

Das RTA-Handbuch

Der Rheingold-Express

Was für die Eisenbahn die Dampfmaschine war, war für den Computer der Assembler: Die Urkraft, mit der einst die gesamte Technologie zum Laufen gebracht wurde.

Damals konnte man noch sehen, fühlen und *riechen*, wie ein Computer funktionierte. Es gab da noch keine von diesen glitzernden Entwicklungsumgebungen. Um einen Fehler im Programm zu finden, wurde einfach ein HALT-Befehl in den Code hineingesetzt. Wenn er dann dort drauf lief, gab das einen Trap und dann stand der Vierzigtonner. Nun konnte man in aller Seelenruhe den Hauptspeicher aufmachen und nachsehen. Befehl für Befehl, Maschinenwort für Maschinenwort. Wow! In älteren James-Bond-Filmen kann man die Typen noch an den MI-V-Computern im Hintergrund Magnetbänder auflegen sehen.

Wie die damals programmiert haben? In Assembler. Assembler gilt heute als kryptisch gefährlich, ungeheuer schwer zu debuggen, äußerst fehleranfällig, überhaupt nicht portabel und ungeheuer komplex. *Vergiss es*: In seinem Herzen ist Assembler vor allem eines: Eine wunderbar einfache, klare, geniale und mächtige Art – zu denken. In einer Sprache, die Rechenmaschinen verstehen. Und Menschen.

Wie das geht? Etwa so: Den Mittelwert m zweier Zahlen z_1 und z_2 berechnet man mit $m = (z_1+z_2) / 2$. Das sieht in Assembler dann etwa wie folgt aus:

```
input    z1      1. ~Zahl
input    z2      2. ~Zahl
mov      m       z1
add      m       z2
div      m       2
output   m       Mittelwert
```

Wollen wir weitermachen?

Assembler heißt aller allerdings auch gewöhnlich, dass man eine Hardware braucht. Beim Programmieren zeitgenössischer Architekturen hat man es schnell einmal mit 700 Millionen Schaltelementen zu tun. So etwas ist natürlich nicht einfach.

Seit Don E Knuths MIX Assembler Language MIXAL wissen wir aber auch: Es muss gar kein verdrahteter Prozessor sein. Eine hypothetische Maschine kann auch ganz gut rechnen.

Wir machen nun die Maschine nicht möglichst komplex, sondern reduzieren sie auf das Wesentliche. Somit wird auch der Assembler einfach werden und eine einfache Übersetzung gestatten. Das wird der Kerngedanke von RTA sein, dem Reduced Transliteration Assembler.

1. Der Quelltext

1.1 Das Programm

```
;
; Das Programm
;
$m1:
    mov     r0     a           ; Winkel
    add     r0     10         ; plus 10
    mul     r0     pi         ; mal Pi
    div     r0     180        ; durch 180
    cmpgt   r0     i         $m5 ; Wenn > i Sprung nach $m5
    ...
```

Ein RT-Assemblerprogramm besteht – wie alle Programme – aus einzelnen Zeilen. Die Zeilen bestehen aus Token. Token bestehen aus aufeinanderfolgenden Druckzeichen. Token werden durch ein oder mehrere Leerzeichen oder Tabulatoren voneinander getrennt. Token sind also z. B. `$m1:` `mov` `r0` `a` `pi/2` `180`.

In Assembler trennt man Token üblicherweise durch Tabulatoren. So wird der Quelltext in einzelne Spalten gegliedert. Jede Spalte ist 8 Zeichen breit.

1.2 Die Markenspalte

Die erste Spalte heißt Markenspalte.

```
$m1:      add     r0     10         ; plus 10
```

Markentoken tragen als letztes Zeichen immer einen Doppelpunkt. In RT dürfen in der Markenspalte keine Befehle stehen. (In anderen Assemblern kann dies zulässig sein.)

1.3 Die Befehlsspalte

Assembler programmieren heißt, Befehle aufschreiben. Jeder Befehl wird auf einer neuen Zeile notiert. Die Befehle haben einen bestimmten festen Befehlscode, z. B. `add`, `sub`, `mul`, `div`:

```
$m1:      add     r0     10         ; plus 10
```

Dieser `add`-Befehl weist die Addition von 10 auf eine Variable `r0` an.

1.4 Die Operandenspalten

Früher wurden Ein-, Zwei- und Dreiadressmaschinen unterschieden. Die virtuelle Maschine von RT ist eine Dreiadressmaschine. Dies bedeutet, dass jeder Befehlscode drei Operanden haben kann. Entsprechend gibt es drei Operandenspalten:

```
$m1:
    add    r0    10    .    ; plus 10
```

Die Operanden geben an, womit gerechnet wird. Hier wird z. B. zu einer Variablen mit dem Namen „r0“ die Zahl 10 hinzuaddiert. Der Punkt `.` steht hier für einen nicht benutzten Operanden und kann weggelassen werden.

