IV.

Adressierungsarten

INHALT

1. Allgemeines
2. direkte Adressierung
3. indirekte Adressierung
4. Beispiele mit Informationsangaben
5. Einschub für Praktikum
   (Block – Codes)

1. Allgemeines

Adressbereich für Daten und Programm :

Die 65xx Prozessorfamilie hat einen Adressbereich von

$0000 - $FFFF

Das high-Byte wird als Seite bezeichnet

$\Rightarrow$ 256 Seiten

Besondere Bedeutung einiger Seiten :

- zero-page : schneller Zugriff (1 Zyklus weniger, Spezielle Addressierungsarten, Operand nur ein Byte)
- one-page : Systemstapel (Stack) angelegt : der Stapelzeiger (Register) gibt das „low“ Adressbyte der nächsten leeren Speicherzelle an, das „high“ Adressbyte wird automatisch 01 gesetzt.
- FF-page : Rücksetz- und Unterbrechungsvektoren werden hier angelegt, d.h. , Startadressen der Routinen, die bei entsprechenden Interrupts abzuarbeiten sind.

Page 0 , 1	im RAM (Schreib / Lese)
Page FF	im ROM (Festwertspeicher)

Anmerkung (Abspeichern von Adressen) :

Sollen die beiden Adressbytes auf Speicheradressen geschrieben werden, so gilt in der Regel :

- **low Adressbyte → low Speicherbyte**
- **high Adressbyte → high Speicherbyte**

Sinnvoll bei Shifts in Adressen (Add. / Sub.), die vor dem Ablegen der Adresse in den Speicher ausgeführt werden

Anwendung zum Beispiel bei :

- **Aufruf von Unterprogrammen (s. Kapitel I)**
  Rücksprungadresse auf Stapel legen
- **indirekte Adressierung (s. Kapitel V)**
- **Interruptvektoren (s.später)**

2. direkte Adressierung

Direkte Adressierung wird auch effektive Adressierung genannt; es ist höchstens eine einfache Adress-rechnung erforderlich.

(i) absolut :

Adresse wird im Operanden geführt (explizit oder symbolisch).

Befehl i.a. 3 Byte lang (Ausnahme zero-page)

Beispiele :

```
adc   Summand     ; symbolisch
sta  $4050        ; explizit
```

zero-page :

```
Summand = $AB     ; high -Byte nicht nötig
adc Summand
sta  $50          ; zero-page Adresse
```

(ii) unmittelbar (immediate) : (2 Byte lang)

Konstantenadressierung :

Operand ist das Datenbyte selbst.

Beispiele :

```
ldx  #$0C          ;  hexadezezimale  Wertangabe
lda  #12           ;  dezimale Wertangabe
adc #%00001100 ;  binäre Wertangabe
```

(iii) implizit (implied) : (1 Byte lang)

keine Adressangabe notwendig, da sich Befehl auf bestimmte Register bezieht (ein oder zwei).

Beispiele :

```
inx    ;  Inhalt des X-Register um eins inkrementieren
tax    ;  Inhalt des Akkumulator  ⟶  X-register
rol    ;  Rollen im Akkumulator über C-Bit (s.o.)
```

rol, asl etc. können sowohl implizit als auch absolut benutzt werden.

(iv) relative :

Assembler berechnet Displacement (offset Δ).

Befehl i.a. 2 Byte lang

$$\Delta = \text{Ziel} - \text{aktueller Befehlszählerstand}$$

$$-128 \leq \Delta \leq 127$$

Im Programmablauf wird Δ zum aktuellen Befehlszählerstand mit mod $100 addiert.

- **Entsteht für Δ>0 dabei ein Carry, liegt eine Seitenüberschreitung vor.**
- **Entsteht für Δ<0 dabei kein Carry, liegt eine Seitenunterschreitung vor.**

$\Rightarrow$ **in beiden Fällen 1 Rechenzyklus mehr**

Vorteil zur absoluten Adressierung :

Programmkode ist verschiebbar (relocatable programming code),
z.B. bei Verschiebungen durch ein (Multitasking / -programming) Betriebssystem
oder bei wiederverwendbaren Programmteilen.

Beispiel : (vgl. Branch-Befehle)

```
Loop :  ldx #10          42FE   A2  0A
                         4300
          .                .
          .                .
          .                .
                           .
        dex              430C   CA
        bne  Loop        430D   D0  EF  ←  Δ
        nop              430F
          .
          .
          .
        rts
```

Berechnung der Sprungadresse :

```
   $ 43 0F
 + $    EF        Rücksprung, da MSB = 1
     0 1          Carry
   $ 43 FE
```

Da Carry=0 folgt Seitenunterschreitung

⇒ Sprungadresse $ 42FE

(v) absolut indiziert (immer nachindiziert) :

Die effektive Adresse ergibt sich durch Addition (mod $1000) von Basisadresse (Operand) und dem Inhalt des Indexregisters X bzw. Y.

Der Befehl ist i.a. 3 Byte lang.

Anwendung besonders bei Tabellen, oft integriert in Schleifen.

Beispiel : lda Tab, x

Tab symbolisiert die Basisadresse (base address).

Ist x=0 wird die Basisadresse angesprochen.

Statt X- kann auch mit gleicher Funktionsweise das Y-Register verwendet werden.

Gesamtbeispiel :

```
.org $4000
        ldx #0
        ldy #0
loop:   lda Tab_1, x
        sec
        sbc Tab_2, x
        bmi cont
        sta Diff, y
        iny
cont:   inx
        cpx #5
        bne loop
        rts

Tab_1: .byte  $FF, $16 , $A5, $27, $B3
Tab_2: .byte  $45, $3A , $84, $E2, $9C
Diff:     .byte  $00
```

Schleife kann nur aufsteigend programmiert werden, da Tabellenwerte für x=0 auch gelesen werden muss.

Trick zur absteigenden Lösung
Wird später besprochen

3. indirekte Adressierung

Die effektive Adresse ist in einem Speicherplatz (2 Byte) abgelegt. Dieser Speicherplatz heißt :

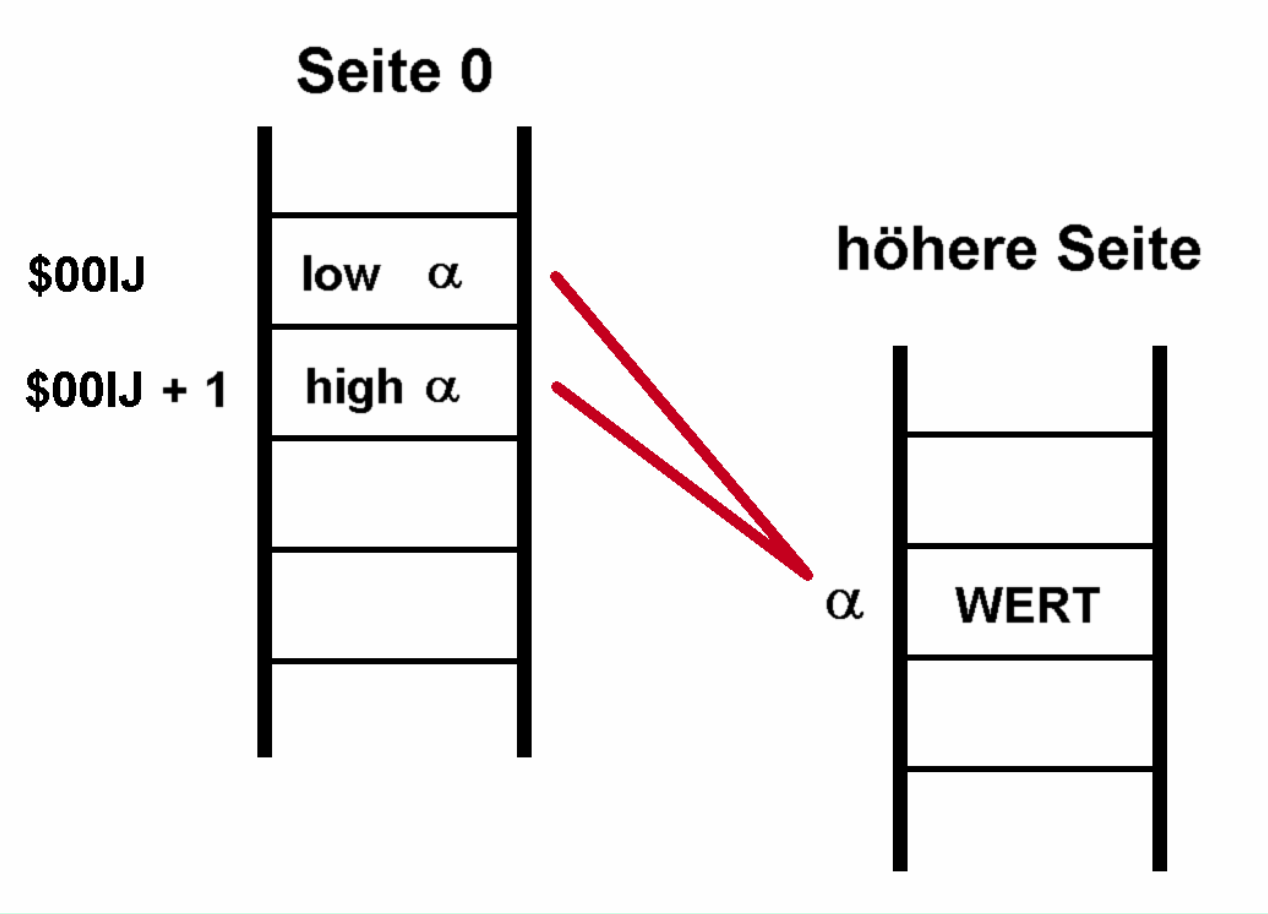
- **Zeiger (Pointer) vgl. C++ oder**
- **Vektor (Bezeichnung bei Interruptsbearbeitungen)**

Bei heutigen Prozessoren fungieren 16-bit Register, die schnellen Zugriff erlauben, als Speicher für Adressen.

Die 65xx Prozessoren gibt es keine 16-bit Register. Um dennoch einen möglichst schnellen Zugriff zu realisieren, wird die zero-page als Speicherbereich für die Zeiger verwendet.

Indirekte Adressierung für den Wert in der Speicherzelle α = $5010

Low α = $10
High α = $50



Vorteile der indirekten Adressierung :

hohe Flexibilität

- **anzusprechende Datenpakete können leicht gewechselt werden.**

  Keine mühsame Suche nach Lese-Aufrufen im gesamten Programm , sondern ausreichend ist eine neue Initialisierung des Zeigers an zentraler Stelle.

- **Zeiger ist programmierbar.**
- **Der Aufruf von Unterprogrammen kann von Parametern abhängig gemacht werden (siehe später) .**

Wie wird ein Zeiger deklariert :

```
Pointer = $80        ;  zero-page  Adresse wird für
                        Zeiger festgelegt.
          .
          .
          .
lda #<Table          ;  „low address-byte“ laden
Sta Pointer
lda #>Table          ;  „high address-byte“ laden
Sta Pointer+1
          .
          .
          .
Table: .byte $02, $FF, ….. , $6A
```

Vorsicht bei zero-Page Adressierung

```
    1:                          .org $4010
    2:
    3: 4010 : 2E 23 41          rol A
    4: 4013 : 2E 23 41          rol A
    5: 4016 : 20 1A 40          jsr B
    6:
    7:                          A=$4123
    8:
    9: 4019 : 60                rts
   10:
   11: 401A : A9  6             B:   lda #$06
   12: 401C : 60                     rts
   13:
   14:                          .end
Block      0 Level      0
B:401A
A:4123
```

```
    1:                          .org $4010
    2:
    3: 4010 : 26 1D             rol A
    4: 4012 : 26 1D             rol A
    5: 4014 : 20 1A 40          jsr B
    6:
    7:                          A=$1D    ;  falsch
    8:
    9: 4017 : 60                rts
   10:
   11: 4018 : A9  6             B:   lda #$06
   12: 401A : 60                     rts
   13:
   14:                          .end
Block      0 Level      0
B:401A
A:  1D
```

nachträgliche Korrektur des Programmspeichers aufgrund Zero-page Benutzung

```
    1:                          .org $4010
    2:
    3:                          A=$1D    ;  richtig
    4:
    5: 4010 : 26 1D             rol A
    6: 4012 : 26 1D             rol A
    7: 4014 : 20 18 40          jsr B
    8:
    9: 4017 : 60                rts
   10:
   11: 4018 : A9  6             B:   lda #$06
   12: 401A : 60                     rts
   13:
   14:                          .end
Block      0 Level      0
B:4018
A:  1D
```

Indirekte Adressierungsarten :

(i) absolut :

nur Sprungbefehl

jmp (Zeiger)

Anmerkung (Beispiele) :

externe Ereignisse führen zu indirekt adressierte Sprünge.

$\overline{\text{RES}}$; reset jmp (RESVec) = jmp ($FFFC)

$\overline{\text{NMI}}$; hardware Interrupt
jmp (NMIVec) = jmp ($FFFA)

$\overline{\text{IRQ}}$; hardware Interrupt
jmp (IRQVec) = jmp ($FFFE)

(ii) implizit :

Angabe des Operanden nicht nötig, da Befehl direkt eine Speicheradresse anspricht, in der die Adresse des anzusprechenden Speichers abgelegt ist.

Beispiele :

Stapelbefehle: **pha , pla , php , plp**

Hier wird der Stapelpointer angesprochen, in dem die anzusprechende Adresse des Stapels (Seite 1) eingetragen ist.

Software-break **brk**

Der Interruptvektor auf $FFFE muss definiert sein.

