multiplikator abgearbeitet ist.
- Es lohnt sich, ggf. den kleineren der beiden Operanden als Multiplikator zu verwenden (vorgeordnete Umsortierung).
- Addition in doppelter Verarbeitungsbreite (auch wenn in den niederen Stellen nur Nullen zu addieren sind).



```

clr mulcand_hi
clr res_lo
clr res_hi
w2:
lsl multor
brcc w1
add res_lo, mulcand_lo
adc res_hi, mulcand_hi
w1:
lsl mulcand_lo
rol mulcand_hi
tst multor
brne w2
    
```



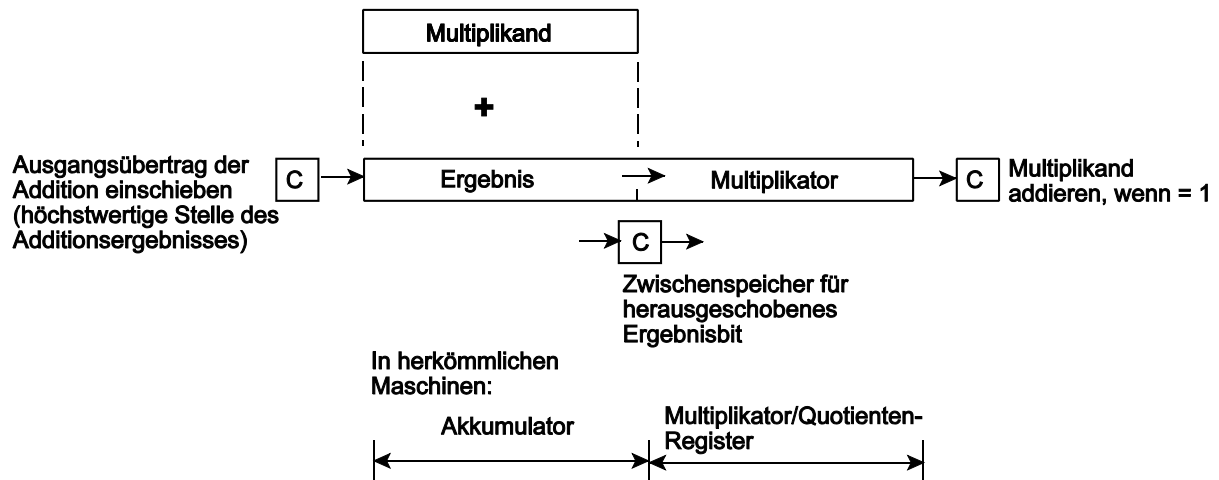
Ein Ablauf, der Register einspart:

Multiplikandenregister: einfache Länge.

Multiplikator- und Ergebnisregister: doppelte Länge. Das Ergebnis wird in der höherwertigen Hälfte aufgebaut, der Multiplikator wird in der niederwertigen Hälfte bitweise abgebaut, d. h. nach rechts

herausgeschoben (ins C-Flag). Dabei werden fertige Ergebnisstellen aus der höherwertigen Hälfte von links eingeschoben. Ist das Multiplikatorbit im C-Flag gleich 1, wird der aktuelle Inhalt des Multiplikanderegisters zur höherwertigen Hälfte des Multiplikator- und Ergebnisregister addiert (Akkumulatorwirkung).

- Multiplikation ist erst dann zu Ende, wenn alle Multiplikatorstellen herausgeschoben sind.
- Ergebnis wird verschoben, Multiplikand bleibt stehen.
- Es sind alle Bitstellen gemäß Operandenlänge abzuarbeiten (keine Ablaufverkürzung).
- Addition in einfacher Verarbeitungsbreite.



```

clr res
clc
ldi count, 8
loop:
ror res
ror multor
brcc w1
add res, mulcand
w1:
dec count
brne loop
ror res
ror multor

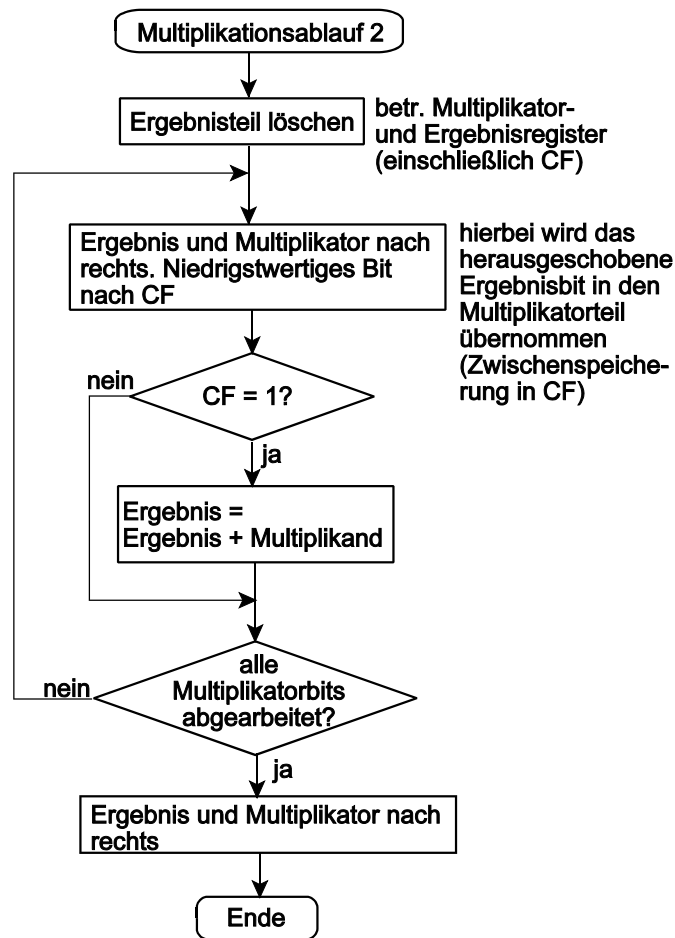
```

Variante:

```

clr res
clc
ldi count, 9
loop:
ror res
ror multor
dec count
breq ready
brcc loop
add res, mulcand
rjmp loop
ready:

```

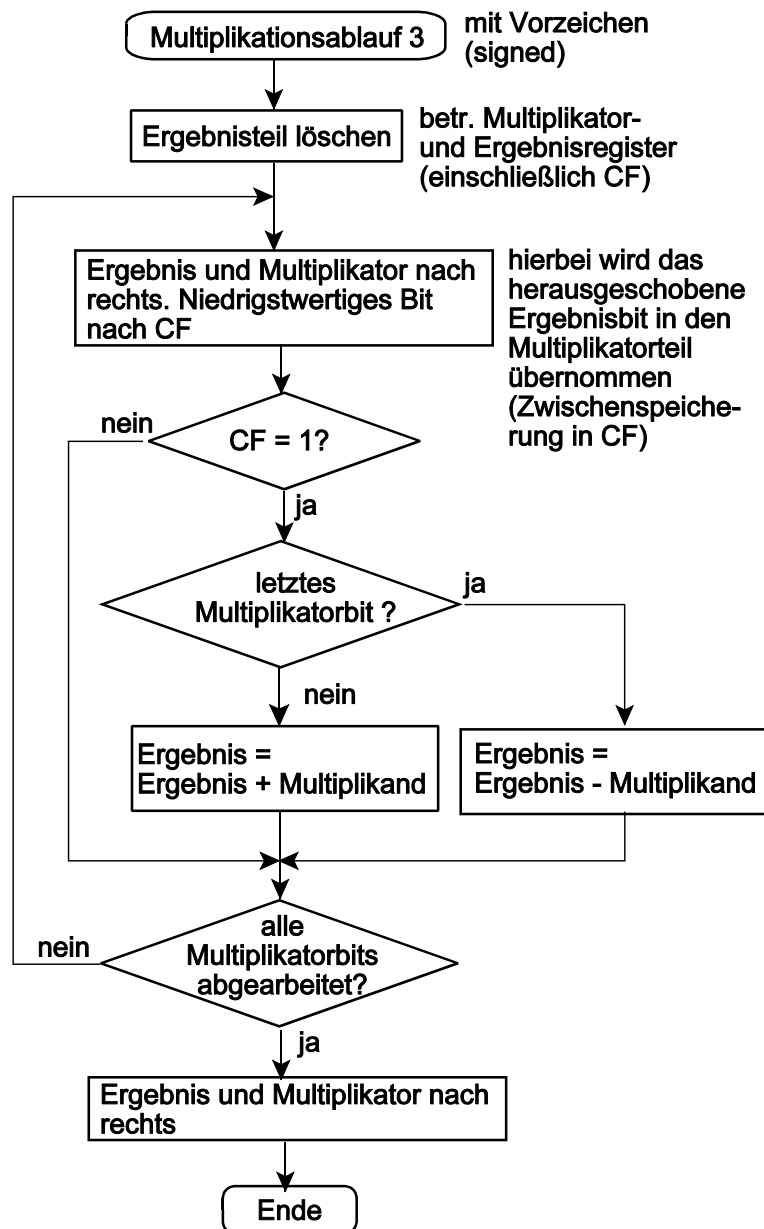


		7	6	5	4	3	2	1	0									
		Ergebnis = Ergebnis + Multiplikand																
Anfangszustand:		Inhalt = Null								7	6	5	4	3	2	1	0	
1.									0									0
2.									1	0								1
3.									2	1	0							2
4.									3	2	1	0						3
5.									4	3	2	1	0					4
6.									5	4	3	2	1	0				5
7.									6	5	4	3	2	1	0			6
8.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		7
Abschluss:		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

Nach dem Abarbeiten des letzten (höchstwertigen) Multiplikatorbits ist eine weitere Rechtsverschiebung erforderlich, um das endgültige Ergebnis komplett (einschließlich der im CF als Ausgangsübertrag abgelegten höchstwertigen Stelle) in das Register zu bringen.

Ganze Zahlen multiplizieren (mit Vorzeichen)

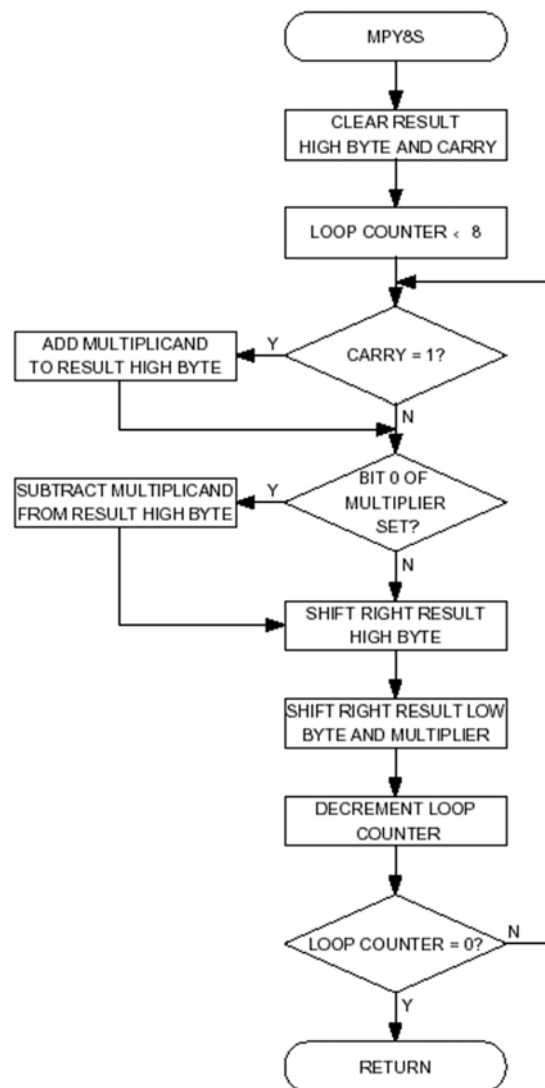
Die naheliegende Lösung: die Beträge multiplizieren (vorzeichenlos) und nachfolgende Vorzeichenrechnung (s. weiter unten). Bildung des Zweierkomplements, wenn sich ein negatives Vorzeichen ergibt. Alternativ dazu kann ein für ganze Zahlen ausgelegter Multiplikationsalgorithmus ausgeführt werden.



Der abgewandelte gewöhnliche Multiplikationsablauf. Im letzten Schritt wird nicht addiert, sondern subtrahiert. Hinweis: Beim Addieren/Subtrahieren eine Schutzstelle voranstellen, damit beim Überlauf das Vorzeichen erhalten bleibt.

Um das Vorzeichen zu erhalten, wird im folgenden Code das Zwischenresultat um 1 Byte verlängert. Hierzu werden die Register r\_z (fest auf Null) und r\_p (Vorzeichenstelle) verwendet.

```
clr res
clr r_z
clr r_p
clc
sbrcl multor,7
ldi r_p,255 ; Vorzeichenerweiterung in r_p
ldi count, 7
loop:
asr r_p
ror res
ror multor
brcc w1
add res, mulcand
adc r_p, r_z
w1:
dec count
brne loop
asr r_p
ror multor
ror res
brcc w2
sub res, mulcand
sbc r-p, r_z
w2:
asr r_p
ror multor
ror res
```



Ganzzahlige Multiplikation mittels Booth's Algorithmus (Atmel). Für jedes Multiplikatorbit ist zu entscheiden, ob addiert oder subtrahiert werden soll. Kann sein, daß Betragsbildung am Anfang und Vorzeichenrechnung am Ende schneller sind.

#### *Das Ergebnis beim vorzeichenlosen und ganzzahligen Multiplizieren*

Bei gleichen Operandenbitmustern ergibt sowohl die vorzeichenlose als auch die ganzzahlige Multiplikation ein gleiches Bitmuster in der ist die niederwertige Hälfte des Ergebnisses. Anwendung: beispielsweise bei der Adreßrechnung. Manche Prozessoren unterstützen nur die ganzzahlige Multiplikation, weil bei der Adreßrechnung die höherwertige Ergebnishälfte ohnehin nicht genutzt wird.