1.5 Kommentar, Kommentarzeilen, Fortsetzungszeilen, Leerzeilen

Kommentar: Alles, was nach einem Semikolon `;` folgt, ist Kommentar. Kommentar liest die Maschine nicht mit. Kommentare beginnen vorzugsweise nach 5 Tabulatoren:

```
$m1:
    add    r0    10
           ; plus 10
```

Kommentarzeilen: Eine Programmzeile kann auch ausschließlich Kommentar enthalten. Dann steht das Semikolon ganz vorn:

```
;
; Programm
;
    mov    a    b
```

Explizite Zeilenverkettung: Drohen Zeilen zu lang zu werden, so kann man sie mit dem Zeilenverkettungszeichen `¶` („Paragraph“) auf mehrere Zeilen verteilen. Immer dann, wenn ein `¶` als letztes Zeichen einer Zeile unmittelbar vor dem Zeilenumbruch notiert wird, ist die nächste Zeile die Fortsetzung der vorangegangenen Zeile:

```
input    a    Meridianstreifenabstand_in_Bog¶
           enminuten
```

Das `¶`-Zeichen, der unmittelbar folgende Zeilenumbruch, sowie in der folgenden Zeile führende Leerzeichen und Tabulatoren werden ignoriert. `Meridianstreifenabstand_in_Bogenminuten` ist folglich ein Token.

Hinweis: Für das Zeichen `¶` gebe man bei gedrückter Alt-Taste „0182“ auf dem Ziffernblock ein, Kurzschreibweise hierfür auch `<Alt/0182>`.

Leerzeilen: Eine oder mehrere Programmzeile(n) können auch ganz leer sein:

```


```

2. Operanden

Die hinter den Befehlscodes stehenden Operanden geben an, *mit welchen Daten* gerechnet werden soll. Die Operanden in RTA heißen *Symbole*. Alle Symbole werden in einer Symboltabelle verzeichnet. In dieser Tabelle gibt es eine Spalte für Symbolnamen und eine Spalte für Symbolwerte:

Symboladresse	Symbolname	Symbolwert
1	a	300
2	R0	0
3	alpha	0
4	P23	-15

Der *Symbolname* ist das Token, welches im Programmtext steht. Der *Symbolwert* ist ein Speicherplatz, auf welchem eine Zahl gespeichert und mit dem gerechnet werden kann. Die Zeilennummer der Tabelle heißt *Symboladresse*.

RT kennt keine Datentypen. Es gibt somit keine Unterscheidung von Integerzahlen, Gleitkommazahlen, Zeichenketten etc. Um es vorwegzunehmen: RTA rechnet alles in Double float.

Ein Symbol kann auf 4 verschiedene Arten genutzt werden:

- als Variable,
- als Zahl,
- als Marke,
- als Zeichenkette.

2.1 Variablen

In den meisten Programmiersprachen müssen Variablennamen grundsätzlich aus Buchstaben bestehen, denen oft Ziffern und Unterstriche folgen können.

In RT ist jede Kombination von Druckzeichen zugelassen. Das können Buchstaben wie **A, B, C, a, b, c**, Umlaute wie **ä, ö, à, ø** oder Sonderzeichen, wie **:, -, +, (, :** sein – ja es sind sogar Exotica wie **°, °, ¢, ¢** oder **±** zulässig. Ziffern wie **0, 1, 2, 3** sind natürlich ebenfalls erlaubt. So sind z. B. Token wie

```
a      aa      r0      lE1e1    ..
Lücke  MAB      â      (        z1_in_m
M12    M(33)    $er    alpha    ' ' '
ad     m      z2     ¤¥      100
```

gültige Variablennamen. Man kann also auch Variablen

```
a'      b''      delta²   A(%)
```

nennen. RT unterscheidet Groß- und Kleinbuchstaben. So bezeichnen die drei Symbole

```
alpha   Alpha   ALPHA
```

drei verschiedene Symbole.

Sobald eine Variable erstmalig auftaucht, wird sie in die Symboltabelle eingetragen. Es gibt also keine Deklarationspflicht für Symbole.

2.2 Zahlen

Zahlen sind in RT ebenfalls ganz normale Variablen.