Einziger Unterschied zum Hardware Interrupt ist das gesetzte B-Bit.

iii. (vor)indizierte indirekte Adressierung : (indexed indirected)

Befehl ist 2 Byte lang

Formaler Befehl :

sta (P-list,x) ; **P-list ist als Seite-0-Adresse deklariert.**

Anwendung :

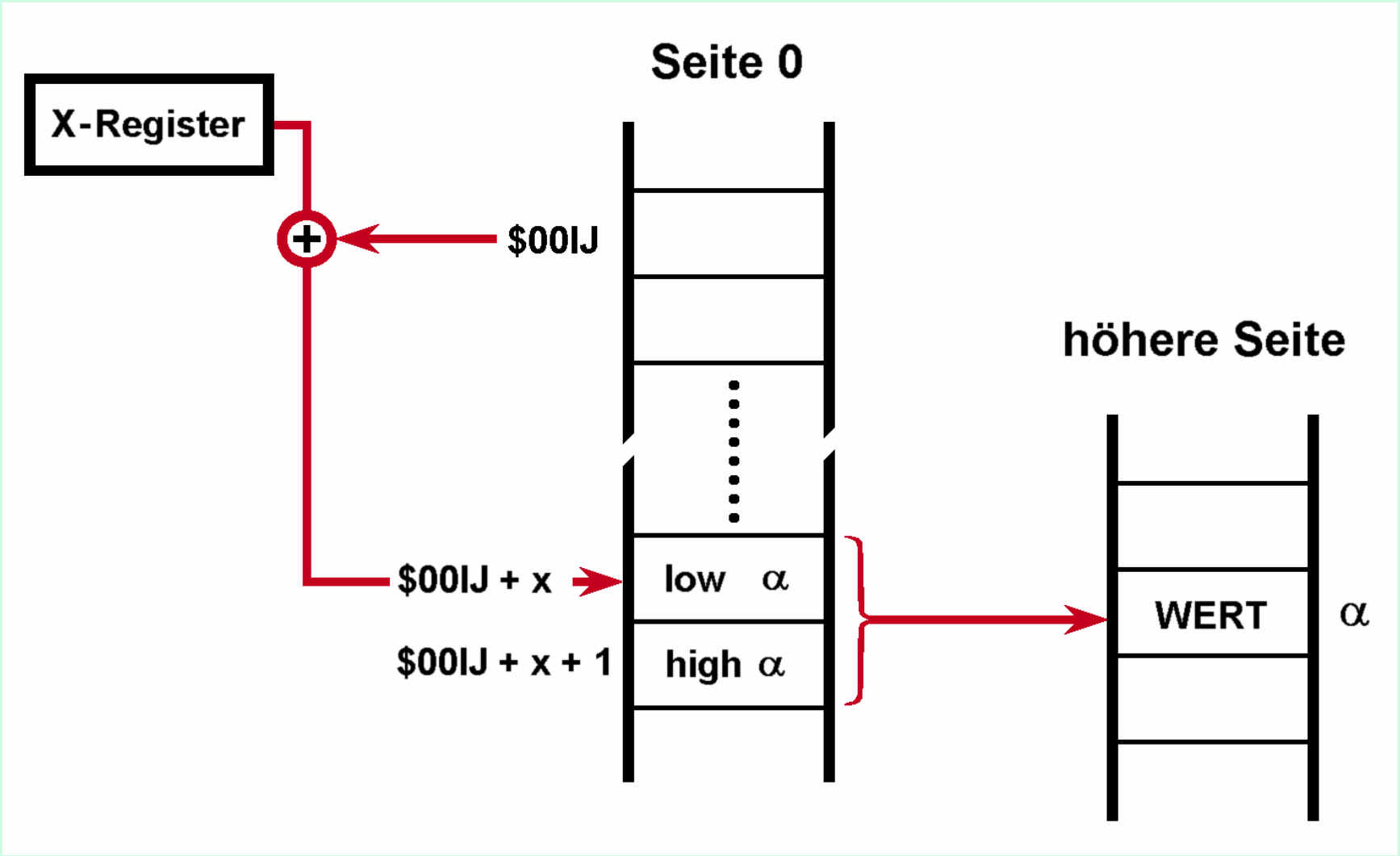
a) **Indirekte absolute Adressierung für andere Befehle als jmp**

adc (Pointer,0)

b) **Zeiger-Listen**

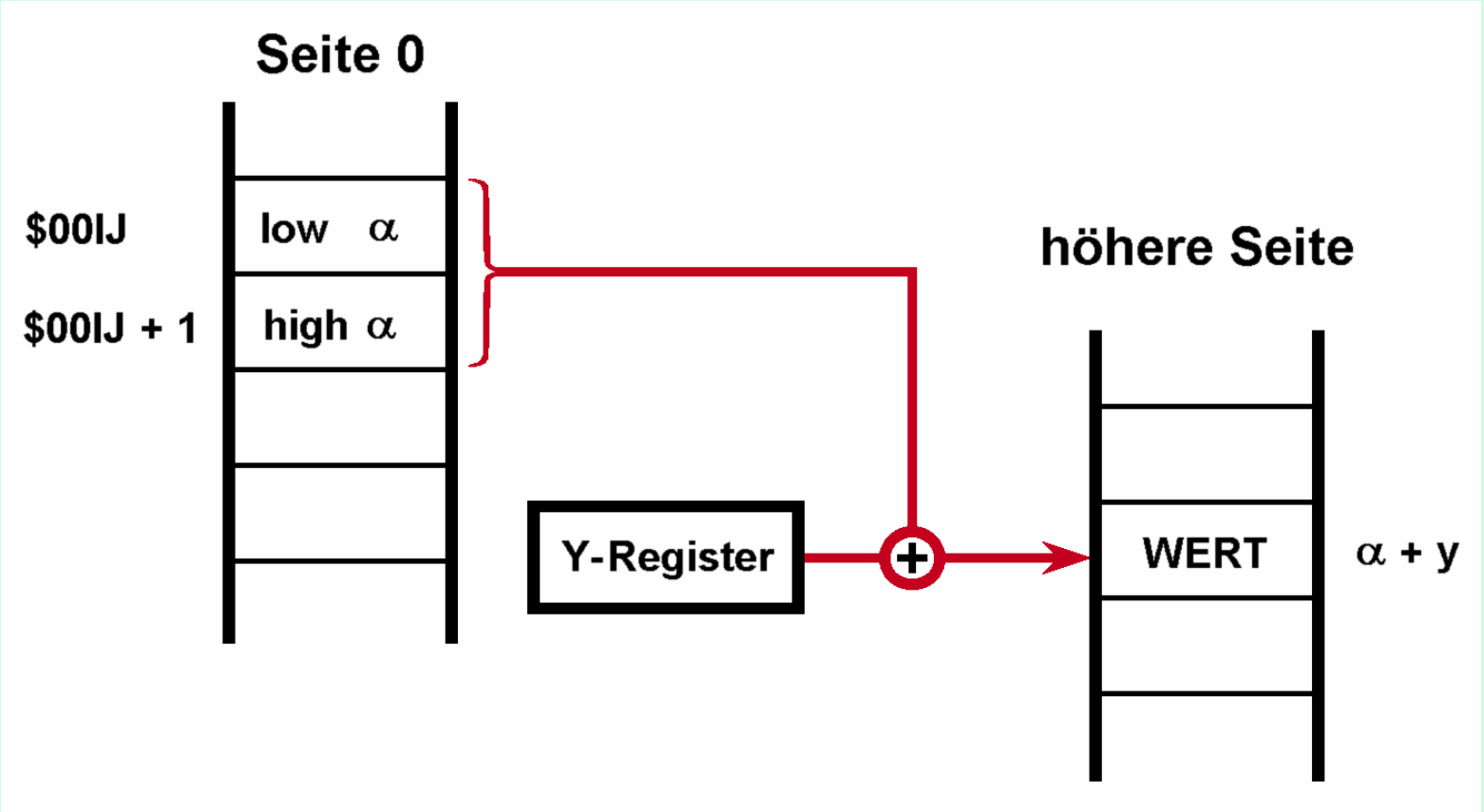
x muss zweimal inkrementiert werden, um zum nächsten Zeiger überzugehen.

Schemazeichnung zur vorindizierten indirekten Adressierung :



iv. Indirekt (nach)indizierte Adressierung : (indirected indexed)

Schemazeichnung zur nachindizierten indirekten Adressierung :



Befehl ist 2 Byte lang

Formaler Befehl :

```
sta (Pointer),y    ;  Pointer ist als Seite-0-Adresse
                      deklariert.
```

Diese Adressierungsart verbindet indirekte Eigenschaften mit indizierter Adressierung (wie bei direkter Adressierung).

⇒ **sehr mächtige Adressierungsart**

4. Beispiele mit Informationsangaben

Benutzte Abkürzungen :

- **IMP** implizit
- **Akku** implizit Akkumulator
- **IM** unmittelbar (immediate)
- **R** relativ
- **ABS** absolut
- **ABX** absolut , x-indiziert
- **ABY** absolut , y-indiziert
- **ZP** absolut , zero page
- **ZPX** absolut , zero page , x-indiziert
- **IND** indirekt (nur JMP-Befehl)
- **(IND,X)** indirekt , x-vorindiziert
- **(IND),Y** indirekt , y-nachindiziert

Beispielbefehle :

Mnemo-Code	Maschinen Code	Adressierung	Byteanzahl	Zyklen
ADC #Operand	$69	IM	2	2
ADC Operand	$65	ZP	2	3
ADC Operand , x	$75	ZPX	2	4
ADC Operand	$6D	ABS	3	4
ADC Operand , x	$7D	ABX	3	4*
ADC Operand , y	$79	ABY	3	4*
ADC (Operand , x)	$61	(IND , X)	2	6
ADC (Operand) , y	$71	(IND) , Y	2	5*

* falls page Grenze überschritten wird

rot: Korrektur zum studentischen Skript

Mnemo-Code	Maschinen Code	Adressierung	Byteanzahl	Zyklen
ROR	$6A	AKKU	1	2
ROR Operand	$66	ZP	2	5
ROR Operand , x	$76	ZPX	2	6
ROR Operand	$6E	ABS	3	6
ROR Operand , x	$7E	ABX	3	7

Mnemo-Code	Maschinen Code	Adressierung	Byteanzahl	Zyklen
BMI	$30	R	1	2*

* falls page Grenze überschritten wird
und falls Bedingung wahr 1 zusätzlicher Zyklus

BIT – Befehl :

Virtuelle konjunktive Verknüpfung des Operanden mit Akkumulator (virtuelles „AND“) , Akku-Inhalt bleibt erhalten

Mnemo-Code	Maschinen Code	Adressierung	Byteanzahl	Zyklen
BIT Operand	$24	ZP	2	3
BIT Operand	$2C	ABS	3	4

Beinflussung von Flags :

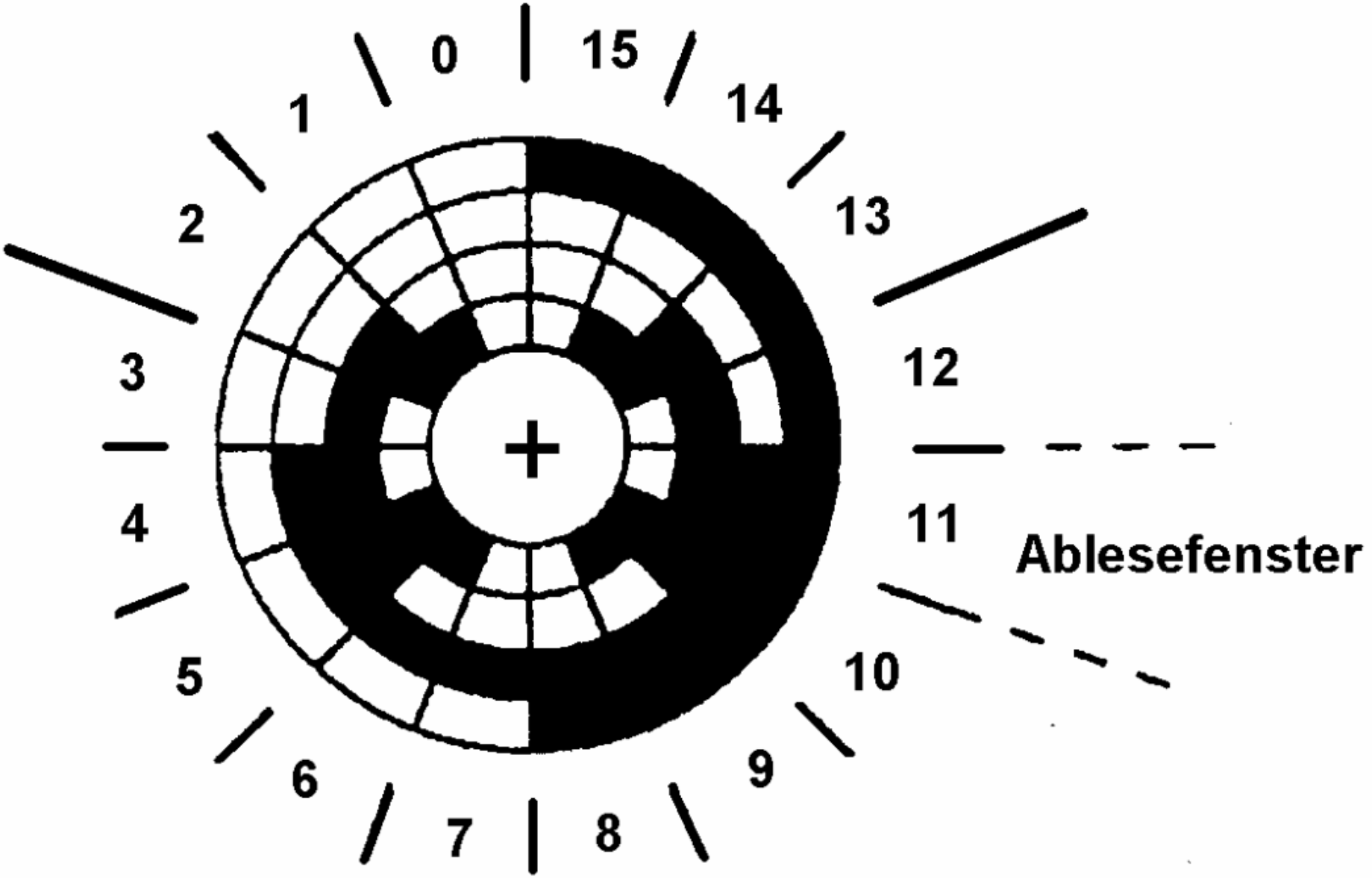
N	V		B	D	I	Z	C
x	x		-	-	-	x	-

```
LDA        #% 10101110
BIT        Test
Test:  .Byte % 01010000
```

ergibt : N = 0
V = 1
und wegen
virtuellem Ergebnis % 00000000
Z = 1

5. Einschub für Praktikum
Block - Codes

	gewichtet			einstufig		
Dezimal digit	BCD 8421	2421	Excess-3	Gray	Excess-3 gray	Glixon
0	0000	0000	0011	0000	0010	0000
1	0001	0001	0100	0001	0110	0001
2	0010	0010	0101	0011	0111	0011
3	0011	0011	0110	0010	0101	0010
4	0100	0100	0111	0110	0100	0110
5	0101	1011	1000	0111	1100	0111
6	0110	1100	1001	0101	1101	0101
7	0111	1101	1010	0100	1111	0100
8	1000	1110	1011	1100	1110	1100
9	1001	1111	1100	1101	1010	1000
Pseudotetraden						
10	1010			1111		
11	1011			1110		
12	1100			1010		
13	1101			1011		
14	1110			1001		
15	1111			1000		


0
15
1
14
2
13
3
12
4
11
5
10
6
9
7
8
Ablesefenster

V.

