3

3. Wie man programmiert

Nur Befehle hintereinanderschreiben, reicht meist nicht aus, um zu programmieren. Oft müssen Fälle unterschieden und unterschiedliche Rechenwege durchlaufen werden. Dafür gibt es in höheren Programmiersprachen Anweisungen wie „if", „ for", „while", „do", „until", „call" und „return". In Assembler gibt es so etwas nicht. Schleifen und Sprünge werden *zu Fuß* abgecodet. Wie das geht, zeigt dieses Kapitel.

3.1 Die Ja-Nein-Entscheidung („if-Anweisung")

Es wird nur dann *b* durch *a* dividiert, wenn *a* nicht 0 ist ...

```
          tsteq     a         m1                ;  Wenn a=0 Sprung nach m1
          div       b         a                 ;# Nur Dividieren, wenn a≠0
m1:
          ...
```

Der mit ;# gekennzeichnete Code wird nur bedingt durchlaufen. Man beachte, dass die Bedingung, bei der der Code durchlaufen werden soll und die Testbedingung im Testbefehl entgegengesetzt formuliert sind.

3.2 Die Ja-Nein-Entscheidung mit Alternative („if-else-Anweisung")

Es wird nur dann b mit 10 multipliziert, wenn a nicht 0 ist. Andernfalls wird b durch 5 dividiert. Um zu verhindern, dass nach der Multiplikation in die Division „hineingelaufen wird", wird ein jump-Befehl eingesetzt ...

```
          tstne     a         m1                ;  Wenn a≠0 Sprung nach m1
          mul       b         10                ;# Wenn a=0 dann b mal 10
          jump      m3:                         ;# Wegspringen zu m3
m1:
          div       b         5                 ;% Sonst b durch 5 dividieren
m3:                                             ;
          ...                                   ;
```

Der mit ;# gekennzeichnete Code wird bei Nichterfüllung der Testbedingung durchlaufen, der mit ;% gekennzeichnete Code bei Erfüllung der Testbedingung.

3.3 Die Auswahlentscheidung („switch-case-Anweisung“)

Sind mehrere Alternativen möglich, werden im Prinzip Ja-Nein-Entscheidungen hintereinandergeschaltet. Da es hier meist nicht auf 0 zu testen ist, kommt hier eher der Comparebefehl in Frage.

Das Folgende Beispiel führt je nachdem eine „Codevariable“ *c* den Wert 0, 1, 2 oder 3 hat, eine Addition, Subtraktion, Multiplikation oder Division von *a* und *b* aus:

```
        cmpne   c       0       m1      ;  Wenn c≠0 bei m1 weiter
        add     a       b               ;0 Wenn c=0 Addieren
        jump    m9                      ;0 Wegspringen zu m9
m1:
        cmpne   c       1       m2      ;  Wenn c≠1 bei m2 weiter
        sub     a       b               ;1 Wenn c=1 Subtrahieren
        jump    m9                      ;1 Wegspringen zu m9
m2:
        cmpne   c       2       m3      ;  Wenn c≠2 bei m3 weiter
        mul     a       b               ;2 Wenn c=2 Multiplizieren
        jump    m9                      ;2 Wegspringen zu m9
m3:
        cmpne   c       3       m9      ;  Wenn c≠3 bei m9 weiter
        div     a       b               ;3 Wenn c=3 Dividieren
        jump    m9                      ;3 Kann man auch weglassen
m9:
        ...                             ;
```

Je nachdem Wert von *c* wird der mit ;0, ;1, ;2 oder ;3 gekennzeichnete Code durchlaufen. Den letzen jump-Befehl kann man auch weglassen, denn er „läuft“ in Marke m9 automatisch „hinein“.

3.4 Die Auswahlentscheidung mit berechnetem Sprung („on-goto-Anweisung“)

Dies ist eine *old style* Assembler-Raffinesse, die es in höheren Programmiersprachen kaum gibt. Sie ist blitzschnell und kann angewendet werden, wenn zwischen ganzzahligen Alternativen entschieden werden soll, die einen geschlossenen Zahlenbereich abdecken.

Dies ist im gerade vorgestellten Beispiel der Fall. Die „Codevariable“ *c* kann Werte von 0 bis 3 haben. Bei *c*=0 soll addiert werden, bei *c*=1 soll er subtrahieren, bei *c*=2 multiplizieren, bei *c*=4 dividieren. Gut aufpassen!