Wenn der RT-Assembler ein neues Symbol erstmals auffindet, so versucht er den Symbolnamen als *Zahlenwert* zu interpretieren. Das Ergebnis wird dann sofort als Symbolwert eingetragen. Auf diese Art erhält ein Symbol mit dem Namen `1` z. B. den Wert 1 und ein Symbol mit dem Namen `10` z. B. den Wert 10. So erhalten Zahlen-Symbolnamen entsprechende Zahlenwerte zugewiesen.

Zahlen werden so, wie in Programmiersprachen üblich, notiert. Das Dezimaltrennzeichen ist der Punkt `.`, nicht das Komma. Token `3.141592653` erzeugt Symbolwert 3,141592653. Man sollte nicht `3,141592653` schreiben, denn dann wird lediglich Symbolwert 3 erzeugt, da beim Komma abgebrochen wird.

Die Zeichen `-` und `+` können Vorzeichen sein. Das Exponentialkennzeichen `E` oder `e` hilft, große und kleine Zahlen kürzer zu notieren. `1E6` steht für den Wert 1 Million und `25e-6` für den Wert 0,000025.

2.3 Marken

Mitunter müssen Positionen im Programm markiert werden, z. B. um Sprungadressen festzulegen. Dies erfolgt mit einer Markendefinition.

Eine Markendefinition ist ein Token an einem Zeilenbeginn mit abschließendem Doppelpunkt. Die Markendefinition

```
$m001:
```

trägt ein Symbol „`$m001`“ in die Symboltabelle ein. Als Symbolwert wird die Zeilennummer im Programm eingetragen, in welcher sich die Markendefinition befindet.

Wie man sieht, gehört der Doppelpunkt selbst nicht zum Symbolnamen (siehe hierzu auch unten, Befehl `_lab`). Auf Markendefinitionszeilen dürfen keine (anderen) Befehle stehen.

2.4 Zeichenketten

Mitunter sind Dialogtexte erforderlich. Hierfür nutzt RT den *Symbolnamen* als Zeichenkette. Der Befehl

```
input      a      Anzahl
```

arbeitet z. B. nutzt z. B. das Token `Anzahl` als Zeichenkette *Anzahl*.

Weil Leerzeichen Token trennen, können diese in Zeichenketten nicht auftreten. Hier gilt nun die Tilde `~` als Leerzeichen-Stellvertreter.

Eine Sonderbedeutung hat auch der Backslash `\`. Dies ist der Zeilenumbruch-Stellvertreter. In dem Befehl `pause` steht z. B. die Notation

```
pause      Programm~fortsetzen?\ („Fortsetzen“-Taste~drücken)
```

für den Mitteilungstext

*Programm fortsetzen?
(„Fortsetzen“-Taste drücken)*

In RT gibt es also nicht die Zeichenketten-Form `"abc def ghi"`, oder `'abc def ghi'`, wie in anderen Programmiersprachen. Dies erscheint zunächst etwas gewöhnungsbedürftig, ist aber einfach und verständlich. Es gibt auch keine Spezialnotationen, wie `\\`, `\"`, `\'`, `\n`, `\t`, `\0112`, `\0xa3` etc, die oft wechseln und sich von Programmiersprache zu Programmiersprache fein unterscheiden.

Mit dem Zeilenverkettungszeichen `\¶` an kann auch schreiben:

```
pause      Programm~fortsetzen?\¶  
           („Fortsetzen“-Taste~drücken)
```

Das verbessert die Lesbarkeit. Die Zeichen `\` und `\¶` haben allerdings völlig unterschiedliche Bedeutungen:

- Das `\`-Zeichen ist der Zeilenumbruchs-Stellvertreter. Es ersetzt einen Zeilenumbruch *im* Symbolnamen. Es ist Tokenbestandteil und Zeichen *innerhalb* einer Zeichenkette.
- Das `\¶`-Zeichen ist das Zeilenverkettungszeichen. Es weist, wenn es das letzte Druckzeichen einer Programmzeile ist, an, einen folgenden Zeilenumbruch zu ignorieren. Es ist *kein* Tokenbestandteil.

3. Vordefinierte Variablen

Zum Programmstart stehen eine Reihe von Variablen automatisch bereit und können sofort genutzt werden. Dies sind:

	Wert	Bedeutung
<code>.</code>	0	Das Leersymbol – immer mit dem Wert Null
<code>..</code>	1	Der Befehlszeiger mit der Nummer des aktuellen Befehls
<code>pi</code>	3,14159265358979	Die Kreiszahl π
<code>pi/2</code>	1,5707963267949	$\pi/2$
<code>pi/4</code>	0,785398163397448	$\pi/4$
<code>e</code>	2,71828182845906	Die Basis der natürlichen Zahlen e
<code>@</code>	6371004,2029572	Die Erdumfang in Metern