Rechengang:



Ergebnis:



Rechenbeispiel:

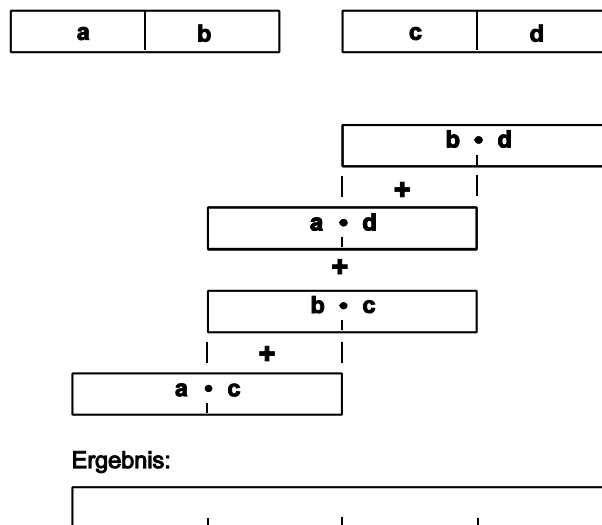
a) vorzeichenlos:  $194 \cdot 63 = 0xC2 \cdot 0x3F = 12\ 222 = 0x2F\ BE$

b) ganzzahlig:  $-62 \cdot 63 = 0xC2 \cdot 0x3F = -3906 = 0xF0\ BE$



*Längere Zahlen mit Multiplikationsbefehlen multiplizieren*

Die Multiplikationsbefehle (z. B. 8 • 8 Bits) werden im Sinne des kleinen Einmaleins verwendet. Die Operanden werden in Abschnitte (“Stellen“) gemäß der Operandenlänge des Multiplikationsbefehls unterteilt. Mit diesen Stellen wird schulmäßig gerechnet.



*Dividieren*

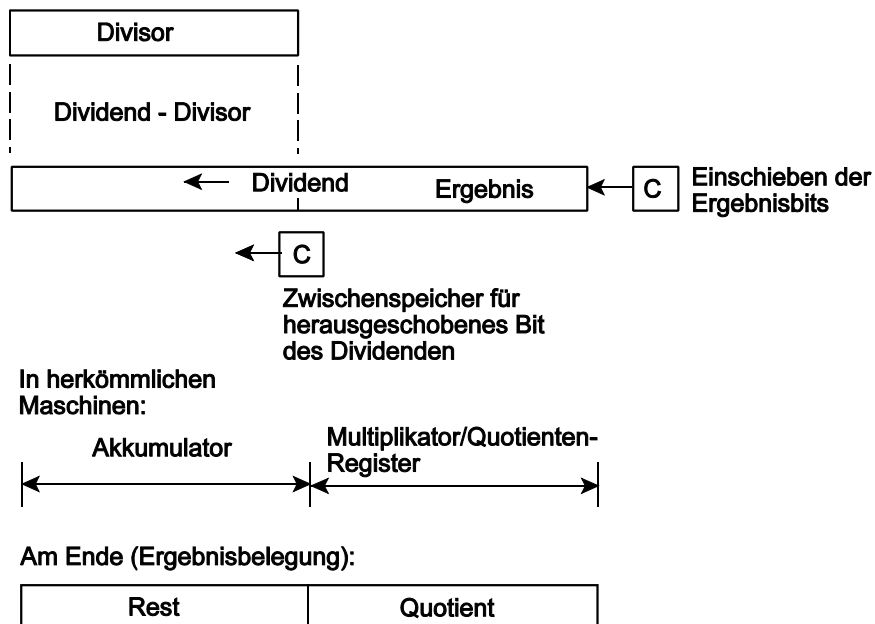
Es wird zunächst vorzeichenlos dividiert. Der Dividend wird von der höchstwertigen Stelle an mit dem Divisor verglichen (Rest = Abschnitt des Dividenden - Divisor). Die Ergebnisbildung beginnt mit dem höchstwertigen Bit. Ist der Rest Null oder positiv (Abschnitt des Dividenden  $\geq$  Divisor), so ist das Ergebnisbit = 1, und es wird mit dem ermittelten Rest weitergerechnet. Ist der Rest negativ, so ist das Ergebnisbit = 0, und der ursprüngliche Wert des Dividenden wird wiederhergestellt (Rest + Divisor). Das höchstwertige Bit wird links herausgeschoben; dafür wird das nächste Bit des Dividenden rechts nachgeschoben. Hierbei wird das Ergebnisbit in die niedrigste Bitposition des Registers eingeschoben.

```

EOR m2,m2
EOR m3,m3
MUL mb,md
MOV m0,r0
MOV m1,r1
MUL ma,md
ADD m1,r0
ADC m2,r1
BRCC w1
INC m3
w1:
MUL mb,mc
ADD m1,r0
ADC m2,r1
BRCC w2
INC m3
w2:
MUL ma,mc
ADD m2,r0
ADC m3,r1
    
```

*Die typische Sparlösung*

Der Dividend steht in einem Register doppelter Länge. Am Ende der Rechnung enthält dessen niederwertige Hälfte das Ergebnis und dessen höherwertige Hälfte den Rest.



Dies ist die typische Auslegung der Divisionsbefehle in vielen Prozessoren (z. B. Intel 86). Sie entspricht dem Grundgedanken, die Division als Umkehrung der Multiplikation anzusehen. Das Produkt zweier gleich langer Operanden hat doppelte Länge, und es liegt nahe, zu fordern, daß  $(A \cdot B) : B$  wieder  $A$  ergibt und  $(A \cdot B) : A$  wieder  $B$ . Dieser Auslegung entspricht - seitens der Hardware - das klassische Registermodell Akkumulator - Multiplikator/Quotienten-Register.

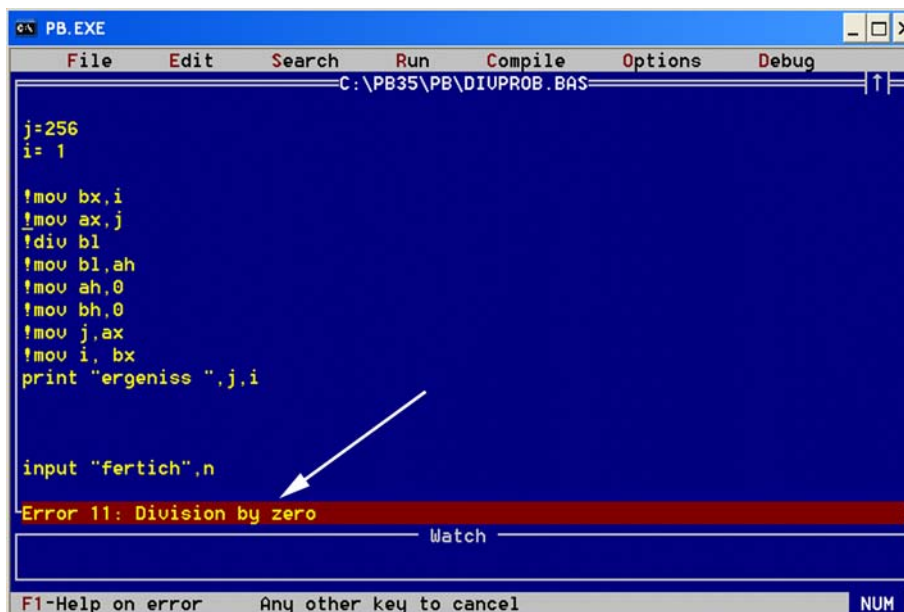
*Das Problem:*

Der Quotient darf einen bestimmten Wertebereich nicht überschreiten - er darf nicht größer werden als der Divisor. Typische Wertebereiche (vorzeichenlose Division):

- 16 Bits : 8 Bits : 0...255,
- 32 Bits : 16 Bits: 0...65 535,
- 64 Bits : 32 Bits: 0... $2^{32}-1$

Es ist aber ohne weiteres möglich, größere Quotienten zu erhalten. Der Grenzfall: Division : 1. Dann ergibt sich ein Quotient, der dem Divisor entspricht.