Wir notieren:

```
          mov       d         c         ;  d ist jetzt 0, 1, 2 oder 3
          mul       d         2         ;  d ist jetzt 0, 2, 4 oder 6
          add       d         m0        ;  d ist jetzt m0, m1, m2 oder m3!!
          jump      d                   ;  nun Sprung nach m0, m1, m2, m3
m0:
          add       a         b         ;0
          jump      m9                  ;0
m1:
          sub       a         b         ;1
          jump      m9                  ;1
m2:
          mul       a         b         ;2
          jump      m9                  ;2
m3:
          div       a         b         ;3
          jump      m9                  ;3
m9:
```

Zunächst wird eine „Destinationsvariable“ *d* eingeführt. Diese wird als Sprungadresse genutzt werden. — Nun ist bekannt, dass die Marken m0, m1, m2 und m3 je 2 Befehle auseinanderliegen: Es liegt nämlich zwischen ihnen (1.) der jeweilige add/sub/mul/div und (2.) der jeweilige „Wegsprung-Jump“. Der Code funktioniert nun wie folgt:

1. „`mov d c`“ kopiert zunächst *c* nach *d*.

2. „`mul d 2`“ rechnet den Markenabstand 2 ein. Wichtig ist, dass alle einzelnen Codestücke der Auswahl genau *die gleiche Anzahl Befehle* enthalten. Dies können auch 3, 4 oder 10 Befehle sein. In diesen Fällen wäre d mit 3, 4 oder 10 zu multiplizieren. So wird *d* zu einer *„relativen Codeadresse“.*

3. „`add d m0`“ addiert nun die „Basisaddresse“ oder „Basismarke“ *m0* hinzu. Damit wird *d* die entgültige *„absolute Codeadresse“.*

4. Nun kommt es: Mit „`jump d`“ wird exakt die Zielmarke m0, m1, m2 bzw. m3 erreicht. In Abhängigkeit von c wird der mit ;0, ;1, ;2 oder ;3 gekennzeichnete Code durchlaufen.

Die Marken m1, m2, m3 kann man im Programmtext sogar weglassen! Sie dienen nur der Verbesserung der Programmlesbarkeit.

Berechnete Sprünge sind viel schneller, als Auswahlentscheidungen vom „switch-case“-Typ. Wenn z. B. 1000 Alternativen zu trennen sind, rechnet die „switch-case-Auswahl“ möglicherweise bis zu 1000 Vergleichsbefehle! Der berechnete Sprung kommt mit vielleicht 5 Befehlen aus.

Der berechnete Sprung ist eine ganz typische Assemblercodierung – ein bewahrenswertes Kulturgut, das im Zeitalter der strukturierten Programmierung leider auf das Abstellgleis geraten ist.

3.5 Die Solange-Bedingungsschleife („do … while-Anweisung“)

Oft muss eine Schleife **solange** durchlaufen werden, wie eine Bedingung erfüllt ist:

```
          mov       a         100000              ;  Variable laden
m1:
          ...                                     ;# Beliebiger Schleifencode
          ...                                     ;# Beliebiger Schleifencode
          div       a         2                   ;# a herabsetzen
          cmpgt     a         1         m1        ;# Wenn a≥1 neuer Durchlauf
          ...                                     ;
```

Der mit ;# gekennzeichnete Code wird **solange** durchlaufen, wie *a* größer als Eins ist.

Dadurch, dass *a* bei jedem Schleifendurchgang halbiert wird, wird irgendeinmal die Bedingung $a \geq 1$ nicht mehr erfüllt sein, was zum Verlassen der Schleife führt. Dies ist wichtig, denn ansonsten hätte man sich eine Endlosschleife programmiert. Das Programm müsste dann von außen abgebrochen werden.

Ist die Bedingung nicht erfüllt, so wird die Schleife zumindest einmal durchlaufen. Dies liegt daran, dass sich der Test am Schleifenende befindet.

3.6 Die Bis-Bedingungsschleife („do … until-Anweisung“)

Eine andere Variante hat folgenden Code:

```
          mov       a         100000              ;
m1:
          div       a         2                   ;# a herabsetzen
          cmplt     a         1         m3        ;# Wenn a≤1 Sprung nach m1
          ...                                     ;# Beliebiger Schleifencode
          ...                                     ;# Beliebiger Schleifencode
          jump      m1:                           ;# Neuer Schleifendurchlauf
m3:                                               ;
          ...                                     ;
```

Hier wird, wenn die Bedingung erfüllt ist, aus der Schleife herausgesprungen. Ein jump sorgt für die Zyklenbildung. Die Schleife wird also durchlaufen, **bis** die Bedingung erfüllt ist.

Hier ist es wichtig, dass die Bedingung $a \leq 1$ irgendeinmal erfüllt wird, denn sonst springt er nie aus der Schleife heraus. Wir hätten wieder eine Endlosschleife.