@f	0	Die Abplattung des Erdellipsoids
° (0,0174532925199433	Ein Faktor, der Gradmaß in Bogenmaß umrechnet
(°	57,2957795130824	Ein Faktor, der Bogenmaß in Gradmaß umrechnet
eps	1E-99	Die kleinste Zahl, die sich auf Null addieren lässt
max	9.999999999999999E+99	Die größte zulässige Zahl
r0	0]
r1	0]
r2	0]
r3	0] 8 zur beliebigen freien Nutzung zur Verfügung
r4	0] stehende „Register“
r5	0]
r6	0]
r7	0]

Von besonderer Bedeutung ist das Leersymbol. Es hat den Namen „.“, ist nicht beschreibbar und wird, wenn es gelesen wird, immer mit dem Wert Null gelesen. Wenn ein Befehl zu wenig Operanden enthält, so wird für diese das Leersymbol eingesetzt. Dies führt meistens zu einem sinnvollen Standardverhalten.

Der Befehlszeiger hat den Namen „.“ und enthält immer die Codeadresse (= Programmzeilennummer) des gerade abgearbeiteten Befehls.

Alle anderen vordefinierten Variablen sind wie gewöhnliche Variablen beliebig les- und schreibbar. Man darf also z. B. die Variable „e“ nicht verändern, wenn man zugleich das e als Basis der natürlichen Logarithmen nutzen will.

Hinweis: Die Vimage-Implementation von RT kennt noch einige weitere vordefinierte Variablen, wie z. B. x , x' , y , y' . Dies wird hier nicht weiter behandelt.

4. Befehle

Befehle sind das Herz eines jeden Assemblers. Die meisten Befehle führen Rechenoperationen aus. Darüber hinaus kennt RT Verzweigungsbefehle, Ein- und Ausgabebefehle, Pointerbefehle, Verwaltungsbefehle und Pseudobefehle.

4.1 Rechenbefehle

4.1.1 Einfache Befehle

Der einfachste Befehl ist der Lösch- oder Clearbefehl `clr`. Dieser Befehl setzt den Wert einer Variablen auf Null:

```
clr    a
```

Der häufigste Befehl ist der Transport- oder Move-Befehl `mov`. Mit diesem Befehl wird ein Wert von einem Speicherplatz auf einen anderen kopiert:

```
mov    a    b
```

Dieser Befehl füllt einen Wert „a“ mit dem Zahlenwert der Variablen „b“.

Der der `clr`-Befehl ist ein Einoperandenbefehl. Der der `mov`-Befehl ist ein Zweioperandenbefehl. Der erste Operand, „a“ wird beschrieben, der zweite Operand „b“, gelesen.

Der Inkrementbefehl `inc` erhöht eine Zahl um 1. Der Dekrementbefehl `dec` vermindert eine Zahl um 1:

```
inc    a
dec    a
```

Wie man sieht, sind dies ebenfalls Einoperandenbefehle.

4.1.2 Grundrechenarten

Nur mit `inc` und `dec` zu rechnen, wäre sehr umständlich. Für die Grundrechenoperationen gibt es darum die Befehle

```
add    a    b    ; Addiere b zu a hinzu
sub    a    c    ; Subtrahiere c von a
mul    a    d    ; Multipliziere a mit d
div    a    e    ; Teile a durch e
```

Dies sind Zweioperandenbefehle. Sie funktionieren intern etwas komplexer, als der `mov`-Befehl: Der erste Operand wird gelesen, dann wird der zweite Operand gelesen. Anschließend werden beide Operanden miteinander verknüpft und das Ergebnis wird auf den ersten Operanden zurückgeschrieben. Grundsatz: Der zweite (und bei anderen Befehlen evtl. auch dritte) Operand bleibt in RT immer unverändert. Ein Operand, der geändert wird, steht immer an *erster* Stelle.

Die Schreibweise `add a 10` entspricht also der Schreibweise `a=a+10` höherer Programmiersprachen.

Nun können wir ein erstes Programm schreiben. Dieses soll den Mittelwert „m“ aus zwei Zahlen „z1“ und „z2“ berechnen. Indem man einfach die Befehle zeilenweise hintereinander aufschreibt, entsteht der Code einer Formel:

```
clr    m                ; m löschen
add    m    z1          ; nun steht z1 in m
add    m    z2          ; nun steht z1+z2 in m
div    m    2           ; Mittelwert fertig
```

4.1.3 Potenz und Wurzel

Mit den Befehlen `power` und `root` kann RT Potenzen und Wurzeln berechnen:

power	a	3	; Kubikzahl von a
root	a	2	; Quadratwurzel aus a

Bitte den Exponenten 2 insbesondere bei der Wurzel nicht vergessen. Wurzeln sind in RT (im Gegensatz zu anderen Programmiersprachen) nicht selbstverständlich Quadratwurzeln. Ohne Exponent wird die „nullte“ Wurzel gerechnet.