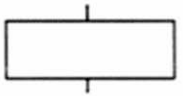
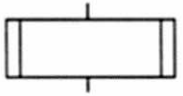
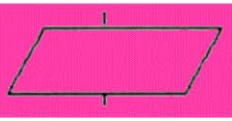
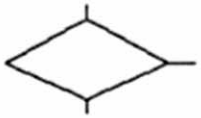
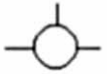
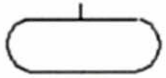
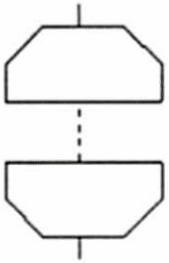
Strukturiertes Programmieren

INHALT

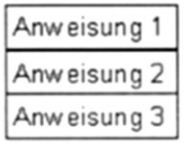
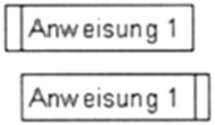
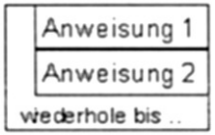
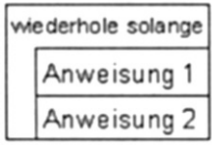
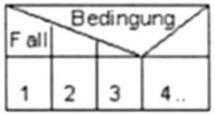
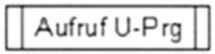
1. Programmstruktur Diagramme
2. Optimieren von Schleifen
3. andere Strukturelemente

1. Programmstruktur Diagramme

Sinnbilder für Programmablaufpläne

Sinnbild	Benennung
	Operation allgemein
	Unterprogrammaufruf
	Ein- bzw Ausgabe nur in Programmnetzen
	Verzweigung
	Übergangsstelle
	Grenzstelle
	Ablauflinien
	Schleifenbegrenzung für zählergesteuerte Wiederholungen

Elemente von Struktogrammen

Sinnbild	Benennung
	Folgeblock für Wertzuweisungen, Rechenoperationen und Bildschirmbefehle
	Eingabeanweisung (nicht genormt) Ausgabeanweisung (nicht genormt)
	Wiederholungsstruktur mit Endebedingung
	Wiederholungsstruktur mit Anfangsbedingung
	Verzweigungsblock a) einseitig b) zweiseitig
	Verzweigungsblock mehrfach
	Aufruf einer Subroutine a) Funktion b) Prozedur

2. Optimierung von Schleifen

Schleifen in höheren Programmiersprachen:

drei Typen :

repeat ….. until ;	***Schleifen Parameter beliebig ,*** ***Ende bei konkreter Parameter-bedingung***
while ….. do ;	***Schleife bei konkreter Parameterbedingung ,*** ***Ende bei beliebiger Abweichung***
for …. to …. step …. do ;	***Parameter mit Schrittweite durchzählen,*** ***Ende wenn Endwert erreicht***

Diese Schleifentypen sind bei Mikroprozessoren nicht anwendbar, da Mikroprozessoren i.a. Schleifen nicht unmittelbar unterstützen.

bis auf wenige Ausnahmen :

- **repetierende E/A-Befehle beim Z80 Prozessor**
- **rep-Befehle bei den 80x86 Prozessoren**
- **Wiederholungsbefehle bei PDP-11 und VAX-11**

Sonst allenfalls auf der Ebenen der Assembler-direktiven zu finden.

Eine Schleife verfügt über 3 markante Stellen

1. **Einsprungstelle (E)**
2. **Absprungstelle (A)**

Dies bedingt im Assembler die Benutzung von „branch“- Befehlen, d.h. es wird immer ein Abbruchtest durchgeführt.

3. Schleifenbegrenzung, programmtechnisch gegeben durch :

(1) Schleifenanfang (SA)

(2) Schleifenende (SE)

Der innere Schleifenkörper wird mit SK abgekürzt.

Die Schleifenbegrenzungen wird auch als Schnittstellen bezeichnet, da
letzter Befehl der Schleife
(des vorangestellten Strukturblocks)
mit erstem Befehl des nachgestellten Strukturblockes
(der Schleife)
verbunden wird.

Ist das Schleifenende nicht die Absprungstelle (SA ≠ A), wird vom Schleifenende ein unbedingeter Sprung zum Schleifenanfang (SA) durchgeführt. SE ist dann mit U markiert.

Mit Hilfe der markanten Stellen lassen sich drei sinnvolle Kategorien von Schleifen einführen :

I. Schleifentest ist zugleich Einsprungstelle

(E = A)

sogenannte Abweisschleifen (while – Schleifen)

II. Einsprung auf den ersten Befehl, der auf die Absprungstelle folgt.

(E = A + 1)

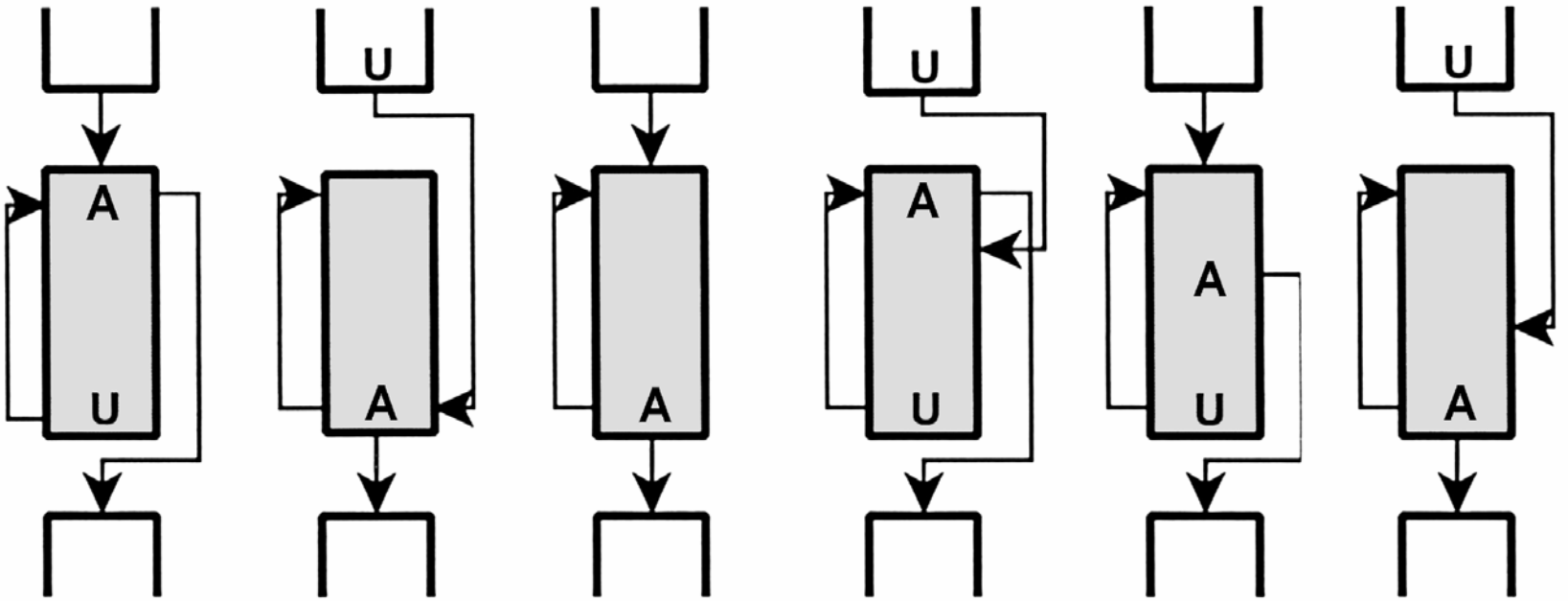
Schleife wird mindestens einmal durchlaufen
(repeat … until - Schleifen)

III. Einsprung hat keine Korrelation zur Absprungstelle.

a) E = SA , E = U ist hierin enthalten, da U unbedingter Sprung zu SA bedeutet.

b) E = SK

Darstellung der wichtigsten Schleifen-Varianten



Zusammenfassung aller möglichen Varianten

Kategorie	Konfigurationsbeschreibung				Bewertung
		Eingangsstelle (E)	Absprungstelle (A)	Sprung von SE nach SA	
I Abweisung	a	SA = A	*siehe links*	U	**akzeptabel**
	a'	SE	*SA**	U	**unsinnig**
	b	SE = A ; mit U	*siehe links*	T	**gut**
	c	SK ; mit U	*siehe links*	U	**mäßig**
II	a	SA	SE	T	**gut**
	a'	SE ; mit U	SE - 1	U	**unsinnig**
	b	SA+1 ; mit U	SA	U	**akzeptabel**
	c	SK_E ; mit U	SK_E - 1	U	**mäßig**
III	a	SA	SK_A	U	**akzeptabel**
	a'	SE ; mit U	SK_A *	U	**unsinnig**
	b	SK_E ; mit U	SE	T	**gut**
	c	SK_E ; mit U	SA	U	**akzeptabel**
	d	SK_E ; mit U	SK_A vor SK_E	U	**mäßig**
	e	SK_E ; mit U	SK_A nach SK_E	U	**mäßig**
	f	SE ; mit U	SK_A	U	**mäßig**

Diese Fälle sind in obiger Graphik dargestellt

* **Aufgrund des unmittelbaren unbedingten Sprunges vom Eingangspunkt SE nach SA ist dieser Fall identisch mit dem Fall a, d.h. es ist praktisch E=SA statt SE. Wegen der zwei direkt aufeinander folgenden unnötigen Sprünge, sind diese Fälle unsinnig**

Version I a

			Zyklen
	Bef_V_E	; letzter Befehl der Vorgängerstruktur	
	Ldy # n	; Schleifen Initialisierung	
SA :	beq NaSt	; Abbruch/Abweisung wenn Z-flag gesetzt	2 (3)
	Befehl_S_1	; 1. Befehl der Schleife	} k
	:		
	:		
	dey	; gegebenenfalls Z-flag setzen	
SE :	jmp SA	; unbedingter Sprung zum Schleifenanfang	3
NaSt:	Befehl_N_1	; nur durch Sprung von SA erreichbar	
	:		
	:		

Zyklenbilanz : $N_{Ia} = n \cdot (2+k+3) + 3$

Version I b

			Zyklen
	Bef_V_E	; letzter Befehl der Vorgängerstruktur	
	Ldy #n	; Schleifen Initialisierung	
	jmp SE	; Sprung in die Schleife	3
SA :	Befehl_S_1	; 1. Befehl der Schleife	} k
	:		
	:		
	dey	; gegebenenfalls Z-flag setzen	
SE :	bne SA	; Abbruch/Abweisung, wenn Z-flag gesetzt	3 (2)
NaSt:	Befehl_N_1	; ohne Sprung von SE erreichbar	
	:		
	:		

Zyklenbilanz : $N_{Ib} = 3 + n \cdot (3+k) + 2$

Version IIa

			Zyklen
	Bef_V_E	; letzter Befehl der Vorgängerstruktur	
	Ldy #n	; Schleifen Initialisierung	
SA :	Bef_S_1	; 1. Befehl der Schleife (keine Abweisung !)	} k
	:		
	:		
	dey	; gegebenenfalls Z-flag setzen	
SE :	bne SA	; Abbruch, wenn Z-flag gesetzt	3 (2)
NaSt:	Bef_N_1	; ohne Sprung von SE erreichbar	
	:		
	:		

keine Sprünge !

Zyklenbilanz : $N_{IIa} = n \cdot (k+3) - 3 + 2$

Zusammenfassung :

Gerade beim Assembler dürfen Schleifen nicht unabhängig von Vorgänger- bzw. Nachfolgerblock betrachtet werden.

Zeit-Aspekt :

- **Abweisschleifen (n=0 möglich)**
  - **Ia E = SA**

$$N_{Ia} = n \cdot (k+5) + 3$$

nur ím Falle der Abweisung günstig

  - **Ib E = SE**

$$N_{Ib} = n \cdot (k+3) + 5$$

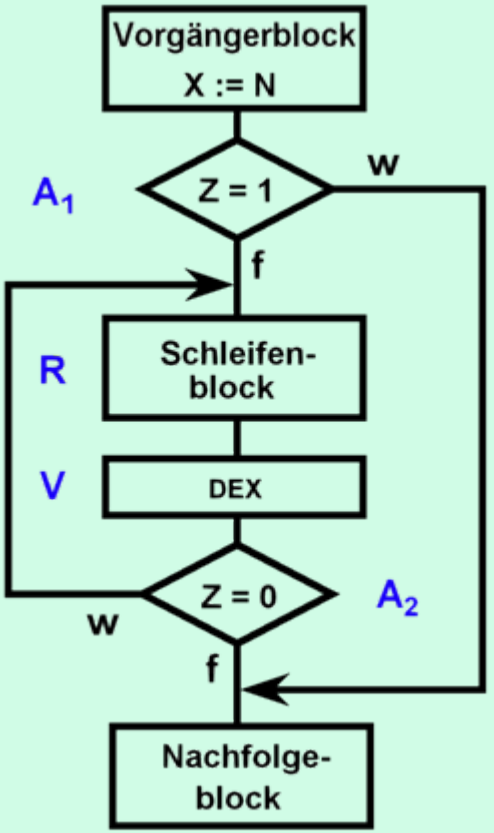
- **noch schneller (n ≥ 1)**

IIa E = SA

$$N_{IIa} = n \cdot (k+3) - 3 + 2$$

Lösung für IIa mit Abweisung

```
        Bef_V_E     ; letzter Befehl der Vorgängerstruktur
        Ldy #n      ; Schleifen Initialisierung
        beq  NaSt   ; Abweisung, wenn Z-flag gesetzt
SA :    Bef_S_1     ; 1. Befehl der Schleife  (ohne Sprung)
          :
        dey         ; gegebenenfalls Z-flag setzen
SE :    bne  SA     ; Abbruch, wenn Z-flag gesetzt
NaSt:   Bef_N_1     ; ohne Sprung von SE erreichbar
          :
```



Im Vergleich zu IIa Abweis-Verzweigung hinzugefügt

⇒ (3) Zyklen mehr

Speicherplatz-Aspekt :

- **Unterschied zwischen Ia und Ib :**

  jmp - Befehl in Vorgängerstruktur bzw. Schleife

  ⇒ gleicher Speicherplatz

- **Schleife IIa im Vergleich zu Typ Ia/Ib**

  kein jmp - Befehl

  ⇒ 3 byte weniger Speicher

  Nachteil : keine Eingangsabweisung

Ein Wort zu Sprüngen bzw. Verzweigungen

Unbedingter Sprung nur mit jmp-Befehl möglich ?