Manche Maschinen (z. B. Intel 86) reagieren mit einer Ausnahmebedingung, wenn sich beim herkömmlichen Dividieren ein zu großer Quotient ergibt. (Bei Intel: Interrupt 0 (Divisionsfehler). Diese Bedingung wird sinnigerweise auch beim Versuch ausgelöst, durch Null zu dividieren. Die Laufzeitumgebung des jeweiligen Systems gibt deshalb gelegentlich Warnungen ab, deren Ursache nicht gleich einleuchtet.)



```

PB.EXE
File Edit Search Run Compile Options Debug
C:\PB35\PB\DIUPROB.BAS

j=256
i= 1

!mov bx,i
!mov ax,j
!div bl
!mov bl,ah
!mov ah,0
!mov bh,0
!mov j,ax
!mov i,bx
print "ergebnis ",j,i

input "fertich",n
Error 11: Division by zero
Watch
F1-Help on error Any other key to cancel NUM

```

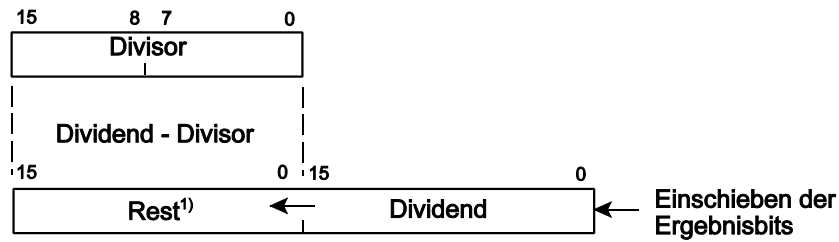
**Abb. 1** Warnung beim Versuch, 256 : 1 zu rechnen

*Ausweg:*

Die Division so implementieren, daß sowohl Quotient als auch Rest die volle Länge des Dividenden haben können, d. h. doppelte Länge der typischen Operanden. Also:

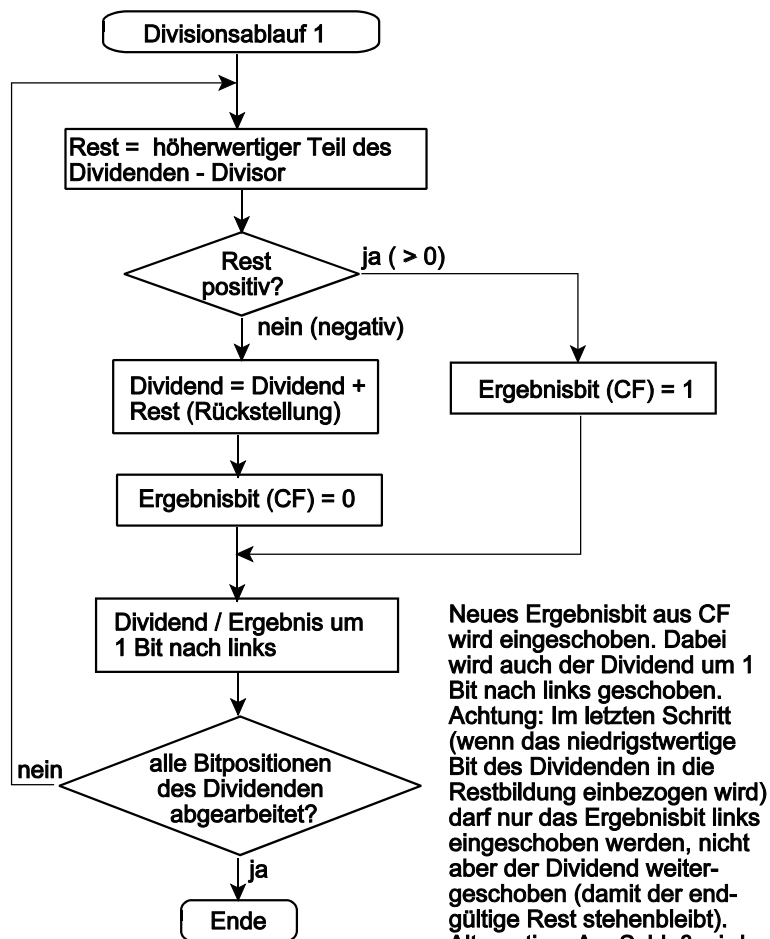
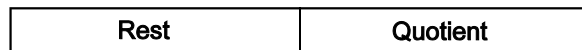
- bei 8-Bit-Arithmetik 16 Bits,
- bei 16-Bit-Arithmetik 32 Bits,
- bei 32-Bit-Arithmetik 64 Bits usw.

Es liegt dann nahe, auch den Divisor in dieser Länge zuzulassen. (Soll das Verhalten der typischen Divisionsbefehle emuliert werden, ist die höherwertige Hälfte des Divisors mit Nullen zu laden.)

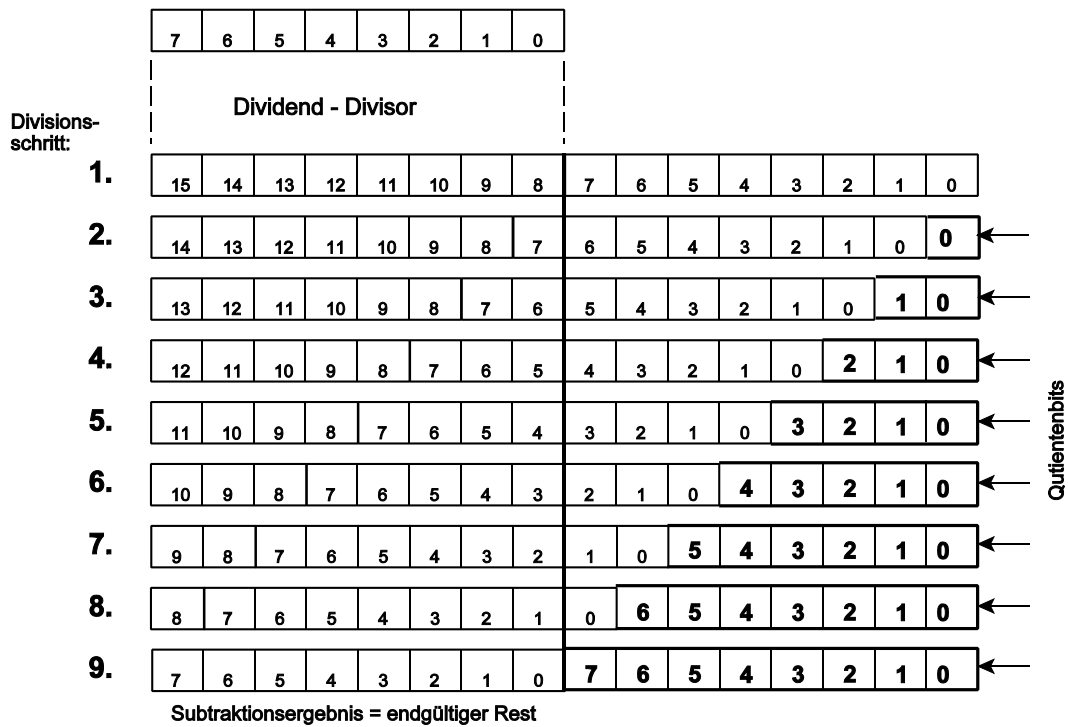


1): anfänglich gelöscht

Am Ende (Ergebnisbelegung):



Neues Ergebnisbit aus CF wird eingeschoben. Dabei wird auch der Dividend um 1 Bit nach links geschoben. Achtung: Im letzten Schritt (wenn das niedrigstwertige Bit des Dividenden in die Restbildung einbezogen wird) darf nur das Ergebnisbit links eingeschoben werden, nicht aber der Dividend weitergeschoben (damit der endgültige Rest stehenbleibt). Alternative. Am Schluß wird der Rest um 1 Bit nach rechts zurückgeschoben.