4.1.4 Logarithmen

Im Gegensatz zu „klassischem Assembler“ unterstützt RT auch zahlreiche Operationen höherer Mathematik. RT kennt vier Befehle, mit denen Logarithmen berechnet werden können, und zwar den Basen e, 10 und 2, sowie zu einer beliebigen Basis:

log	a		; Basis e (natürlich)
log10	a		; Basis 10 (dekadisch)
log2	a		; Basis 2 (dyadisch)
logx	a	b	; beliebige Basis b

4.1.5 Exponentialfunktionen

Die Exponentialfunktionen sind die Umkehrfunktionen der Logarithmen. `exp10` wandelt z. B. eine 3 in eine 1000 um.

exp	a		; berechnet e hoch a
exp10	a		; berechnet 10 hoch a
exp2	a		; berechnet 2 hoch a
expx	a	b	; berechnet b hoch a

4.1.6 Winkelfunktionen

Außer den 4 Haupt-Winkelfunktionen kennt RT die beiden seltener genutzten Winkelfunktionen Secans `sec` und Cosekans `csc`.

sin	a		; Sinus
cos	a		; Cosinus
tan	a		; Tangens
cot	a		; Cotangens
sec	a		; Secans
csc	a		; Cosekans

Es gilt: $\sec \alpha = (1/\cos \alpha)$ und $\csc \alpha = (1/\sin \alpha)$.

4.1.7 Arkusfunktionen

Die Arkusfunktionen sind die Umkehrfunktionen der Winkelfunktionen: Wenn $x = \sin \alpha$ ist, so ist $\alpha = \text{asin } x$.

asin	a	; Arcussinus
acos	a	; Arcuscosinus
atan	a	; Arcustangens
acot	a	; Arcuscotangens
asec	a	; Arcussecans
acsc	a	; Arcuscosekans

4.1.8 Hyperbelfunktionen

sinh	a	; Sinus hyperbolicus
cosh	a	; Cosinus hyperbolicus
tanh	a	; Tangens hyperbolicus
coth	a	; Cotangens hyperbolicus
sech	a	; Secans hyperbolicus
csch	a	; Cosekans hyperbolicus

4.1.9 Areafunktionen

Die Areafunktionen sind die Umkehrfunktionen der Hyperbelfunktionen: Wenn $x = \sinh y$ ist, so ist $y = \operatorname{asinh} x$.

asinh	a	; Area Sinus
acosh	a	; Area Cosinus
atanh	a	; Area Tangens
acoth	a	; Area Cotangens
asech	a	; Area Secans
acsch	a	; Area Cosekans

4.1.10 Logische Operationen

Logische Operationen arbeiten nur mit den beiden „Wahrheitswerten“ 0 (= ‚Falsch‘) und 1 (= ‚Wahr‘). Alle Werte ungleich 0 gelten wie die 1 ebenfalls als ‚Wahr‘. Der Befehl `bin` gibt bei Eingabewert 0 wieder eine 0 zurück, ansonsten eine 1:

bin	a	; Logische Identität
------------	----------	-----------------------------

Dessen Gegenstück ist der Befehl `not`. Dieser macht aus einer 0 eine 1, ansonsten wird eine 0 zurückgegeben. Er ändert (oder negiert) also einen Wahrheitswert von ‚Falsch‘ in ‚Wahr‘ und von ‚Wahr‘ in ‚Falsch‘:

not	a	; Logisch Nicht
------------	----------	------------------------

Der Befehl `and` gibt nur dann eine 1 zurück, wenn beide Eingangswerte, Wert *a* und Wert *b*, ‚Wahr‘ sind. Sofern nur ein Wert auf 0 steht, wird 0 zurückgegeben:

and	a	b	; Logisch Und
------------	----------	----------	----------------------

Dessen Gegenstück ist der Befehl `or`, der immer dann eine 1 zurückgibt, wenn entweder Wert a oder Wert b, 1 sind. Sind beide Werte 0, so wird 0 zurückgegeben:

<code>and</code>	<code>a</code>	<code>b</code>	<code>; Logisch Oder</code>
------------------	----------------	----------------	-----------------------------