- **Im 650X und 680X Prozessoren scheint das der Fall zu sein.**
- **Andere Prozessoren, auch schon der 65C02, kennen den bra-Befehl (unbedingte Verzweigung)**

Unterschied zwischen Sprung und Verzweigung ?

- **jmp : nur absolute Adressierung ; 3 Bytes**
- **bra : nur relative Adressierung ; sogar nur 2 Bytes**

Nachteil : begrenzte Sprungweite von 127 / -128

Wie kann bra im 650X simuliert werden ?

```
clv          ;  1 Zyklus
bvc          ;  3 Zyklus
```

Warum 3 ?

Andere Kategorie Einteilung :

Elemente einer Schleife sind in jedem Fall Rechnung (R) und Abfrage (A), können aber auch eine Veränderung (V) (Laufparameter) enthalten, die in der Vorgängerstruktur initialisiert werden muss. Die Anordnung dieser Elemente ermöglicht eine andere Kategorisierung.

1. **Iterative Schleifen**
   enthalten nur Abfrage und Rechnung ,
   in der Regel als AR Anordnung reine Abweisschleifen.

2. **Induktive Schleifen**
   enthalten zusätzlich eine Veränderung.

6 Möglichkeiten der Anordnung von A , R , V

3! = 6

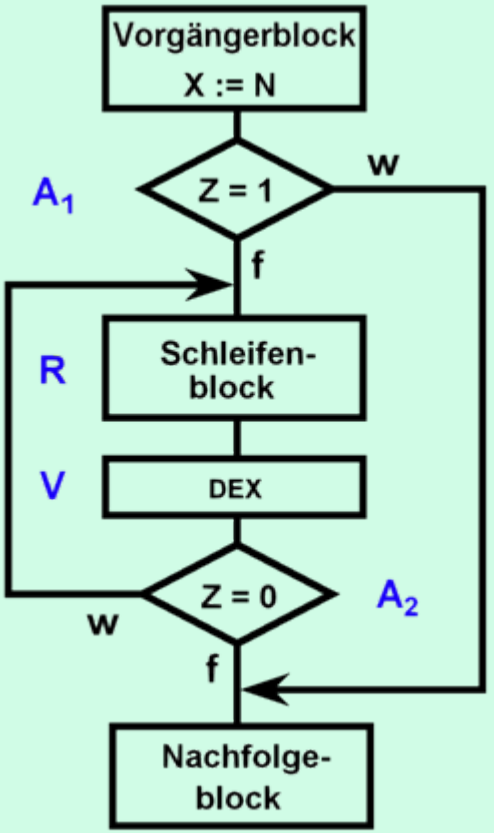
Die vorne stehende Einheit beginnt mit der Eingangsstelle,
die Abfrage entspricht der Ausgangsstelle

Typ	obige Kategorie	Bemerkung
ARV / AVR	I	Abweisschleifen
RVA / RAV	II	mindestens einmal durchlaufen
VAR / VRA	III	dynamisch ungünstig

Nicht alle Permutationen sind immer sinnvoll, z. B. eine Veränderung direkt nach der Abfrage bei abwärts zählende Schleifen im Assembler.
Ebenso ist eine Veränderung beim Einstieg dynamisch ungünstig.

A_1RVA_2 - Lösung

```
        Bef_V_E      ; letzter Befehl der Vorgängerstruktur
        Ldy #n       ; Schleifen Initialisierung
        beq  NaSt    ; Abweisung, wenn Z-flag gesetzt
-------------------------------------------------------------
SA :    Bef_S_1      ; 1. Befehl der Schleife  (ohne Sprung)
          :
        dey          ; gegebenenfalls Z-flag setzen
SE :    bne  SA      ; Abbruch, wenn Z-flag gesetzt
-------------------------------------------------------------
NaSt:   Bef_N_1      ; ohne Sprung von SE erreichbar
          :
```

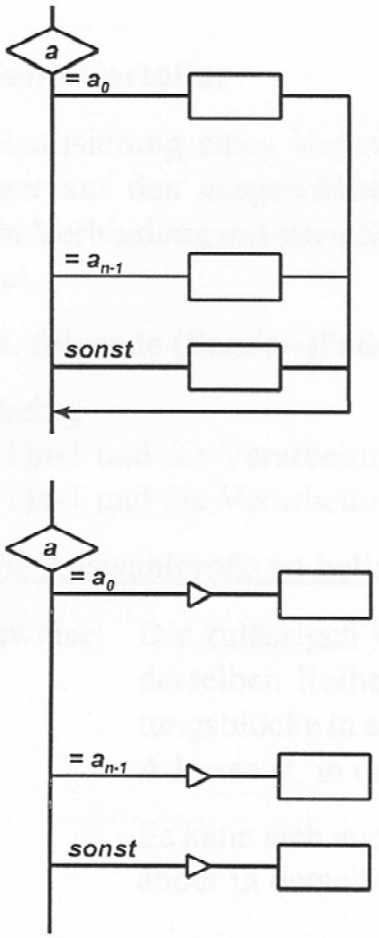


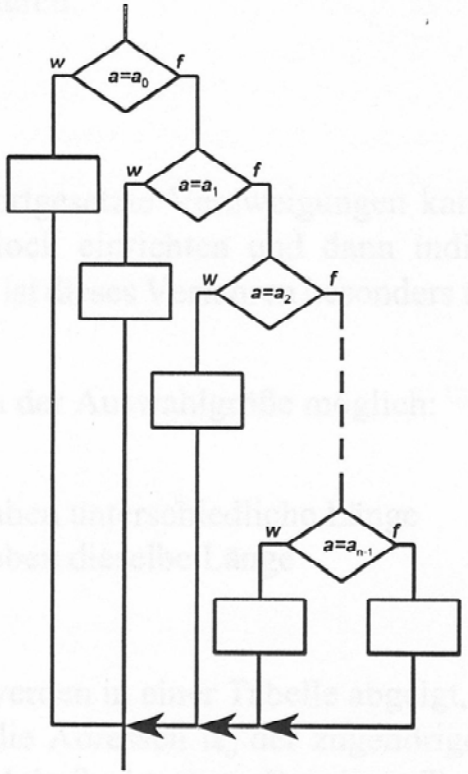
Im Vergleich zu IIa Abweis-Verzweigung hinzugefügt

$\Rightarrow$ **(3) Zyklen mehr**

3. andere Strukturelemente

Darstellungsmöglichkeiten
von
Verteiler Strukturblöcken

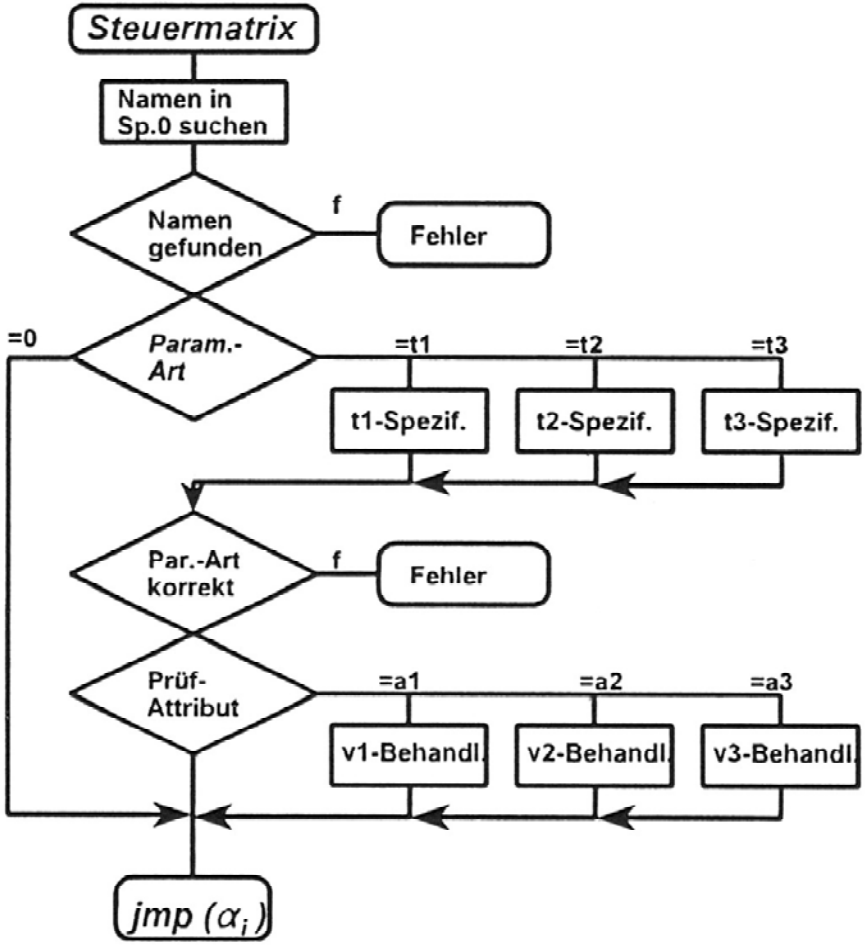




Realisierung

Beispiel für Steuermatrix

0	1	2	3	4
Name	Sprg.-Adr.	Par.-Art	Prüf-attr.	Prüfgröße
Aktionsteil		*Beschreibungsteil*		
A	α_1	t_1	a_1	v_1
B	α_2	t_2	a_2	v_2
C	α_3	t_3	a_3	v_3



V.

Unterprogramm-techniken

INHALT

1. Parameterübergabe
2. Koroutinentechnik
3. Rahmentechnik
4. Wiedereintrittsfähigkeit
5. Multiplikationsalgorithmen

1. Parameterübergabe :

Parameter können sein :

- **Rechenwerte bzw. Zeiger darauf**
- **Indizes, Laufparameter**
- **Zählerstände**
- **Kennwerte von Speicherzellen**

Bei der 650X Prozessor Familie sind Parameterlisten im Unterprogrammaufruf nicht möglich (im Gegensatz zu Macros).

Wie lassen sich hier Parameter übergeben ?

1. **über Register**
2. **über RAM – Speicherzellen**
   a) **Absolute Adressen (nicht wiedereintrittsfähig [wef])**
   b) **Rampen (Speicherbereich im Programmcode) (nicht wef)**
   c) **Argumentenzeiger (wef, wenn auf Stapel geretttet)**
   d) **Deskriptoren (Datentyp, Länger und Zeiger) aufwendig, in alten VAX realisiert.**
3. **über den Stapel**

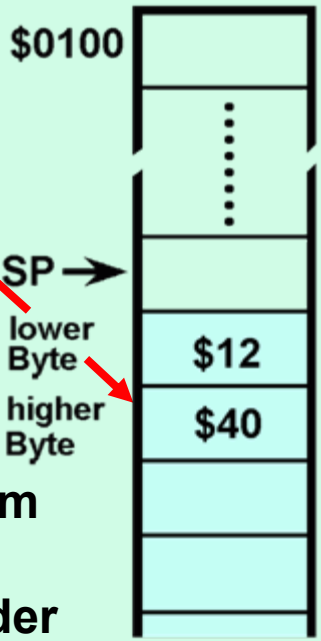
2. Koroutinen

zur Erinnerung :

Beispiel :

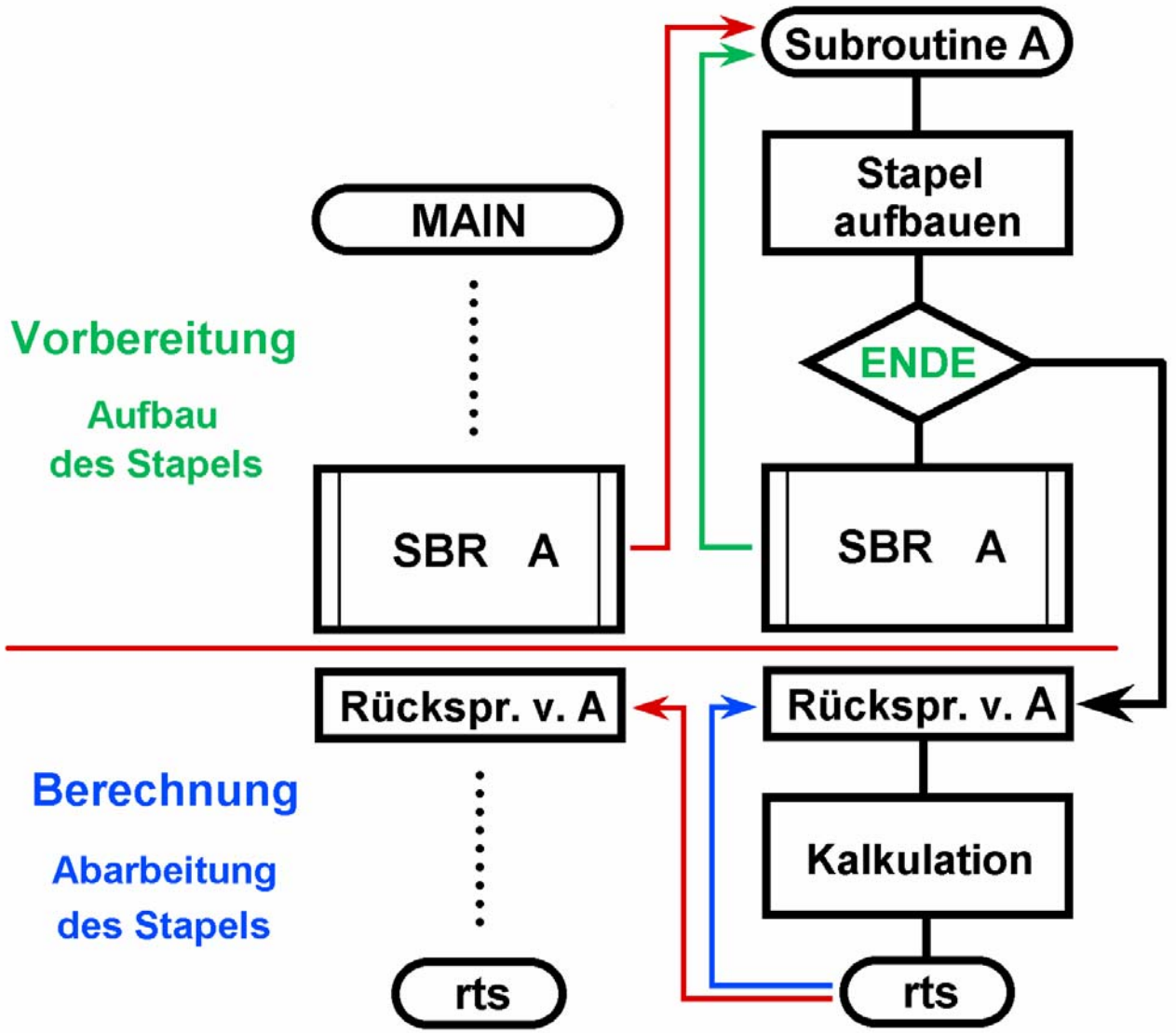
```
                :
$4010   JSR  Potenz ; $4012 wird als Rücksprungadresse ρ
$4013   Befehl_cont                  auf den Stapel gebracht
                :
                :

$wxyz   Potenz
                :
                :
                rts
```

; Speicheradresse $4012 wird vom Stapel genommen und in den Programmzähler geschrieben, der dann um „1“ inkrementiert wird, also auf $4013 gesetzt wird.

