# Der 6502-Assembler

Im folgenden werden die Konzepte des 6502-Assemblers für das ultraFORTH83 dargestellt. Es wird kein vollständiges Glossar angegeben, da die Mnemonics des Assemblers allen Programmierern vertraut sein dürften. Eine genaue Darstellung der Funktionsweise findet Sie in den FORTH DIMENSIONS, Vol III,5 p. 143ff . Im folgenden wird eine kurze Zusammenfassung angegeben sowie Änderungen gegenüber dem Original dargestellt.

Die Funktionsweise des Adressinterpreters sowie der Routine NEXT wird in Kapitel 2 dargestellt.
Der 6502-Assembler gestattet strukturierte Programmierung. Die Strukturelemente sind analog zu den Kontrollstrukturen des Forth aufgebaut, tragen jedoch andere Namen, um die Verwechselungsgefahr zu verringern und die Übersichtlichkeit zu erhöhen.
Ein Beispiel:

```
cc  ?[ <ausdruck1> ][ <ausdruck2> ]?
```

cc steht für "condition code" . <ausdruck1> wird ausgeführt, wenn cc zutrifft, andernfalls <ausdruck2> . Der Teil :

```
][ <ausdruck2>
```

kann auch weggelassen werden. Das Analogon in Forth ist IF ... ELSE ... THEN .
Beachten Sie bitte, daß vor ?[ immer (!) ein condition code stehen muß. Außerdem findet keine Prüfung auf korrekte Verschachtelung der Kontrollstrukturen statt.
Weitere Kontrollstrukturen sind:

```
[[ <ausdruck1>  cc  ?[[  <ausdruck2>  ]]?
[[ <ausdruck1>  cc  ?]
[[ <ausdruck1> ]]
```

Die analogen Ausdrücke in Forth wären :

```
BEGIN  <ausdruck1>  WHILE  <ausdruck2>  REPEAT
BEGIN  <ausdruck1>  UNTIL
BEGIN  <ausdruck1>  REPEAT
```

Auch hier darf bei den Assemblerworten cc nicht weggelassen werden. Außerdem ist nur genau ein ?[[ zwischen [[ und ]]? zulässig. Beachten Sie bitte auch den Unterschied zwischen ]] und ]]? !

Als condition code sind zulässig :

```
0=  0<>  0<  0>=  CS  CC  VS  VC
```

Sie können den Prozessor-Flags Z N C und V zugeordnet werden. Im ersten Beispiel wird also <ausdruck1> ausgeführt, wenn cc durch 0= ersetzt wird und das Z-Flag gesetzt ist. Jeden der condition codes kann man durch ein folgendes NOT erweitern, also z.B.:

```
0= NOT ?[  0 # lda  ]?
```

Neben allen anderen Opcodes mit ihren Adressierungsarten gibt es auch die Sprünge BCC BCS usw. Sie sind nur in der Adressierungsart Absolut zulässig, d.h. auf dem Stack befindet sich die Adresse des Sprungzieles. Liegt diese Adresse außerhalb des möglichen Bereiches, so wird die Fehlermeldung "out of range" ausgegeben.

Für die anderen Opcodes sind, je nach Befehl, die folgenden Adressierungsarten zulässig:

```
.A  #  ,X  ,Y  X)  )Y  )
```

Die Adressierungsart Absolut wird verwendet, wenn keine andere angegeben wurde. Wird mit einem Mnemonic eine nicht erlaubte Adressierungsart verwendet, so wird die Fehlermeldung "invalid" ausgegeben.

Beispiele für die Verwendung des 6502-Assemblers (zur Erläuterung wird die herkömmliche Notation hinzugefügt) :

```
.a rol              rol a
1 # ldy             ldy #1
data ,X sta         sta data,x
$6 x) adc           adc ($6,x)
vector )y lda       lda (vector),y
vector ) jmp         jmp (vector)
```

Zusätzlich enthält das System noch mehrere Macros, die alle nur Absolut adressieren können:

winc - Inkrementiert einen 16-Bit-Zeiger um 1. wdec dekrementiert analog.