Beispiele:

```

mov     a     -10
bin   a
mov     b     0
bin   b           ; b bleibt 0

mov     a     1
not   a           ; not(1) = 0
mov     b     0
not   b           ; not(0) = 1

mov     a     0
and   a     0           ; 0 and 0 = 0
mov     b     1
and   b     0           ; 1 and 0 = 0
mov     c     0
and   c     1           ; 0 and 1 = 0
mov     d     1
and   d     1           ; 1 and 1 = 1

mov     a     0
or    a     0           ; 0 or 0 = 0
mov     b     1
or    b     0           ; 0 or 1 = 1
mov     c     0
or    c     1           ; 1 or 0 = 1
mov     d     1
or    d     1           ; 1 or 1 = 1

```

RT ist ein einfacher Assembler. Darum gibt es keine Unterscheidung von „bitweisen“ und „logischen“ Und-/Oder-/Not-Befehlen etc. RT arbeitet immer „logisch“. (Anmerkung: Weil es keine bitweisen Operationen gibt, gibt es übrigens auch keine bitweisen Verschiebefehle; statt „shift a“ nehme man `mul a 2`.)

4.1.11 Vorzeichenbefehle

Der `neg`-Befehl ändert das Vorzeichen einer Zahl, macht also aus -1 eine $+1$ und aus einer $+1$ eine -1 . Die 0 bleibt unverändert:

<code>neg</code>	<code>a</code>	<code>; Vorzeichenwechsel</code>
------------------	----------------	----------------------------------

Der `abs`-Befehl liefert den absoluten Betrag einer Zahl, d. h. er setzt das Vorzeichen der Zahl immer auf Plus. So werden alle negativen Zahlen positiv:

<code>abs</code>	<code>a</code>	<code>; Absolutbetrag</code>
------------------	----------------	------------------------------

Der `sgn`-Befehl liefert das Vorzeichen einer Zahl. Das Ergebnis ist bei negativen Zahlen `-1`, bei Null `0`, bei positiven Zahlen `+1`:

```
sgn    a                                ; Vorzeichen ermitteln
```

Beispiele:

```

mov     a     -100
neg    a     ; neg(-100) ergibt 100
mov     b     0
neg    b     ; neg(0) ergibt 0
mov     c     1100
neg    c     ; neg(1100) ergibt -1100

mov     a     -100
abs    a     ; abs(-100) ergibt 100
mov     b     0
abs    b     ; abs(0) ergibt= 0
mov     c     1100
abs    c     ; abs(1100) ergibt 1100

mov     a     -100
sgn    a     ; sgn(-100) ergibt -1
mov     b     0
sgn    b     ; sgn(0) ergibt 0
mov     c     1100
sgn    c     ; sgn(1100) ergibt 1

```

4.1.12 Kommastellenbefehle (Rundung und Fraktionierung)

Die Befehle `round`, `ceil`, `floor` und `fix` runden auf verschiedene Art. Der Befehl `fix` ist zugleich der Vorkommastellen-Abtrenner. Er besitzt in `frac`, dem Nachkommastellen-Abtrenner sein Gegenstück:

```

round   a                                ; Runden
ceil    a                                ; Aufrunden
floor   a                                ; Abrunden
fix     a                                ; Zur 0 hin runden
frac    a                                ; Nachkommastellen von a

```

Die Beispiele erläutern die z. T. feinen Unterschiede zwischen den Befehlen:

```

mov     a     3.4
round   a     ; a = 3.0
mov     b     3.6
round   b     ; b = 4.0
mov     c     -3.4
round   c     ; c = -3.0
mov     d     -3.6
round   d     ; d = -4.0

mov     a     3.4
ceil    a     ; a = 4.0

```

```

mov      b      3.6
ceil   b      ; b = 4.0
mov      c     -3.4
ceil   c      ; c = -3.0
mov      d     -3.6
ceil   d      ; d = -3.0

mov      a      3.4
floor  a      ; a = 3.0
mov      b      3.6
floor  b      ; b = 3.0
mov      c     -3.4
floor  c      ; c = -4.0
mov      d     -3.6
floor  d      ; d = -4.0

mov      a      3.4
fix    a      ; a = 3.0
mov      b      3.6
fix    b      ; b = 3.0
mov      c     -3.4
fix    c      ; c = -3.0
mov      d     -3.6
fix    d      ; d = -3.0

mov      a      3.4
frac   a      ; a = 0.4
mov      b      3.6
frac   b      ; b = 0.6
mov      c     -3.4
frac   c      ; c = 0.4
mov      d     -3.6
frac   d      ; d = 0.6

```

4.1.13 Sonstige Rechenbefehle

Der `clip`-Befehl ist ein Dreioperandenbefehl. Der Befehl setzt `a` auf `b`, wenn $a < b$ ist und auf `c`, wenn $a > c$ ist. Ansonsten bleibt `a` unverändert. Der Wert von `a` wird also auf den Wertebereich `b ... c` beschränkt.