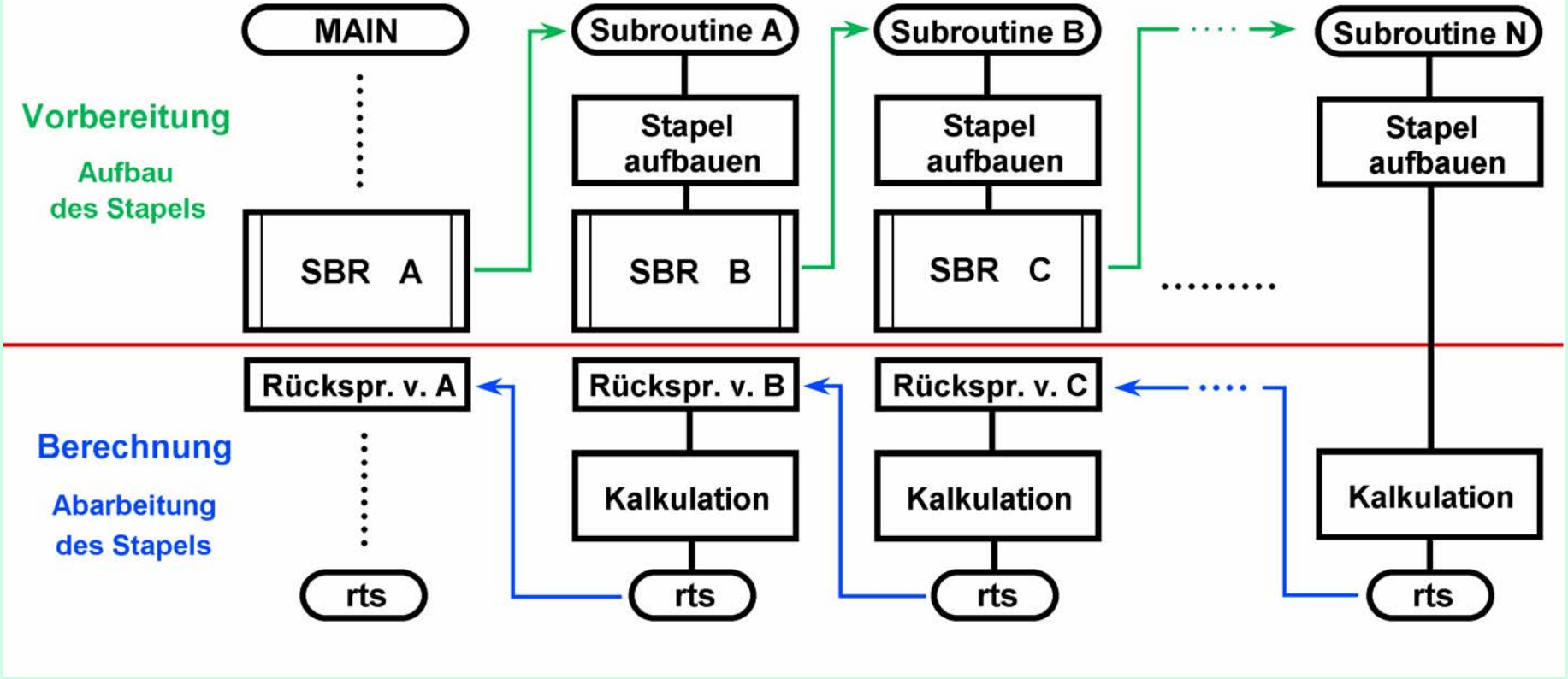


Unterprogramme werden durch Sprünge angesprochen, d.h. , Programmzähler muss neu beschrieben werden.

Rekursive Technik

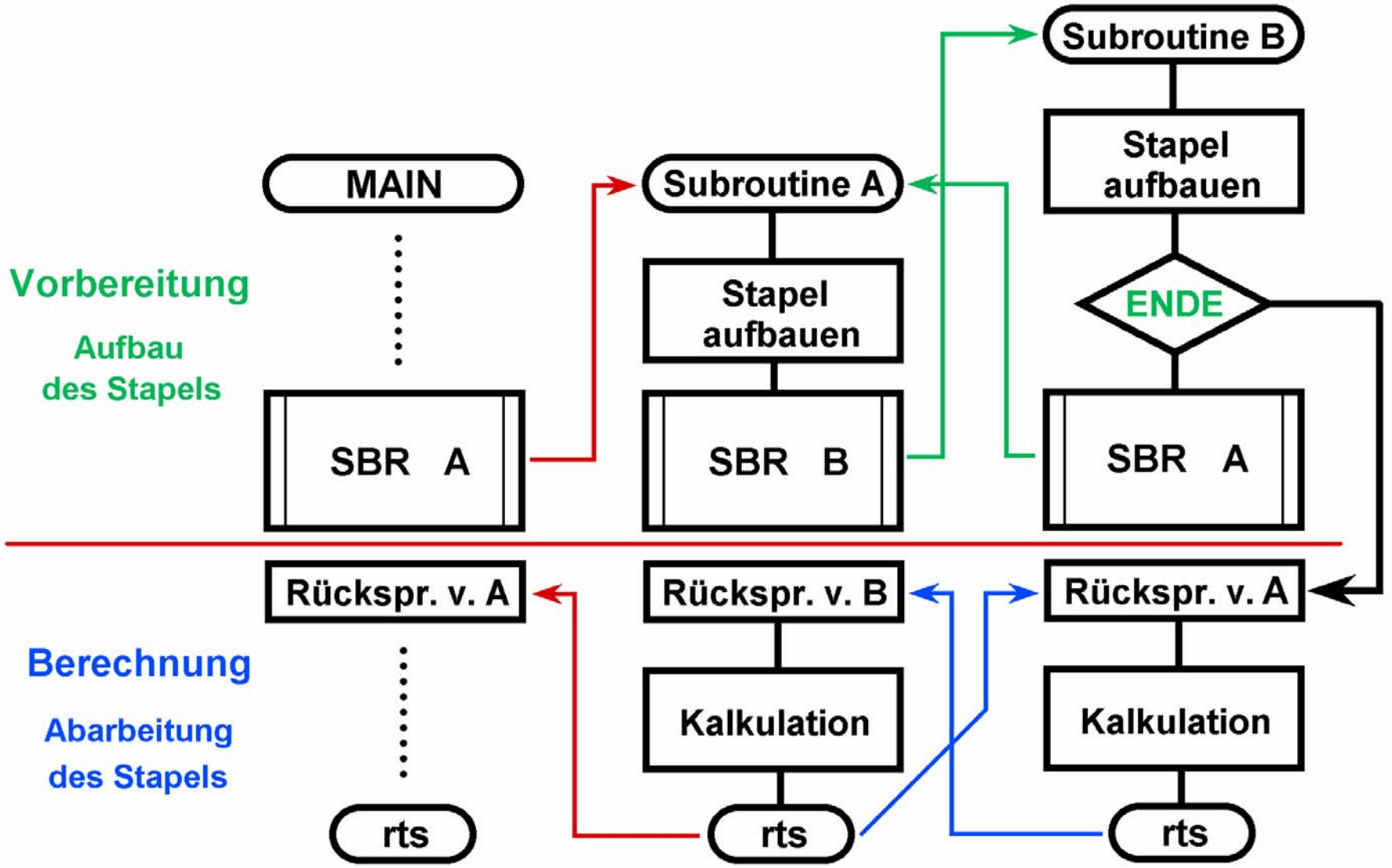


KOROUTINEN - TECHNIK



Koroutinen Technik

Paar Hin- und Rücksprung



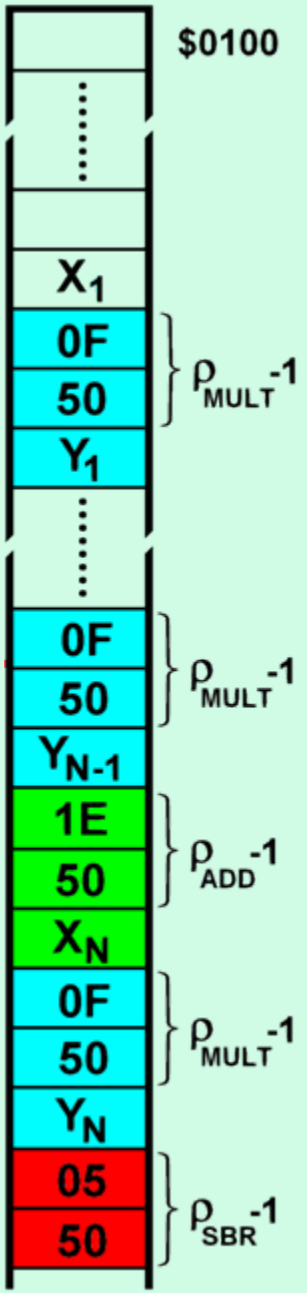
Beispiel zur Paar-Koroutine :

FORMEL :

$$F = Y_N(X_N+Y_{N-1}(X_{N-1}+\ldots+Y_2(X_2+Y_1(X_1+0))\ldots))$$

```
5000 SBR:   LDX  TabX
5003        JSR  MULT
5006        JMP  ENDE

5009 MULT:  LDA  TABY, x
500C        PHA
500D        JSR  ADD

5010        PLA
5011        JSR  Mult_Erg_mal_Y
5014        RTS

5015 ADD:   LDA  TabX, x
5018        PHA
5019        DEX
501A        BEQ  WEITER
501C        JSR  MULT

501F WEITER: PLA
5020        CLC
5021        ADC  Ergebnis
5024        LDA  #0
5026        ADC  Ergebnis + 1
5029        RTS

502A ENDE:  RTS

502B    TabX:     .byte  N , X1 , X2 , ..... XN
502B+N  TabY:     .byte  N , Y1 , Y2 , ..... YN
502B+2N Ergebnis: .word  $0000
```

Inhalt des Stapels



3. Rahmentechnik

Kann man auf tiefer liegende Stapelzellen zugreifen ?

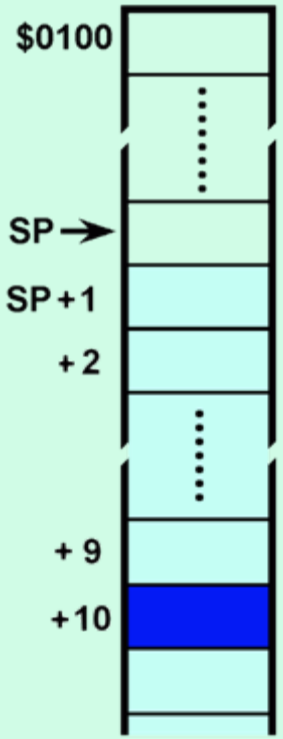
- Natürlich nicht mit PLA oder PHA ;
- aber mit LDA $01xy lässt sich jede Stapelzelle ansprechen

Woher weiß man die exakte Adresse, in der gewünschter Wert steht ?

Wenn die Differenz zum Stapelzeiger bekannt ist, lässt sie die gewünschte Adresse berechnen.

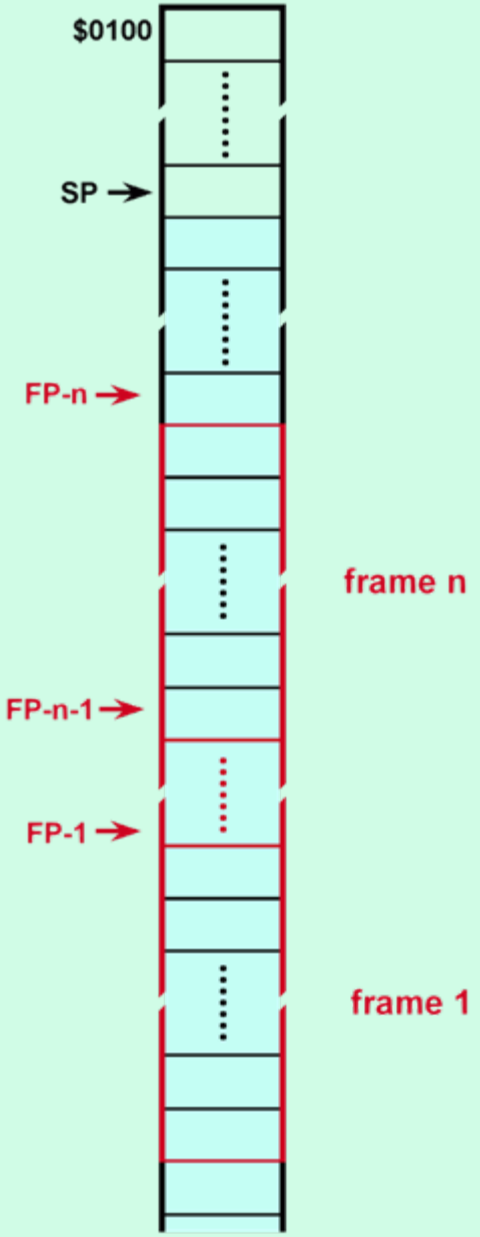
Beispiel :

```
TSX
TXA
CLC
ADC #10
TAX
LDA $0100 , x
```



Rahmen - Technik

- **einzelne frames können unterschiedliche Länge haben**



Wie kann ein Unterprogramm indirekt aufgerufen werden ?

- **Hierzu wird die absolut indirekte Adressierung des jmp-Befehls ausgenutzt.**

Beispiel :

```
                    •
                    •
        LDA #<Subr
        STA Pointer          ;  Programm gesteuerte Festlegung
        LDA #>Subr           ;  des Zeigers
        STA Pointer + 1
                    •
                    •
        JSR Inter_Subr       ; direkter Unterprogrammaufruf
   ρ :              •        ; auf Stapel gelegte Rücksprungadresse
                    •
                    •
Inter_Subr : JMP(pointer)    ; indirekter Sprung zum eigentlichen UP
        RTS                  ;                    Rücksprung zu ρ
     Subr :         •        ; ausgewähltes
                    •        ; Unterprogramm
                    •
        JMP Inter_Subr + 3   ; Sprung zu RTS von „Inter_Subr“ ,
                             ; da JMP(Pointer) 3 Bytes belegt.
```

4. Wiedereintrittsfähigkeit

Wiedereintrittsfähigkeit :

In einem normalen Programmpaket reicht es aus, darauf zu achten, genügend Variablen bzw. deren Speicherplatz festzulegen, um ein Überschreiben von später benötigter Information zu verhindern.