Ist die Bedingung gleich beim ersten Durchlauf erfüllt, so wird die Schleife überhaupt nicht durchlaufen. Dies liegt daran, dass der Test zu Schleifenanfang erfolgt.

Mit geeigneten Sprunganordnungen lassen sich auch do … while-Anweisungen mit Test am Schleifenanfang und do … until-Anweisungen mit Test am Schleifenende programmieren.

3.7 Die Zählschleife („for-Anweisung“)

Sehr häufig ist der Fall, dass eine Zahl angibt, wie oft eine Schleife durchlaufen werden soll.

```
          mov       c         10                  ;  Zähler i mit 10 füllen
m1:
          mul       a         b                   ;# Wenn a≤0 Sprung nach m1
          dec       c                             ;# c weiterstellen
          tstgt     c         m1                  ;# Sprung nach m1, wenn c>0
          ...
```

Der mit ;# gekennzeichnete Code wird solange durchlaufen, solange die Zählvariable c größer als Null ist, hier also zehnmal. Das Weiterstellen von Zählvariablen erfolgt in typischem Assemblerstil vorzugsweise mit den Befehlen inc und dec. Addition oder Subtraktion gelten als *unstylish*.

3.8 Der Unterprogrammsprung und -rücksprung („call-return-Anweisung“)

Unterprogramme sind sehr effektiv, wenn ein kleiner Codeabschnitt (z. B. mit einer bestimmten Formel) häufig und mit wechselnden Parametern an unterschiedlichen Stellen in einem längeren Programm abgearbeitet werden soll. Das längere Programm ist dann das Hauptprogramm. Die Formel wird in einem Unterprogramm abgecodet und dieses Unterprogramm wird jedes Mal aufgerufen, wenn die Formel berechnet werden soll.

RTA verfügt über keine speziellen Unterprogrammbefehle. Für den Sprung in das Unterprogramm und zurück ist aber der jump-Befehl völlig ausreichend. Man programmiere wie folgt:

```
          _var      return                        ; Rücksprungadresse
          ...

; HAUPTPROGRAMM: Rahmen

          input     r0        1.~Zahl
          input     r1        2.~Zahl
          mov       return    m3                  ; Rücksprungadresse speichern
          jump      .subgeo                       ; Sprung zum Unterprogramm
m3:                                               ; Wiedereinsprungmarke
          output    r2        Mittelwert
          ...
```

Unterprogramm nächste Seite …

```
; UNTERPROGRAMM .subgeo: Liefert in r2 geometrisches Mittel aus r0 und r1

.subgeo:                                          ; Anfang Unterprogramm
          mov       r2        r0
          mul       r2        r1
          root      r2        2
          jump      return                        ; Sprung zurück in Hauptpr.
```

Wesentlich ist, dass das Unterprogramm nicht auf eine feste Rücksprungadresse zurückspringen darf. Der Rücksprung wird organisiert, indem man hinter dem Absprung aus dem Hauptprogramm eine Marke („m3“) anordnet. Diese hinterlegt das Hauptprogramm vor dem Unterprogrammaufruf auf einer globalen Variablen „return“. Dorthin wird dann vom Unterprogramm zurückgesprungen.

Es haben sich verschiedene Stile eingebürgert, Operanden („Parameter“) aus dem Hauptprogramm an das Unterprogramm zu übergeben.

Register: Eine assemblertypische Parameterübergabe ist die Nutzung von Registern. Hierfür eignen sich die vordefinierten Variablen r0 bis r7. Das Hauptprogramm legt die Parameter in den Registern ab, das Unterprogramm bearbeitet die Werte in den Registern und nach dem Unterprogrammende stehen dort die Ergebnisse.

Globale Symbole: Eine andere Möglichkeit besteht darin, Operanden, auf bestimmten, sowohl dem Haupt- als auch dem Unterprogramm bekannten Symbolnamen zu übergeben. Weil RTA mit der Symboltabelle nur einen einzigen *„globalen Namensraum“* kennt, kann das Unterprogramm auf alle Symbole des Hauptprogramms zugreifen. Hier ist darauf zu achten, dass Symbolnamendoppelungen vermieden werden. Wenn das Hauptprogramm eine Variable „a“ benutzt, darf das Unterprogramm „a“ nicht unqualifiziert verändern. Es wird empfohlen, alle lokale Variablen im Unterprogramm mit einem Punkt beginnen zu lassen, eine lokale Variable „a“ also besser „.a“ zu nennen. Über deren Namen kann das Hauptprogramm auch Parameter übergeben.