2inc - Inkrementiert einen 16-Bit-Zeiger um 2. 2dec dekrementiert analog.

;c: - Schaltet den Assembler ab und den Forth-Compiler an. Damit ist es möglich, von Maschinencode in Forth überzuwechseln. Ein Gegenstück ist nicht vorhanden.

Ein Beispiel für die Verwendung von ;C: ist:

```
... 0< ?[  ;c: ." Fehler" ;  Assembler  ]?  ...
```

Ist irgendwas kleiner als Null, so wird "Fehler" ausgedruckt und die Ausführung des Wortes abgebrochen, sonst geht es weiter im Code.

Schließlich gibt es noch die Worte >LABEL und LABEL . >LABEL erzeugt ein Label im Heap, wobei es den Wert des Labels vom Stack nimmt. LABEL erzeugt ein Label mit dem Wert von HERE. Beispiel:

```
Label schleife    dex
                  schleife bne
```

Ein Codewort muß letztendlich immer auf NEXT JMP führen, damit der Adressinterpreter weiter arbeitet. Im folgenden Glossar werden Konstanten angegeben, auf die gesprungen werden kann und die Werte auf den Stack bringen bzw. von ihm entfernen. Wichtig ist insbesondere die Routine SETUP. Sie kopiert die Anzahl von Werten, die im Akkumulator angegeben wird, in den Speicherbereich ab N.

Für den Zugriff auf den Stack wird, soweit das möglich ist, die Benutzung der Worte SETUP und PUSH ... empfohlen. Das reicht allerdings häufig nicht aus. In diesem Fall kann man die Werte auf dem Stack folgendermaßen zugreifen:

```
SP x) lda     \ Das untere Byte des ersten Wertes
SP )y lda     \ Das obere Byte des ersten Wertes
```

sowie durch Setzen des Y-Registers auch die zweiten, dritten etc. Werte. Beachten Sie bitte, das in NEXT verlangt wird, daß das X-Register den Inhalt $00 und das Y-Register den Inhalt $01 hat. Das wurde im obigen Beispiel ausgenutzt. Beispiele für Assemblercode in Forth, den wir als gut empfinden, sind unter Anderem die Worte FILL und -TRAILING . Wollen Sie Assembler programmieren, so sollten Sie sich diese Worte und noch einige andere ansehen.

**Beim Assemblerprogrammieren muß beachtet werden, daß ultraFORTH das ROM abschaltet. Daher müssen Lesezugriffe ins ROM etwas anders organisiert werden.**
**Beispiel für den C16:**

```
ffd2 jsr                        springt eine RAM-Routine an.
ff3e sta  ffd2 jsr  ff3f sta    springt eine ROM-Routine an.
```

**Sie funktioniert nur, wenn sie im unteren RAM-Bereich (<$8000) steht. Sonst folgen undefinierte Reaktionen.**
**Beim C64 ist eine Bankumschaltung nur für Lesezugriffe in das BASIC-ROM erforderlich. Hierbei ist zusätzlich zu beachten, daß eine Bankumschaltung mit SEI vorbereitet werden muß, da andernfalls der periodische Tastaturinterrupt zu einem Absturz führen würde.**
**Auf dem C16 ist kein SEI erforderlich, da im RAM der Vektor $FFFE auf eine eigene Interruptroutine zeigt (sie benötigt ca. 1 Promille der Rechenzeit). Aus dem gleichen Grund führt eine BRK - Instruktion zwar weiterhin in den Monitor, allerdings mit falschem Registerdump, da der Monitor auf dem Stack die Daten der Interruptroutine statt der Register vorfindet.**

# Assembler

PushA -- addr
Eine Konstante, die die Adresse einer Maschinencode-Sequenz enthält, die den Inhalt des Akku vorzeichenbehaftet auf den Datenstack legt und dann zu NEXT springt. Wird als letzter Sprungbefehl in Code-Worten benutzt.

PushØA -- addr
Eine Konstante, die die Adresse einer Maschinencode-Sequenz enthält, die den Inhalt des Akku auf das Low-Byte des Datenstacks ablegt. Das High-Byte wird grundsätzlich auf Ø gesetzt. Anschliessend wird zu NEXT gesprungen. Wird als letzter Sprungbefehl in Code-Worten benutzt.