Was passiert aber, wenn der normale Programmablauf durch einen Interrupt zu einem beliebigem, nicht exakt vorhersehbarem Zeitpunkt unterbrochen wird.

Ist die dann aufgerufene Dienstroutine völlig isoliert von dem übrigen Programm, ist dies unkritisch.

Benutzt die Dienstroutine aber Teile (in der Regel Unterprogramme) , die auch in der Hauptprozess benutzt werden, kann es zum Verlust von Daten kommen, wenn der Interrupt gerade dann ausgelöst wird, wenn sich der Hauptprozess in einem Teil befindet, der auch von der Dienstroutine benutzt wird.

Solche Programmteile (Unterprogramme) müssen so konzipiert werden, dass sie **wiedereintrittsfähig (reentrant)** sind.

Als Lösung benutzt man den **Stapel** als temporären Speicher.

- Status-Register, Akkumulator , X- und Y-Register werden automatisch durch Prozessor bzw. Blos bei einem IRQ auf den Stapel gerettet. Diese Register können daher gefahrlos in wiedereintrittsfähigen Unterprogrammen benutzt werden.
- **Alle übrigen Ausgangswerte und Ergebnisse müssen zusätzlich im Stapel abgelegt werden.**

A Extremfall

Das gesamte Programm, dass durch Interrupts unterbrochen werden kann, ändert nur Inhalt von Stapel-Adressen und die Register A , X und Y.

$\Rightarrow$ IRQ-Dienstroutine muss nur diese Register retten

Nachteile :

- Programmteile, in den IRQ‘s erlaubt sind, sind i.a. so groß, dass Benutzung von Stapel nicht praktikabel ist (Überlauf).
- Indirekte Adressierung nicht über Stapel möglich, nur über Seite 0 .

B Unterprogramm wiedereintrittsfähig

Unterrpogramm, in dem IRQ auftritt, wird auch von IRQ-Dienstroutine aufgerufen.

Diese Unterprogramme müssen wiedereintrittsfähig sein, damit sie nach Beendigung der IRQ-Dienstroutine korrekt weiterlaufen.

Vorsicht bei indirekter Adressierung im Unterprogramm, dann muss in der Interrupt-Dienstroutine auch der Zeiger gerettet werden.

Das folgende Beispiel zeigt auch den Unterschied zwischen iterativer und rekursiver Anwendung.

iterativ

```
Main :      :
            LDA  #0
            STA weight
            LDA string
            BEQ Weiter
            JSR Gewicht
Weiter :    :
            :

Gewicht :   LSR string
            BEQ Ende
            BCC Gewicht
            INC weight
            CLV
            BVC Gewicht

Ende :      INC weight
            RTS
```

rekursiv

```
Main :      :
            LDA  #0
            STA weight
            LDA string
            BEQ Weiter
            JSR Gewicht
Weiter :    :
            :

Gewicht :   LSR string
            BEQ Zähl
            BCC Gewicht
            JSR Gewicht

Zähl :      INC weight
            RTS
```

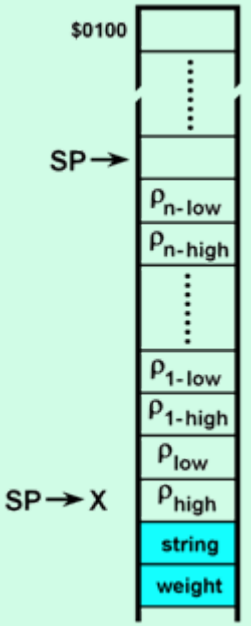
Wiedereintrittsfähig

iterativ

```
Main :      :
            LDA  #0
            PHA
            LDA string
            BEQ Weiter
            PHA
            TSX
            JSR Gewicht
            PLA
Weiter :    PLA
            :
            :

Gewicht :   LSR $0101,x
            BEQ Ende
            BCC Gewicht
            INC $0102,x
            CLV
            BVC Gewicht

Ende :      INC $0102,x
            RTS
```

rekursiv

```
Main :      :
            LDA  #0
            PHA
            LDA string
            BEQ Weiter
            PHA
            TSX
            JSR Gewicht
            PLA
Weiter :    PLA
            :
            :

Gewicht :   LSR $0101,x
            BEQ Zähl
            BCC Gewicht
            JSR  Gewicht

Ende :      INC $0102,x
            RTS
```



5. Algorithmen

zur

Multiplikation

Einige Anmerkungen :

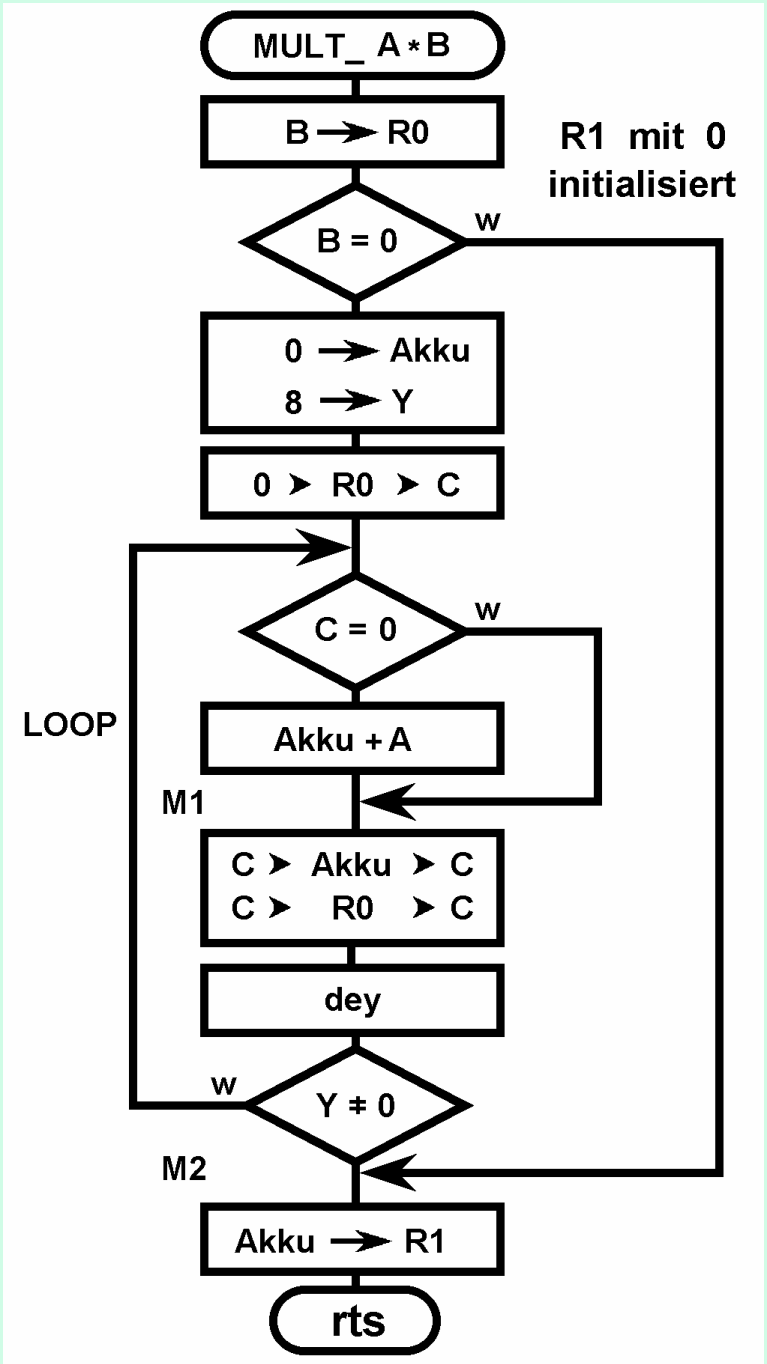
- Multiplikation der beiden 1-Byte Zahlen A und B ergibt maximal ein 2-Byte Resultat ;
  die Speicherzelle R0 wird für das lower Byte und R1 für das higher Byte reserviert.
- B ist die Zahl, deren 1- en bestimmen, wann A entsprechend verschoben addiert wird ;
  Ist eine dieser 1- en ins Carry geschoben worden **(blau)**, so wird im Zyklus eine Addition ausgeführt.
- Manuell wird das Ergebnis sukzessiv von hinten nach vorne erzeugt **(siehe rote Ziffern)**.
- **Diese Ziffern werden im Algorithmus sukzessiv in das lower Byte R0 verschoben.**
- Hier erzeugt der ADC-Befehl nie ein Carry, daher kann anschließend ROR benutzt werden (LSR wäre auch möglich)

Prinzip der Multiplikation

	Befehl	Akk	C	B → R0	
	laden	0 0 0 0 0 0 0 0	0	1 0 0 0 0 1 1 0	
1	lsr R0		0	0 1 0 0 0 0 1 1	0
	C=0				
	ror	0 0 0 0 0 0 0 0	0		
2	ror R0		1	0 0 1 0 0 0 0 1	1
	C=1				
	clc		0		
	ADC A	0 0 0 1 1 0 1 1	0		
	ror	0 0 0 0 1 1 0 1	1		
3	ror R0		1	1 0 0 1 0 0 0 0	1
	C=1				
	clc		0		
	[A]	0 0 0 1 1 0 1 1			
	ADC A	0 0 1 0 1 0 0 0	0		
	ror	0 0 0 1 0 1 0 0	0		
4	ror R0		0	0 1 0 0 1 0 0 0	0
	C=0				
	ror	0 0 0 0 1 0 1 0	0		
5	ror R0		0	0 0 1 0 0 1 0 0	0
	C=0				
	ror	0 0 0 0 0 1 0 1	0		
6	ror R0		0	0 0 0 1 0 0 1 0	0
	C=0				
	ror	0 0 0 0 0 0 1 0	1		
7	ror R0		0	1 0 0 0 1 0 0 1	0
	C=0				
	ror	0 0 0 0 0 0 0 1	0		
8	ror R0		1	0 1 0 0 0 1 0 0	1
	C=1				
	clc		0		
	[A]	0 0 0 1 1 0 1 1			
	ADC A	0 0 0 1 1 1 0 0	0		
	ror	0 0 0 0 1 1 1 0	0		
9	ror R0		0	0 0 1 0 0 0 1 0	0
		Akk → R1		R0	
ERGEBNIS		**0 0 0 0 1 1 1 0**		**0 0 1 0 0 0 1 0**	

Manuell

```
          A       *       B
0 0 0 1 1 0 1 1   *   1 0 0 0 0 1 1 0
                  0 0 0 0 0 0 0 0
                  0 0 0 1 1 0 1 1
                  0 0 0 1 1 0 1 1
                0 0 0 1 1 0 1 1
              0 0 0 1 1 0 1 1
            0 0 1 0 1 0 0 0
          0 0 1 0 1 0 0 0
        0 0 0 1 1 0 1 1
      0 0 0 1 1 1 0 0
    0 0 0 1 1 1 0 0
  0 0 0 1 1 1 0 0
1 1 1 0 0 0 1 0 0 0 1 0
```




```
        LDA B
        STA R0
        BEQ  M2
        LDA #0
        LDY #8
        LSR R0

LOOP :  BCC M1
        CLC
        ADC A
M1:     ROR
        ROR R0
        DEY
        BNE  LOOP

M2:     STA R1
        RTS
```

Wiedereintrittsfähige Version

Dieses Beispiel enthält drei Teile :

das eigentliche Unterprogramm und zwei strukturgleiche Teile, in denen das Unterprogramm vom Hauptprozess (Main) bzw. von der IRQ-Dienstroutine aufgerufen wird.

Ferner ist es in diesem Beispiel nicht nötig, Zwischenergebnisse in temporären Speicherzellen abzulegen. Dies müsste gegebenenfalls natürlich auch auf dem Stapel geschehen.

wiedereintrittsfähige Lösung

obige Lösung

```
MULT:   LDA B
        STA R0
        BEQ  M2
        LDA #0
        LDY #8
        LSR R0

LOOP :  BCC M1
        CLC
        ADC A
M1:     ROR
        ROR R0
        DEY
        BNE  LOOP

M2:     STA R1
        RTS
```

R0 und R1 sind Mit 0 initialisiert

```
        MAIN:   :
                :
4108            LDA A
410B            BEQ C1
410D            PHA
410E            LDA B
4111            BEQ C2
4113            PHA
4114            JSR MULT
4117            STA R1
411A            PLA
411B            STA R0
411E  C2:       PLA
411F  C1:       NOP
                :

        IRQ :   :
                :
4500            LDA F1
4503            BEQ CON1
4505            PHA
4506            LDA F2
4509            BEQ CON2
450B            PHA
450C            JSR MULT
450F            STA ERG1
4512            PLA
4513            STA ERG0
4516  CON2:     PLA           ; Stapelbereinigung
4517  CON1:     NOP
                :
```

```
4200  MULT:     LDA #0
4202            LDY #8
4204            TSX
4205            LSR $0103 , X
4208  LOOP:     BCC  M1
420A            CLC
420B            ADC $0104 , X
420E  M1:       ROR
420F            ROR $0103 , X
4212            DEY
4213            BNE  LOOP
4215            RTS
```

IRQ

MULT-Aufruf aus IRQ-Dienstroutine

Aufruf der IRQ-Dienstroutine

MULT-Aufruf aus MAIN (Hauptprozess)

Prinzip der Methode von Booth

```
 1 0 0 1 1 1 0 1   *   1 0 1 0 0 1 0 1
                       1 0 0 1 1 1 0 1
                   1 0 0 1 1 1 0 1
               1 0 0 1 1 1 0 1
           1 0 0 1 1 1 0 1
       0 1 1 1 2 1 2 1 1 1 1 1
 1 1 0 0 1 0 1 0 0 1 1 0 0 0 1
```

$$X * \underbrace{1\,1\,\ldots\,1}_{\text{n-mal}} = (2^n-1) * X = 2^n * X - X$$

```
  1 0 1 0 0 1 0 1 0 0 0
-       1 0 1 0 0 1 0 1
  1 0 0 1 0 0 0 0 1 1

    1 0 1 0 0 1 0 1 0
-       1 0 1 0 0 1 0 1
        1 0 1 0 0 1 0 1
```