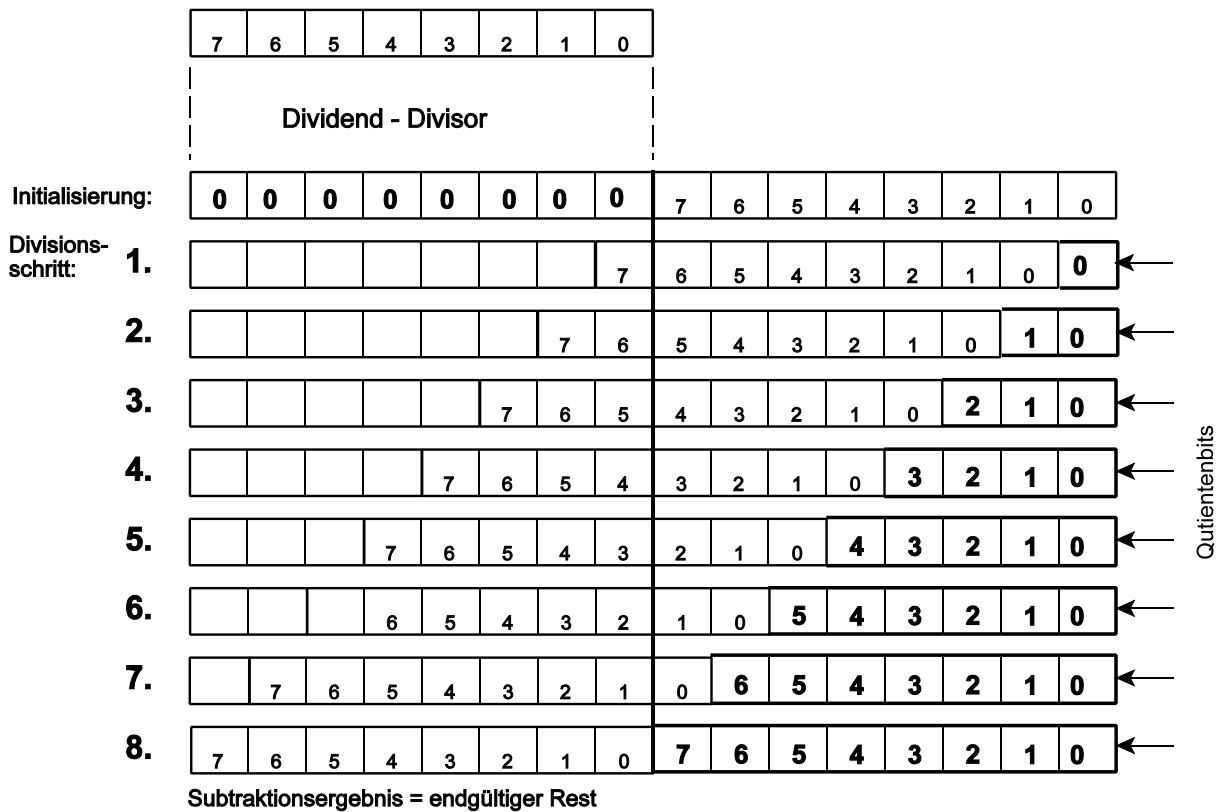


```

ldi count, 9
loop:
sub divid_hi,divisor
brcc w1
add divid_hi,divisor
clc
rjmp w2
w1:
sec
w2:
rol divid_lo
rol divid_hi
dec count
brne loop
ror divihi
    
```

Sonderbedingungen beim Dividieren (ggf. über Flagbits oder Fehlercodes zu signalisieren):

- Divison : 0 (kein Ergebnis),
- Quotient = 0 (ergibt sich, wenn Dividend < Divisor; dann ist Rest = Dividend),
- Rest = 0 (Division ist aufgegangen),
- höherwertige Hälfte des Quotienten = 0 (zur Unterstützung der Emulation herkömmlicher Divisionsbefehle),
- höherwertige Hälfte des Rests = 0.



Komplette 16-Bit-Division (alle Operanden und Ergebnisse 16 Bits):

```

di divirestlo,0
ldi diviresthi,0
ldi divicount,16
clc
divloop1:
rol divilo
rol divihi
rol divirestlo
rol diviresthi
sub divirestlo,divisor
sbc diviresthi,divisihi
brcc shres1 ; kein Carry, also positiv
add divirestlo,divisor
adc diviresthi,divisihi
clc
rjmp roll1
shres1:
sec
roll1:
dec divicount
brne divloop1
rol divilo
rol divihi
    
```

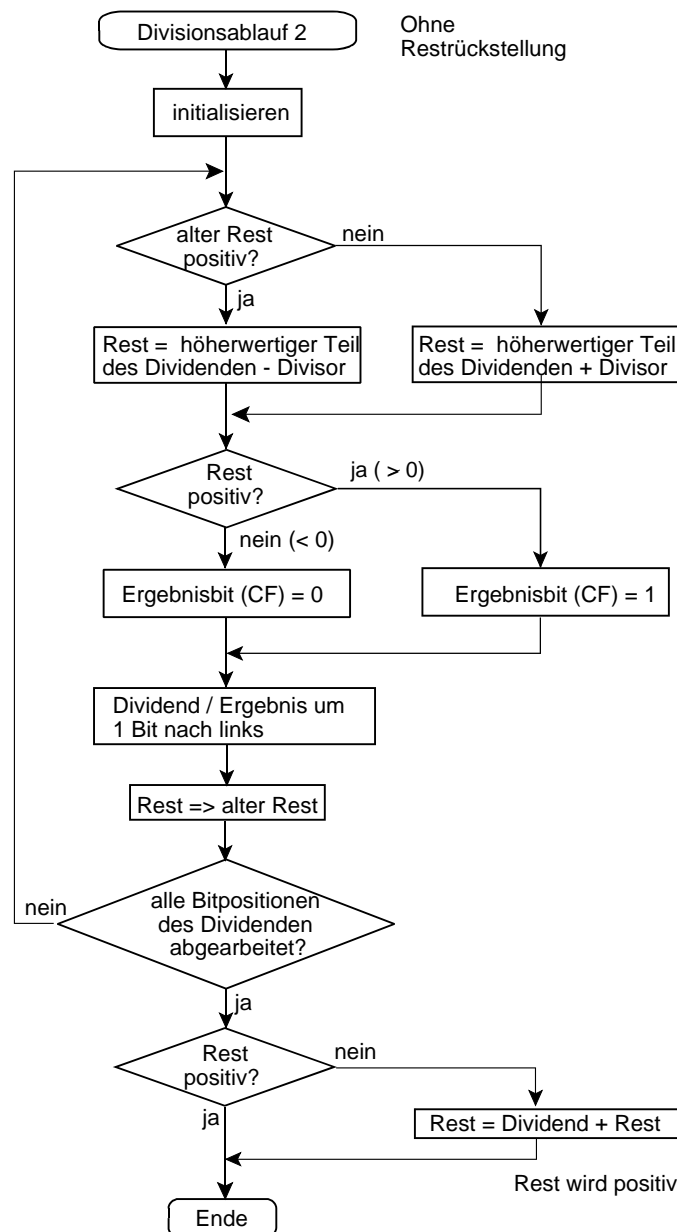
*Dividieren ohne Rückstellen des Restes (Nonrestoring Division)*

Beim Rückstellen wird der Divisor zum negativen Rest addiert, um den ursprünglichen Dividenden wieder zu erhalten. Dann wird der Dividend um ein Bit verschoben, und der Divisor wird wieder subtrahiert. Beide Rechenoperationen können zu einer zusammengefaßt werden. Das Rückstellen entfällt. Hat sich ein negativer Rest ergeben, so wird nach dem Verschieben der Divisor nicht subtrahiert, sondern addiert.