<code>clip</code>	<code>a</code>	<code>b</code>	<code>c</code>
-------------------	----------------	----------------	----------------

Im Normalfall soll $b < c$ sein. Dann ist `b` ein Minimalwert und `c` ein Maximalwert. Der Befehl funktioniert aber auch bei $b > c$; `a` wird in jedem Fall in den Bereich zwischen `b` und `c` „geclippt“. Wenn $b = c$ ist, dann wird immer `b (= c)` zurückgegeben.

Der `cmod`-Befehl („clip and modulo“) ist ebenfalls ein Dreioperandenbefehl.

<code>cmod</code>	<code>a</code>	<code>b</code>	<code>c</code>
-------------------	----------------	----------------	----------------

Er funktioniert ähnlich wie der `clip`, nur wird hier mit einer „Sägezahnfunktion“ gearbeitet. Wenn `a` an einer Wertebereichsgrenze den zulässigen Wertebereich verlässt, (z. B. größer als `c` wird), so wird am anderen Ende des Bereiches (z. B. bei `b`) wieder in den Bereich eingetreten. Beispiel:

```

mov      a      99.8
cmov   a      80      100      ; a = 99.8
mov      b      99.9
cmov   b      80      100      ; b = 99.9
mov      c      100.0
cmov   c      80      100      ; c = 100.0
mov      d      100.1
cmov   d      80      100      ; d = 80.1
mov      e      100.2
cmov   e      80      100      ; e = 80.2

```

Dieses Verfahren ist für die Überlaufbehandlung sehr praktisch. So wird z. B. bei umlaufenden Monaten über 12 („Dezember“) wieder beim „Januar“ fortgesetzt. Nach der Uhrzeit 24 Uhr folgt 0 Uhr und auf Winkel 360° folgt der Winkel 0°.

Schließlich liefert der Befehl `random` einen Zufallswert zwischen 0 und 1:

```
random a ; Zufallswert 0 ... 1 liefern
```

Jeder Aufruf liefert einen anderen Wert. Beispiele:

```

random a ; a = 0.584314 ...
random a ; a = 0.489694 ...
random a ; a = 0.128802 ...

```

4.2 Verzweigungsbefehle

Befehle werden normalerweise immer aufeinanderfolgend abgearbeitet, d. h. so, wie sie zeilenweise im Programm stehen. Oft gibt es aber Code, der manchmal mehrfach oder aber überhaupt nicht durchlaufen werden soll. Hierfür gibt es Sprungbefehle. Diese werden in Abhängigkeit von Bedingungen („bedingte Sprünge“) oder immer („unbedingter Sprung“) ausgeführt.

Ein Sprungbefehl führt immer zu einer Marke.

4.2.1 Bedingte Sprünge mit Vergleich

Diese Befehle vergleichen zwei Werte und springen in Abhängigkeit vom Vergleichsergebnis zu einer Marke.

```

cmpgt a      b      m      ; Wenn a>n Sprung zu m
cmpge a      b      m      ; Wenn a≥n Sprung zu m
cmplt a      b      m      ; Wenn a<n Sprung zu m
cmple a      b      m      ; Wenn a≤n Sprung zu m
cmpeq a      b      m      ; Wenn a=n Sprung zu m
cmpne a      b      m      ; Wenn a≠n Sprung zu m

```

`cmpgt` lies als „compare greather than“, `ge` = greather equal, `lt` = less than, `le` = less equal, `eq` = equal, `ne` = not equal.

4.2.2 Bedingte Sprünge mit Test

Oft wird mit Null verglichen. Hierfür gibt es die Reihe der „Testbefehle“. Die Testbefehle gleichen den Compare-Befehlen, der Vergleichswert „b“ entfällt aber.

<code>tstgt</code>	<code>a</code>	<code>m</code>	<code>; Wenn a>0 Sprung zu m</code>
<code>tstge</code>	<code>a</code>	<code>m</code>	<code>; Wenn a≥0 Sprung zu m</code>
<code>tstlt</code>	<code>a</code>	<code>m</code>	<code>; Wenn a<0 Sprung zu m</code>
<code>tstle</code>	<code>a</code>	<code>m</code>	<code>; Wenn a≤0 Sprung zu m</code>
<code>tsteq</code>	<code>a</code>	<code>m</code>	<code>; Wenn a=0 Sprung zu m</code>
<code>tstne</code>	<code>a</code>	<code>m</code>	<code>; Wenn a≠0 Sprung zu m</code>

`tstgt` lies als „test greather than“, `ge` = greather equal, `lt` = less than, `le` = less equal, `eq` = equal, `ne` = not equal.