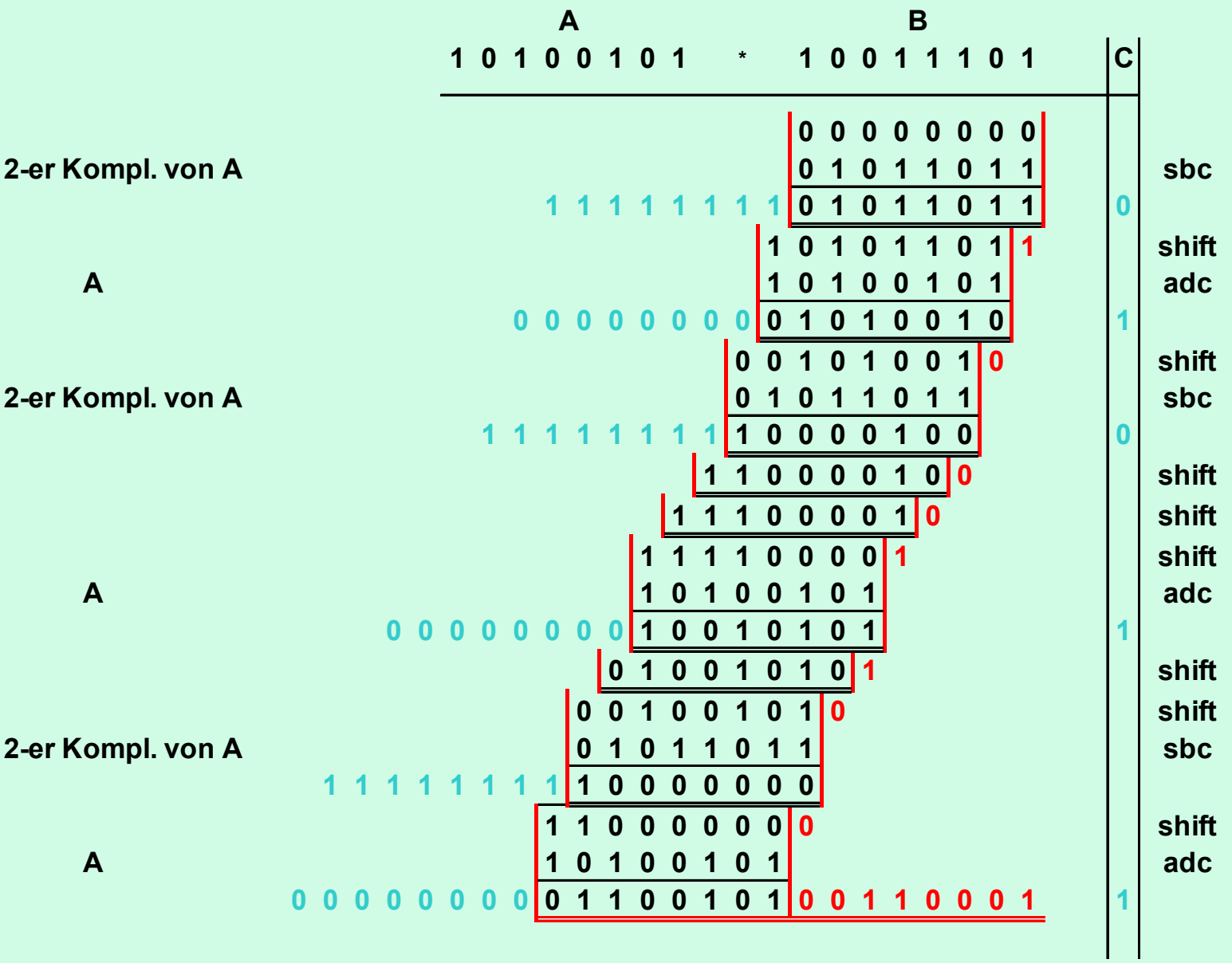
```
 1 0 1 0 0 1 0 1   *   1 0 0 1 1 1 0 1
                       1 0 1 0 0 1 0 1
                   1 0 0 1 0 0 0 0 1 1
               1 0 1 0 0 1 0 1
           0 1     1 0 1       1 1
 1 1 0 0 1 0 1 0 0 1 1 0 0 0 1
```

Einige Anmerkungen :

- **Die Darstellung ist nahezu analog zu erstem Verfahren; daher gilt das Wesentliche von oben auch hier.**
- **Bei der Methode von Booth müssen vier Fälle unterschieden werden**
  - **erste „1“ ; hier wird bereits die Subtraktion vorgenommen.**
  - **folgende „1“ ; nur Shift bei negativem Vorzeichen**
  - **erste „0“ nach 1-er Block , hier wird die Addition durchgeführt.**
  - **folgende „0“ ; nur Shift bei positivem Vorzeichen**

  Gelöst wird dies mit zwei Schleifen (s.u.).
- **Im folgenden werden zwei Beispiele diskutiert, erstes mit voranstehender „1“ , zweites mit mehreren voranstehenden „0-en“**

1. Beispiel zur Methode von Booth



Weitere Anmerkungen :

- Da die Additionen und Subtraktionen von A alternierend durchgeführt werden, allerdings durch Rechtsverschiebungen unterbrochen, ist das Ergebnis der **Subtraktion** immer **negativ** und das der **Addition** immer **positiv**.

  Anders ausgedrückt :
  Das Ergebnis der Subtraktion liegt im Intervall

  $$-A < \text{Akku} < 0$$

  und das der Addition im Intervall

  $$0 < \text{Akku} < A$$

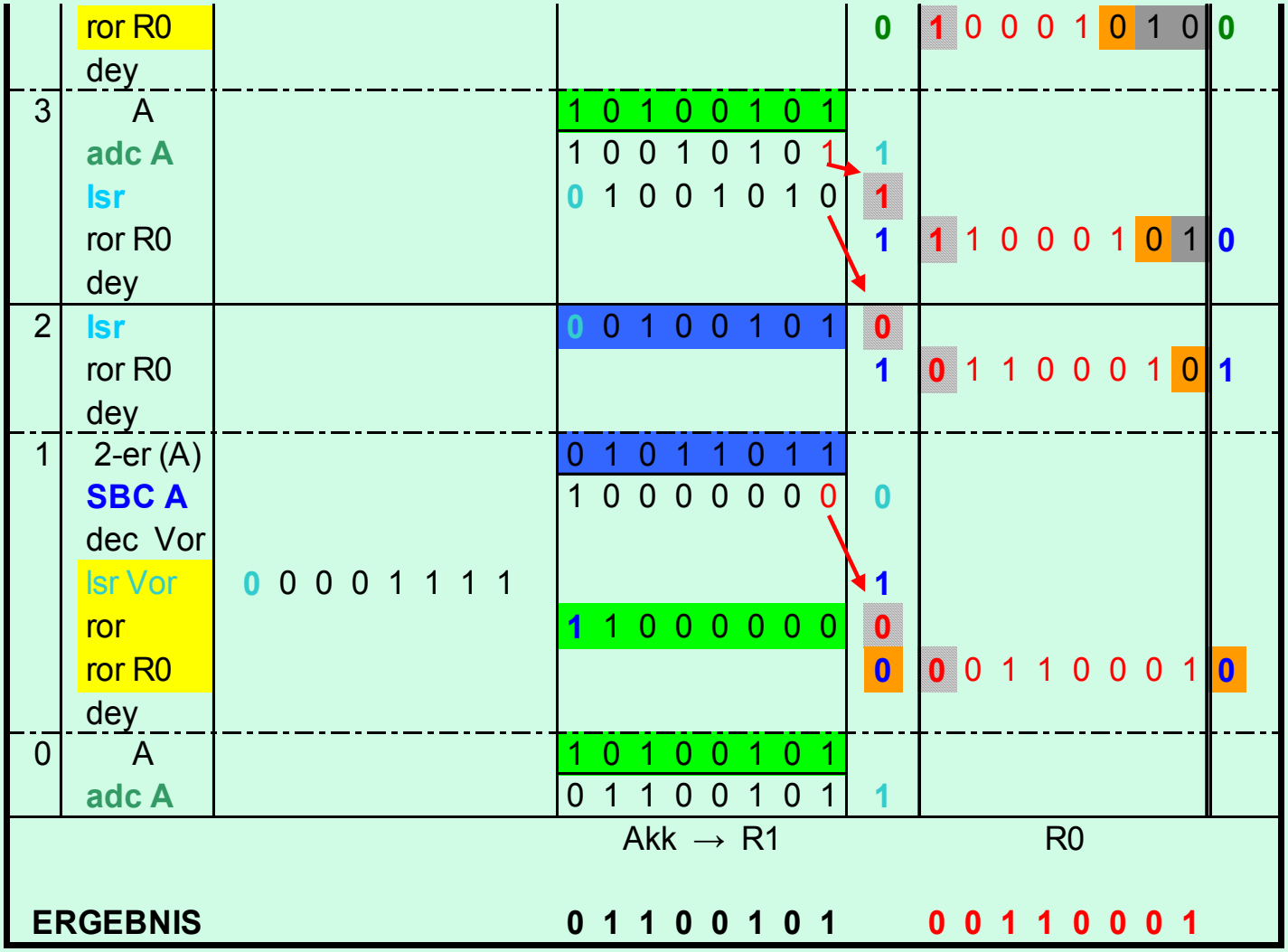
- In den folgenden Folien sind die beiden Bytes, die subtrahiert werden, **grün** und die, die addiert werden, **blau** unterlegt.

- Das orange unterlegte Bit „0“ ist nahezu bedeutungslos , es wird durch R0 durchgeschoben. Danach bewirkt es allerdings, dass vor dem abschließenden ADC kein CLC nötig ist
- Die gelb unterlegten Befehle gehören zur inneren Schleife (s.u. Programme)
- Verfügt B , wie im 1. Beispiel, über eine voranstehende „1“, so muss ausnahmsweise bei „C = 1“ eine Addition als abschließende Prozedur durchgeführt werden.

Programmumsetzung für Beispiel

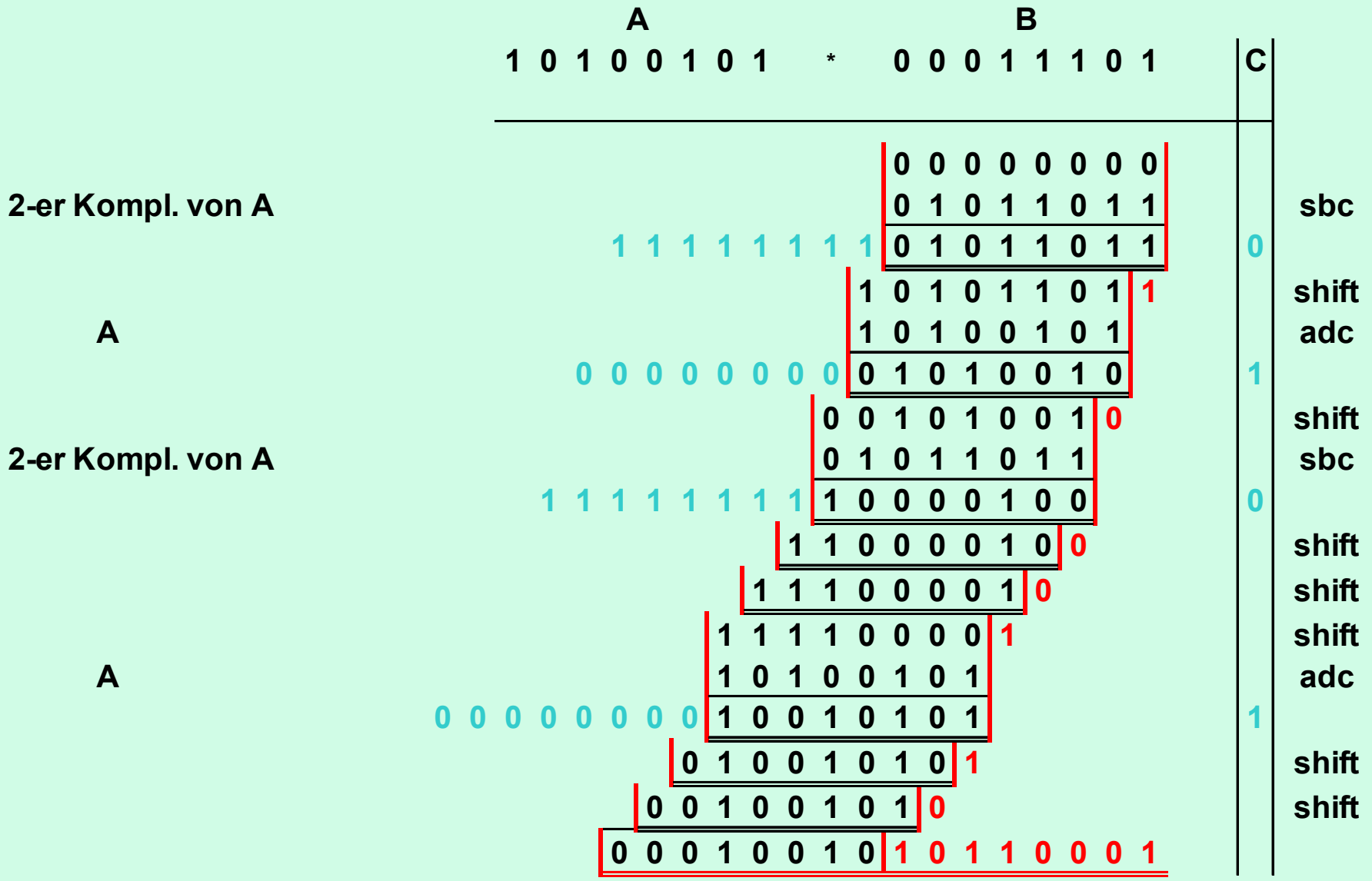
Y	Befehl	Vorzeichen	Akk	C	B → R0	
	laden		0 0 0 0 0 0 0 0	0	1 0 0 1 1 1 0 1	
	lsr R0			1	0 1 0 0 1 1 1 0	1
8	2-er (A)		0 1 0 1 1 0 1 1			
	SBC A		0 1 0 1 1 0 1 1	0		
	dec Vor	1 1 1 1 1 1 1 1				
	lsr Vor	0 1 1 1 1 1 1 1		1		
	ror		1 0 1 0 1 1 0 1	1		
	ror R0			0	1 0 1 0 0 1 1 1	0
	dey					
7	A		1 0 1 0 0 1 0 1			
	adc A		0 1 0 1 0 0 1 0	1		
	lsr		0 0 1 0 1 0 0 1	0		
	ror R0			1	0 1 0 1 0 0 1 1	1
	dey					
6	2-er (A)		0 1 0 1 1 0 1 1			
	SBC A		1 0 0 0 0 1 0 0	0		
	dec Vor	1 1 1 1 1 1 1 1				
	lsr Vor	0 1 1 1 1 1 1 1		1		
	ror		1 1 0 0 0 0 1 0	0		
	ror R0			1	0 0 1 0 1 0 0 1	1
	dey					
5	lsr Vor	0 0 1 1 1 1 1 1		1		
	ror		1 1 1 0 0 0 0 1	0		
	ror R0			1	0 0 0 1 0 1 0 0	1
	dey					