Die Linksverschiebung kann als Multiplikation mit 2 aufgefaßt werden. Der aus Rest r und Divisor d wiederhergestellte und verschobene Dividend ergibt sich zu  $2(r + d)$ .

Hiervon wird der Divisor subtrahiert:  $2(r + d) - d$ . Auflösen der Klammer ergibt  $2r + 2d - d = 2r + d$ .

Ist alles fertig gerechnet, kann es sein, daß der Rest negativ ist. Um einen positiven Rest zu erhalten, ist der Divisor zu addieren.



```

di diflag,0
ldi count,9
loop:
tst diflag
brne divadd
sub divid_hi, divisor
brcc rpos
rjmp rneg
divadd:
add divid_hi,divisor
brcs shresn ; CF divisor groesser
rneg:
ldi diflag,1
clc
rjmp shift
rpos:
sec
ldi diflag,0
shift:
rol divid_lo
rol divid_hi
dec count
brne loop
ror divid_hi
tst diflag
breq ready
add divid_hi, divisor
ready:

```

#### *Ganzzahliges Multiplizieren und Dividieren*

Mit den Absolutwerten rechnen (vorzeichenlos). Die Vorzeichenstelle des Ergebnisses muß Null sein (sonst Überlauf).

Vorzeichenrechnung:

- gleiche Vorzeichen: positives Ergebnis.
- ungleiche Vorzeichen: negatives Ergebnis.

```

SBRS r1_x, 7
RJMP pos_1
SBRC r2_x, 7
RJMP ready
RJMP negate
pos_1:
SBRS r2_x, 7
RJMP ready
negate:
Zweierkomplementbildung
ready:

```

### *Algorithmen in Soft- und Hardware*

Der Unterschied ist gelegentlich zu spüren. Viele Spitzfindigkeiten (in der einschlägigen Literatur) betreffen die Implementierung in Hardware. Die hier möglichen Beschleunigungseffekte können aber in der Software oftmals gar nicht wirksam werden.

- in der Hardware gibt es beträchtliche Unterschiede (Aufwand, Schaltzeiten) zwischen einer Addition und einem einfachen Transport (von einem Register zum anderen). In der Software dauern Addition und Transport zumeist gleich lang (jeweils ein elementarer Befehl, der oftmals in einem einzigen Taktzyklus erledigt wird).
- Entscheidungen laufen in der Hardware praktisch parallel ab (Gatternetzwerke). In der Software sind Entscheidungen mit bedingten Verzweigungen zu implementieren, die vergleichsweise viel Zeit kosten (nur eine einzige Ja/Nein-Entscheidung zu einer Zeit, Dauer der Verzweigungen (z. B. Verzweigungsbefehl zwei Takte, Operationsbefehl ein Takt)). Nicht selten dauert die Entscheidung, mit der man eine Operation vermeiden könnte, länger als das Ausführen dieser Operation.

### *Schieben und Arithmetik*

Das Linksschieben um  $n$  Bits entspricht einem Multiplizieren mit  $2^n$ , das Rechtsschieben einem Dividieren durch  $2^n$ .

#### *Achtung:*

Bei Zweierkomplementarithmetik funktioniert das nur, wenn mit vorzeichenlosen oder mit positiven Zahlen gerechnet wird (z. B. bei der Adreßrechnung).

Zum Dividieren vorzeichenloser Zahlen ist die logische Rechtsverschiebung zu verwenden (die die freiwerdenden Stellen mit Nullen auffüllt).

### *Beim Rechnen mit negativen Zahlen*

- kann beim Linksschieben das Vorzeichen verlorengehen (manche Maschinen haben Befehle, die das Erkennen dieses Sonderfalls unterstützen (Beispiel: IA-32)),
- kann es sein, daß beim Dividieren gerundet wird (SHIFT ergibt anderes Ergebnis als ganzzahlige Division).

#### *Dividieren durch Linksschieben:*

- a) Operand vorzeichenlos oder positiv: das Herausschieben des bzw. der niedrigstwertigen Bits wirkt als Rundung in Richtung Null - wie beim richtigen Dividieren. Es ergibt sich ggf. ein niedrigerer Wert. Z. B.  $7 : 2 = 3$  Rest 1.
- b) Operand negativ: das Herausschieben des bzw. der niedrigstwertigen Bits wirkt als Rundung in Richtung  $-4$ . Somit kann sich - anders als beim richtigen Dividieren - auch hier ein niedrigerer (= mehr negativer) Wert ergeben. Z. B.  $-7 : 2 = -3$  Rest 1 beim Dividieren, aber  $-4$  beim (arithmetischen) Rechtsschieben. Wann das nicht passieren kann: beim Rechnen im Einerkomplement.

### *Ersatz für Linksschieben*

Addieren des Operanden zu sich selbst

LSL rd = ADD rd, rd

*Schnelles Multiplizieren mit kurzen Festwerten*

Statt Multiplikation mit n-fache Addition:

Beispiel  $rs * 3$ :

```
EOR rd,rd
ADD rd,rs
ADD rd,rs
ADD rd,rs
```

Mehrmalige Addition bei variablem Multiplikator

Die Addition in einer Schleife ausführen. Multiplikator als Schleifenzähler.

```
EOR rd,rd
TST rmtor
c1:
BREQ ready
ADD rd,rmd
DEC rmtor
RJMP c1:
ready:
```

Statt Multiplikation mit Zweierpotenz Linksverschieben um n Bits ( $n = \text{Exponent}$ )

Beispiel  $rd * 4$ :

```
LSL rd
LSL rd
```

Kombinationen aus Verschieben und Addieren

Z. B.  $* 10 = \text{Operand um 3 nach links (8)} + \text{Operand um 1 nach links (* 2)}$ .

Statt Division durch Zweierpotenz Rechtsverschieben um n Bits ( $n = \text{Exponent}$ )

Beispiel:  $rd : 4$

```
LSR rd
LSR rd
```

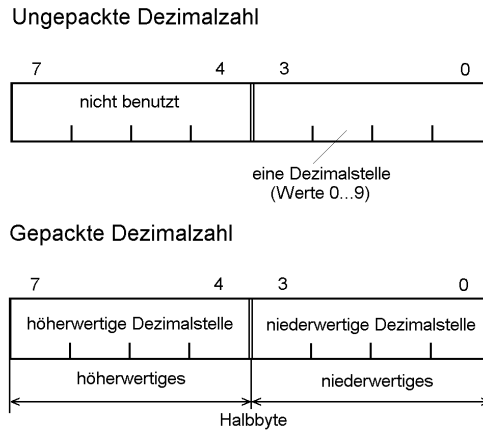
Einfachdivision durch wiederholte Subtraktion