Push -- addr
Eine Konstante, die die Adresse einer Maschinencode-Sequenz enthält, die den Inhalt des Akku als High-Byte auf den Datenstack legt. Das Low-Byte wird vom prozessorstack geholt und muß vorher dort abgelegt worden sein. Anschliessend wird zu NEXT gesprungen. Wird als letzter Sprungbefehl in Code-Worten benutzt.

Next -- addr
Eine Konstante, die die Adresse von NEXT auf den Datenstack legt. Wird als letzter Befehl in Code-Worten benutzt.

xyNext -- addr
Wie NEXT jedoch werden vorher das X-Register mit Ø und das Y-Register mit 1 geladen. Das System erwartet grundsätzlich bei Aufruf von NEXT diese Werte in den Registern. Ansonsten reagiert es mit Absturz.

Puta -- addr
Eine Konstante, die die Adresse einer Maschinencode-Sequenz enthält, die den Akku als Low-Byte auf den Datenstack ablegt. Das High-Byte wird nicht verändert, ebenso wird im Gegensatz zu PUSH kein Platz auf dem Datenstack geschaffen. Anschliessend wird NEXT durchlaufen. Wird als letzter Befehl in Code-Worten benutzt.

Pop -- addr
Eine Konstante, die die Adresse einer Maschinencode-Sequenz enthält, die das oberste Element vom Datenstack entfernt. Anschliessend wird zu NEXT gesprungen. Wird als letzter Sprungbefehl in Code-Worten benutzt.

Poptwo -- addr
Eine Konstante, die die Adresse einer Maschinencode-Sequenz enthält, die die obersten beiden Elemente vom Datenstack entfernt. Anschliessend wird zu NEXT gesprungen. Wird als letzter Sprungbefehl in Code-Worten benutzt.

RP -- addr
Eine Konstante, die die Adresse des Returnstackpointers enthält.

UP -- addr
Eine Konstante, die die Adresse des Userpointers, also des Offsets zu ORIGIN enthält.

SP -- addr
Eine Konstante, die die Adresse des Datenstackpointers enthält.

IP -- addr
Eine Konstante, die die Adresse des Instruktionspointers der Forth-Maschine enthält. Dieser zeigt auf das jeweils nächste abzuarbeitende Wort.

W -- addr
Eine Konstante, die die Adresse des Wort-Pointers der Forth-Maschine enthält. Dieser zeigt auf das jeweils gerade bearbeitete Wort.

N -- addr
Eine Konstante, die die Adresse eines Speicherbereichs in der Zeropage enthält, der dem Anwender zur Verfügung steht.

setup -- addr
Eine Konstante, die die Adresse einer Maschinencode-Sequenz enthält, die n Elemente vom Datenstack abbaut und bei N ablegt. Die Anzahl n muß im Akku stehen, wenn SETUP als Subroutine angesprungen wird. Das oberste Element des Datenstacks liegt Bei N , das zweite bei N+2 etc. Zum Schluß werden X- und Y-Register auf Ø bzw. 1 gesetzt.

wcmp ( addr1 addr2 -- )
Dieses Assemblermakro assembliert eine Sequenz, die bei Ausführung den Inhalt des Wortes an addr1 mit dem Inhalt des Wortes an addr2 vergleicht. Anschließend ist das Carry-Flag high, wenn der Inhalt von addr1 größer oder gleich dem Inhalt von addr2 ist.
Es werden der Akku sowie die Statusregisterflags C Z O N verändert.

ram ( -- )
Makro. Schaltet bei Ausführung auf eine andere Speicherbank. Die genaue Wirkungsweise ist maschinenabhängig.

rom ( -- )
Makro. Schaltet bei Ausführung auf eine andere Speicherbank. Die genaue Wirkungsweise ist maschinenabhängig.

sys ( addr-- )
Makro. Schaltet bei Ausführung auf eine Speicherbank mit Systemroutinen und führt jsr aus. Die genaue Wirkungsweise ist maschinenabhängig.