Fortsetzung nächstes Blatt



Die inerer Schleife 2 wird für jede "1" von B durchlaufen, bei einer "0" von B wird sie verlassen, dann ist eine Addition nötig.
Bei der ersten "1" einer Gruppe wird zuerst die Subtration durchgeführt

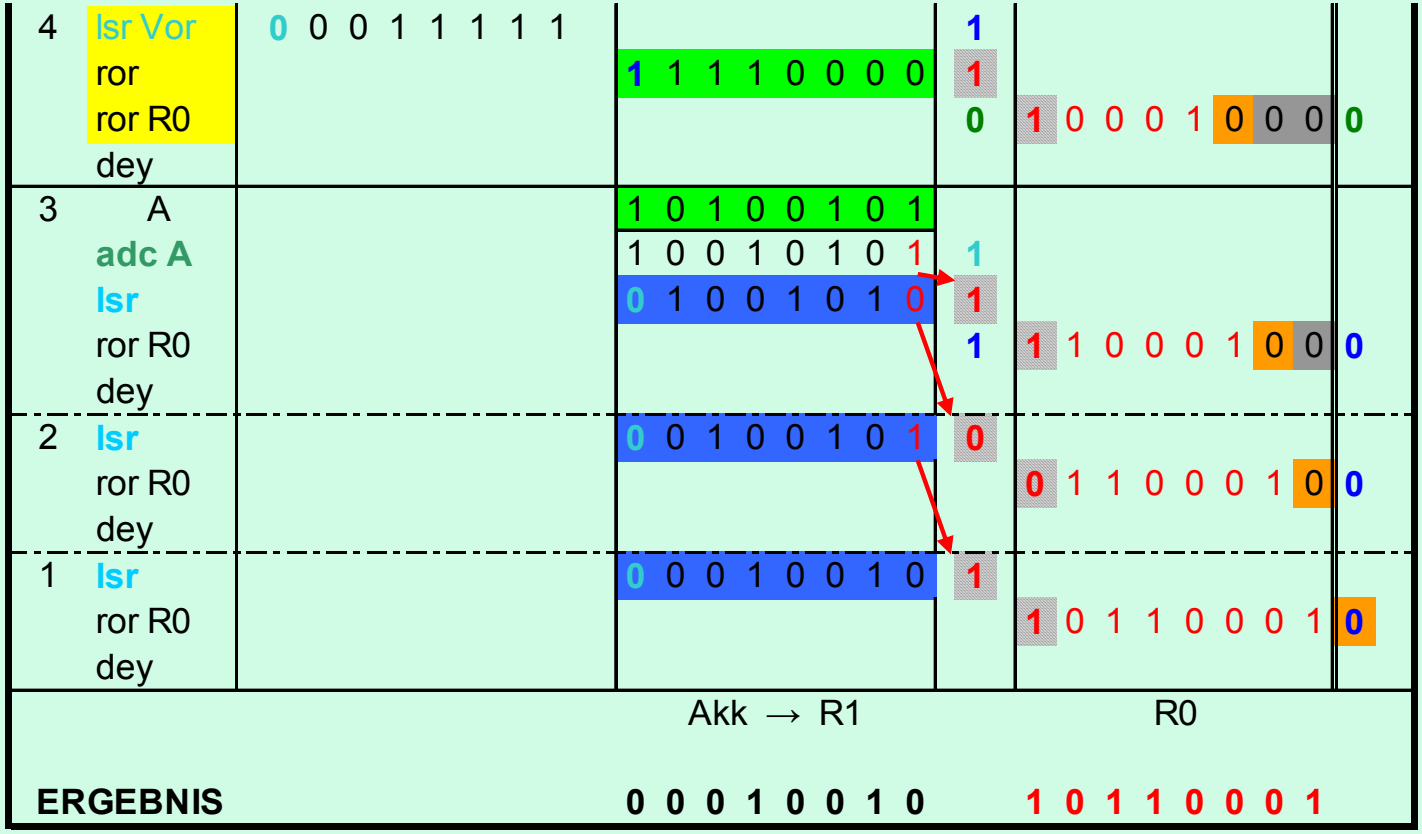
2. Beispiel zur Methode von Booth



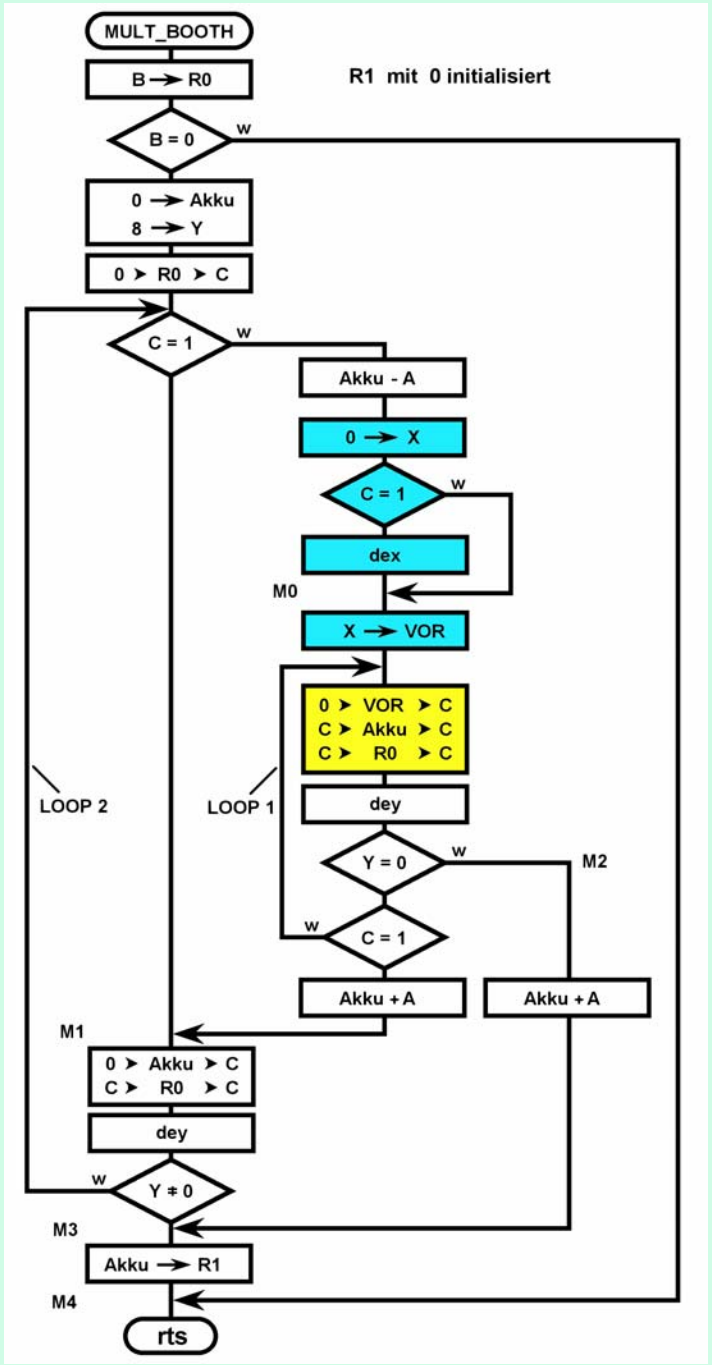
Programmumsetzung für 2. Beispiel

Y	Befehl	Vorzeichen	Akk	C	B → R0	
	laden		0 0 0 0 0 0 0 0	0	0 0 0 1 1 1 0 1	
	lsr R0			1	0 0 0 0 1 1 1 0	1
8	2-er (A)		0 1 0 1 1 0 1 1			
	SBC A		0 1 0 1 1 0 1 1	0		
	dec Vor	1 1 1 1 1 1 1 1				
	lsr Vor	0 1 1 1 1 1 1 1		1		
	ror		1 0 1 0 1 1 0 1	1		
	ror R0			0	1 0 0 0 0 1 1 1	0
	dey					
7	A		1 0 1 0 0 1 0 1			
	adc A		0 1 0 1 0 0 1 0	1		
	lsr		0 0 1 0 1 0 0 1	0		
	ror R0			1	0 1 0 0 0 0 1 1	1
	dey					
6	2-er (A)		0 1 0 1 1 0 1 1			
	SBC A		1 0 0 0 0 1 0 0	0		
	dec Vor	1 1 1 1 1 1 1 1				
	lsr Vor	0 1 1 1 1 1 1 1		1		
	ror		1 1 0 0 0 0 1 0	0		
	ror R0			1	0 0 1 0 0 0 0 1	1
	dey					
5	lsr Vor	0 0 1 1 1 1 1 1		1		
	ror		1 1 1 0 0 0 0 1	0		
	ror R0			1	0 0 0 1 0 0 0 0	1
	dey					

Fortsetzung nächstes Blatt



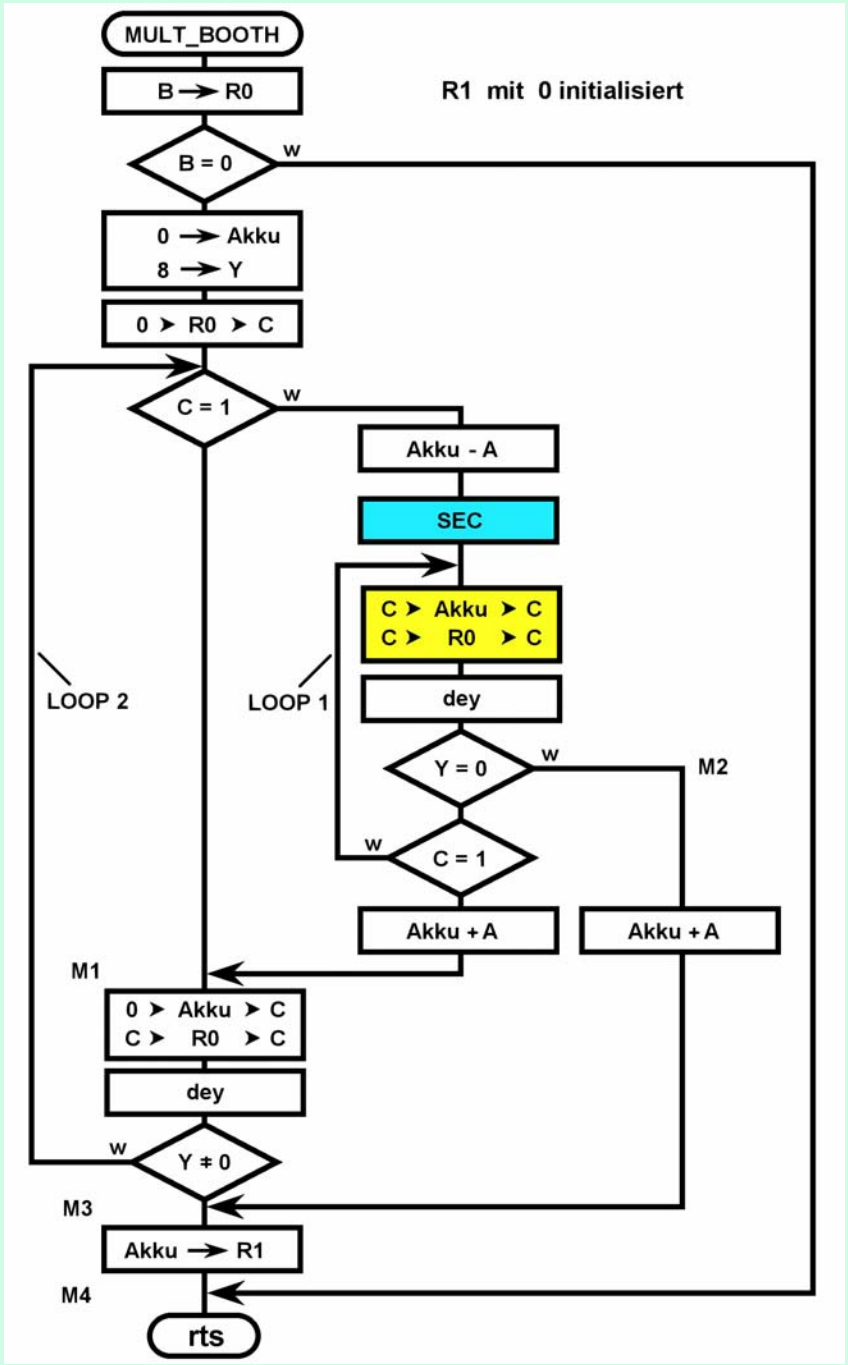
Die inerer Schleife 2 wird für jede "1" von B durchlaufen, bei einer "0" von B wird sie verlassen, dann ist eine Addition nötig.
Bei der ersten "1" einer Gruppe wird zuerst die Subtration durchgeführt




```
            LDA B
            STA R0
            BEQ  M4
            LDA #0
            LDY #8
            LSR R0
LOOP1:      BCC M1
            SBC A       ;  C ist bereits 1
            LDX #0
            BCS M0
            DEX
M0:         STX Vor     ; Vorzeichenbyte setzen
   LOOP2:     LSR VOR
              ROR
              ROR R0
              DEY
              BEQ  M2
              BCS  LOOP2
            ADC A
M1:         LSR
            ROR R0
            DEY
            BNE LOOP1
           BEQ M3
M2:        ADC A
M3:        STA R1
M4:        RTS
```

- **Die Einführung eines Vorzeichen-Bytes ist nicht zwingend nötig. Da, wie oben ausgeführt, das Vorzeichen nach Subtraktion und Addition immer eindeutig ist, reicht es die türkisen Befehle durch SEC zu ersetzen.**

Dies ist in der folgenden Programmvariante realisiert.




```
            LDA B
            STA R0
            BEQ  M4
            LDA #0
            LDY #8
            LSR R0
LOOP1:      BCC M1
            SBC A       ;  C war bereits 1
            SEC         ;  C nach Subtraktion 0
   LOOP2:   ROR
            ROR R0
            DEY
            BEQ  M2
            BCS  LOOP2
            ADC A
M1:         LSR
            ROR R0
            DEY
            BNE LOOP1
          BEQ M3
M2:       ADC A
M3:       STA R1
M4:       RTS
```