Beispiel  $a : n$

```
LDI rquot,0
TST ra
BREQ ready
c1:
SUB ra,rn
BRMI c2
INC rquot
RJMP c1:
c2:
ADD ra,rn
ready:
```

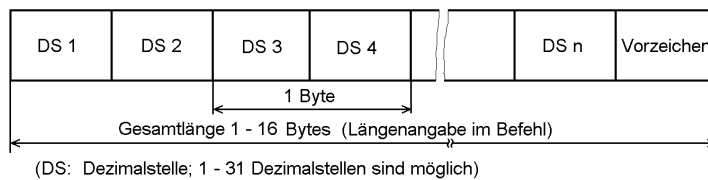
**Rechnen mit Dezimalzahlen**

Dezimalzahlen werden stellenweise codiert und als Zeichenketten dargestellt. Die übliche Codierung: BCD (0 = 0x0H, 9 = 0x9 usw.). Es gibt ungepackte und gepackte BCD-Zahlen.



Eine ungepackte Dezimalzahl ist ein einzelnes Byte mit einer einzigen Dezimalstelle im niederwertigen Halbbyte (Bits 0...3). Gepackte BCD- Zahlen enthalten in jedem Byte zwei Dezimalstellen, wobei die Stelle im höherwertigen Halbbyte die höherwertige ist. Diese Datenstrukturen sind an sich vorzeichenlos. Ein Vorzeichen muß gesondert codiert werden (Vorzeichen und Betrag sind voneinander unabhängig; Sign/Magnitude-Darstellung).

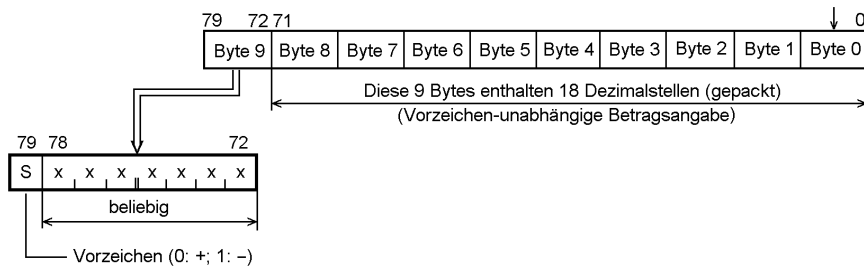
1. S / 370



2. x86 BCD-Format für Gleitkommaverarbeitung

(10 Bytes; wird automatisch in Gleitkommazahl gewandelt)

Das niederwertige Halbbyte enthält die niedrigstwertige Dezimalstelle.



**Abb. 2** Formate vorzeichenbehafteter BCD-Zahlen (Beispiele)

Moderne Mikroprozessoren haben keine ausgebaute BCD-Arithmetik. Die befehlsseitige Unterstützung - falls überhaupt vorgesehen (die AVR's haben lediglich ein Half-Carry-Flag für den Übertrag von Bit 3 nach Bit 4) - betrifft vielmehr nur Hilfsoperationen, die dazu dienen, trotz der an sich binären

Arbeitsweise des Prozessors korrekte dezimale Verarbeitungsergebnisse zu erzeugen (Dezimalausgleich). Die eigentliche BCD-Arithmetik wird softwareseitig implementiert, und zwar stellenweise (wie beim schulmäßigen Rechnen).

BCD-Addition und -Subtraktion (1): mittels binärer Addition/Subtraktion

Die einzelnen BCD-Halbbytes dürfen lediglich die Werte zwischen 0H und 9H einnehmen. Beim binären Rechnen entstehen aber auch Werte zwischen AH und FH. Beispiel:  $5 + 7 = 12 = CH$  (binäre Addition). Im BCD-Code müßte aber herauskommen:  $2H +$  ein Dezimal-Übertrag in die nächste Stelle.

Korrekturmaßnahmen (Dezimalausgleich):

- wenn beim Addieren Ergebnis  $> 9$ : zusätzlich eine 6 addieren und einen Übertrag zur nächsten Dezimalstelle weitergeben
- wenn beim Subtrahieren Ergebnis negativ ( $< 0$ ): zusätzlich eine 6 subtrahieren und einen Übertrag zur nächsten Dezimalstelle weitergeben.

Beim stellenweise Addieren/Subtrahieren ist jeweils der Übertrag aus der vorhergehenden Stelle mit zu verrechnen.

BCD-Addition und -Subtraktion (2): mittels Ergebnistabelle

Je Rechengang eine feste Ergebnistabelle für alle Operandenkombinationen (jeweils Werte 0...9 zuzüglich Eingangsübertrag) aufbauen und mit der jeweiligen Wertekombination zugreifen.

Adreß-Offset zum Eintritt in die Tabelle:

8	7	6	5	4	3	2	1	0
Cin	Wert 2. Operand				Wert 1. Operand			

Jede Tabelle muß insgesamt  $2^9 = 512$  Einträge (= Bytes) haben. Für Addition und Subtraktion sind somit insgesamt 1 kBytes erforderlich.

Der einzelne Eintrag:

7	6	5	4	3	2	1	0
-	-	Cout	VZ	Wert			

BCD-Multiplikation:

Durch schulmäßiges Rechnen oder mittels Multiplikationstabelle (kleines Einmaleins).

Tabellen-Offset:

7	6	5	4	3	2	1	0
Wert 2. Operand				Wert 1. Operand			

Die Tabelle muß insgesamt  $2^8 = 256$  Einträge (= Bytes) haben. Überträge gibt es nicht.

Der einzelne Eintrag:

7	6	5	4	3	2	1	0
höhere Dezimalstelle				niedere Dezimalstelle			

*Alternative zur BCD-Arithmetik:*

Rationale Zahlen (echte Brüche), die mit Binärzahlen dargestellt werden.

Beispiel:  $3,27 = 3 + \frac{27}{100}$  bzw.  $\frac{327}{100}$

Das ist aber oftmals nicht praktikabel (Kompatibilität zu vorhanden Datenbeständen).

BCD-Vorwärtszählung (ungepackte Dezimalzahlen mit Nullen in den höherwertigen Tetraden)

```

INC rd_0
CPI rd_0, 0x0A
BRNE ready
EOR rd_0, rd_0
INC rd_1
CPI rd_1, 0x0A
BRNE ready
EOR rd_1, rd_1
INC rd_2
CPI rd_2, 0x0A
BRNE ready
EOR rd_2, rd_2
INC rd_3
CPI rd_3, 0x0A
BRNE ready
EOR rd_3, rd_3
ready:

```

BCD-Rückwärtszählung (ungepackte Dezimalzahlen mit Nullen in den höherwertigen Tetraden)

```

TST rd_0
BREQ dec_1
DEC rd_0
RJMP ready
dec_1:
LDI rd_0, 0x09
TST rd_1
BRE dec_2
DEC rd_1
RJMP ready
dec_2:
LDI rd_1, 0x09
TST rd_2
BRE dec_3
DEC rd_2
RJMP ready
dec_3:
LDI rd_2, 0x09

```

```
TST rd_3
BREQ dec_wrap
DEC rd_3
R JMP ready
dec_wrap:
LDI rd_3, 0x09
ready:
```

### Vergleichen

BCD-Zahlen sind Binärzahlen, in denen stellenweise (also in 4-Bit-Abschnitten) bestimmte Bitkombinationen nicht vorkommen. Die Ordnungsrelation in den Abschnitten (Dezimalstellen) entspricht aber jener der Binärzahlen  $9 > 8 > 7 \dots > 1 > 0$ . Somit können Dezimalzahlen wie entsprechend lange vorzeichenlose Binärzahlen behandelt und durch binäre Subtraktion miteinander verglichen werden.

- Vorzeichen entfernen.
- Zahlen auf gleiche Länge bringen (ggf. Nullerweiterung).
- In ungepackten Zahlen müssen die höheren Tetraden der zu vergleichenden Operanden mit jeweils gleichen Werten belegt sein. Ggf. nachhelfen.
- Von der niederwertigsten Stelle an beginnen. Ggf. auf Anordnung im Speicher achten. Gewöhnliche Binärzahlen oftmals niedrigstwertige Stellen zuerst (little endian), Dezimalzahlen oftmals höchstwertige Stellen zuerst (big endian - auch in Little-Endian-Maschinen).

### Stellenweises Vergleichen:

Von der höchstwertigen Stelle an beginnen. Beim ersten Ungleich-Ergebnis kann der Ablauf beendet werden.

### Wandlung BCD - binär

#### 1. Verfahren:

Gegeben sei eine 5stellige BCD-Zahl mit den Dezimalstellen a b c d e.

Die Binärzahl ergibt sich zu:

$e + 10d + 100c + 1000d + 10000a$  (Rechnen im Binären).

Der Rechengang läßt sich umformen zu

$e + 10 (d + 10 (c + 10 (b + 10 a)))$

Start:	bin = a
1. Teilrechnung:	bin = 10 bin + b
2. Teilrechnung:	bin = 10 bin + c
3. Teilrechnung:	bin = 10 bin + b
Schlußrechnung:	bin = 10 bin + a

Multiplikation mit 10 = Linksverschiebung um 3 Bits (\* 8) + Linksverschiebung um 1 Bit (\* 2).

#### MUL10:

```
LSL bin_l
ROL bin_h
MOV aux_l, bin_l
MOV aux_h, bin_h
LSL aux_l
```

```

ROL aux_h
LSL aux_l
ROL aux_h
ADD bin_l, aux_l
ADC bin_h, aux_h
RET

```

; Muster

```

ldi r20, 3
ldi r21, 5
ldi r22, 8
ldi r23, 6
ldi r24, 1
ldi r25, 0

```

```

LDI bin_h, 0
MOV bin_l, r20 ; a
RCALL MUL10
ADD bin_l, r21; b
RCALL MUL10
ADD bin_l, r22; c
ADC bin_h, r25
RCALL MUL10
ADD bin_l, r23; d
ADC bin_h, r25
RCALL MUL10
ADD bin_l, r24; e
ADC bin_h, r25

```

2. Verfahren:

Dezimalzahl stellenweise bis auf Null herunterzählen, dabei Binärzahl um Stellenwert erhöhen.

Binärzahl = niedrigste Dezimalstelle

Zehnerstelle auf Null herunterzählen, dabei Binärzahl um 10 erhöhen.

Hunderterstelle auf Null herunterzählen, dabei Binärzahl um 100 erhöhen usw.

```

MOV bin, dec_l
TST dec_h
BREQ ready
c1:
SUBI bin, -10
DEC dec_h
BREQ ready
RJMP c1
ready:

```

*Wandlung binär - BCD*

1. Verfahren:

Zweierpotenzen als Dezimalwerte zueinander addieren (Dezimaladdition).

Zweierpotenzen können fest gespeichert oder durch Verdoppeln (Dezimaladdition) in jedem Schritt immer wieder neu berechnet werden.

## 2. Verfahren

Fortlaufende Division durch 10

bin : 10 => bin; Rest = Einerstelle

bin : 10 => bin; Rest = Zehnerstelle

bin : 10 => bin; Rest = Hunderterstelle

usw. (bis Quotient = 0).

Abwandlung: Statt Division durch 10 Multiplikation mit 0,1 (Gleitkommarechnung)

$$0,1D = 0,000110011001100...B = \frac{1}{2^4} + \frac{1}{2^8} + \frac{1}{2^{12}} + \frac{1}{2^{16}} + \frac{1}{2^{20}} + \frac{1}{2^{24}} + \frac{1}{2^{28}} + \frac{1}{2^{32}} + \frac{1}{2^{36}} + \frac{1}{2^{40}} + \frac{1}{2^{44}} + \frac{1}{2^{48}} + \frac{1}{2^{52}} + \frac{1}{2^{56}} + \frac{1}{2^{60}} + \frac{1}{2^{64}} + \frac{1}{2^{68}} + \frac{1}{2^{72}} + \frac{1}{2^{76}} + \frac{1}{2^{80}} + \frac{1}{2^{84}} + \frac{1}{2^{88}} + \frac{1}{2^{92}} + \frac{1}{2^{96}} + \frac{1}{2^{100}} + \dots$$

Rechengang nach Zuse (Z1, Z3):

$$0,1D = 2^{-4} \cdot 1.1001100110011001B$$

Das 0,1fache einer beliebigen Binärzahl X:

$$0,1 X = 2^{-4} \cdot X \cdot 1.1001100110011001B$$

Wir rechnen zunächst  $X \cdot 1,100110011001100... B$  und multiplizieren dann mit  $2^{-4}$  (= Division durch 16 = Rechtsverschiebung um 4 Bits).

$$\begin{aligned} X_1 &= X + .1 X = X + X/2 && (X + X \cdot 2^{-1}) \\ X_2 &= X_1 + .0001 X_1 = X_1 + X_1/16 = X \cdot 1.10011 && (X_1 + X_1 \cdot 2^{-4}) \\ X_3 &= X_2 + .000 000 01 X_2 = X_2 + X_2/256 = X \cdot 1.1001100110011 && (X_2 + X_2 \cdot 2^{-8}) \\ X_4 &= X_3 + .000 000 000 000 0001 X_3 = X_3/65536 = X \cdot 1.100110011001100110011001100110011 && (X_3 + X_3 \cdot 2^{-16}) \end{aligned}$$

Abschließend wird  $X^4$  mit  $2^{-4}$  multipliziert.

## 3. Verfahren

Fortlaufende Subtraktion von 10 statt Division durch 10. Zahlenwert < 100.

```
LDI dec_l, 0
LDI dec_h, 0
TST bin
BREQ ready
c1:
SUBI bin, 10
BRMI c2
INC dec_h
RJMP c1
c2:
SUBI bin, -10
MOV dec_l, bin
```

#### 4. Verfahren

Division durch absteigende Zehnerpotenzen. Der Quotient ergibt jeweils eine Dezimalstelle, der Rest wird durch die nächst-kleinere Zehnerpotenz dividiert.

Beispiel: 16-Bit-Binärzahl.

Binärzahl : 10 000 = 5. (höchstwertige) Dezimalstelle.

Rest : 1000 = 4. Dezimalstelle.

Rest : 100 = 3. Dezimalstelle.

Rest : 10 = 2. Dezimalstelle

Rest = 1. (niedrigstwertigste) Dezimalstelle.