4.2.3 Der unbedingte Sprung

Wenn in jedem Fall gesprungen werden soll, so nutze man den unbedigten Sprungbefehl `jump`:

<code>jump</code>	<code>m</code>	<code>; Stets Sprung zu m</code>
-------------------	----------------	----------------------------------

4.3 Ein- und Ausgabebefehle

Mit einem Programm soll nicht nur gerechnet werden. Ohne die Möglichkeit, Zahlen in das Programm ein- und ausgeben zu können ist die Rechenmaschine gleichsam gehör- und sprachlos. RT kennt drei Möglichkeiten der Ein- und Ausgabe: Dialoge, einen Ausgabertext und Dateien.

5.3.1 Dialoge

Dialoge kommunizieren mit dem Anwender über Bildschirmein- und Ausgaben. Der Befehl

<code>input</code>	<code>a</code>	<code>Bitte~eine~Zahl~eingeben</code>
--------------------	----------------	---------------------------------------

öffnet auf dem Bildschirm das Fenster



Hier wird der Anwender aufgefordert, eine Zahl eingeben. Diese befindet sich nach Betätigen der Schaltfläche „Übernehmen“ auf der Variablen „a“ abgespeichert. Das Gegenstück bildet der Befehl

```
output c Das~Ergebnis~lautet
```

der mit dem Fenster



eine Variable c auf dem Bildschirm ausgibt.

Schließlich gibt es den Befehl

```
pause c Beginn~des~2.~Teils~der~Berechnung
```

mit dem man einen Mitteilungstext



an den Bediener übermitteln kann.

In allen diesen Fällen hält das Programm an und wartet auf darauf, dass die Schaltfläche „Übernehmen“ bzw. „Fortsetzen“ gedrückt wird. Der Befehl

```
proof c Zwischenergebnis
```

gibt, wie `output` eine Zahl aus, ohne allerdings das Programm anzuhalten. Der Befehl

```
info Zwischenschritt~erreicht
```

zeigt einen kurzen Text, hier „Zwischenschritt erreicht“, an, ebenfalls ohne dass das Programm unterbrochen wird.

4.3.2 Der Ausgabertext

Der Ausgabertext ist ein großer Textspeicher, der global im Hintergrund existiert. In diesen Text lässt sich wie auf einen Drucker Text in einzelnen Zeilen ausgeben.

```
printn a b c ; Zahl a drucken  
prints s ; Zeichenkette s drucken
```

Der Befehl `printn` druckt den Symbolwert der Variablen `a` mit `b` Vorkomma- und `c` Nachkommastellen. Insgesamt werden immer `b+c+2` Stellen ausgegeben, denn auch für ein Vorzeichen und einen Dezimalpunkt muss Raum sein.

Der Befehl `prints` druckt den Symbolnamen `s` als Zeichenkette, also hier genau ein „s“.

Die Ausgabe

```
printn -111.1 3 2  
printn -222.2 3 2  
printn -333.3 3 2  
printn -444.4 3 2
```

erzeugt den Text

```
-111.1-222.2-333.3-444.4
```

Es wird wirklich nur das ausgegeben, was der Befehl anweist. Soll tabelliert, in Zeilen gegliedert (und noch eine Überschrift hinzugefügt) werden, so schreibe man:

```
prints Rechenergebnis:\\  
printn -111.1 3 2  
prints ~~~  
printn -222.2 3 2  
prints \\  
printn -333.3 3 2  
prints ~~~  
printn -444.4 3 2  
prints \\
```

Leerzeichen und Zeilenumbruchszeichen sind also gesondert hinzuzufügen. Man erhält dann:

```
Rechenergebnis:  
  
-111.10 -222.20  
-333.30 -444.40
```

Vor dem ersten „Ausdruck“ kann der Ausgabertext gelöscht werden. Das geschieht mit dem Befehl

```
cls
```

Mit dem Befehl

```
save    ergebnis
```

kann man den gesamten Ausgabertext in einer Datei „ergebnis“ ablegen. `ergebnis` ist hier eine Zeichenkette. Das Token „ergebnis“ ist also ein Dateiname, es wird der Standarddateityp „txt“ angehängt, so dass der Dateiname zu „ergebnis.txt“ wird.

Anmerkung: Aus Betriebssystemgründen dürfen im Dateinamen nur Buchstaben, Ziffern, `_`, `(`, `)` und `$` stehen. Buchstaben werden in Kleinbuchstaben umkodiert, Fremdzeichen in den Unterstrich.

4.3.3 Ein- und Ausgabe in Dateien

Variablen können auch in Dateien geschrieben und aus Dateien gelesen werden. Dies erfolgt mit den Befehlen

```
write   a      1
read    b      1
```

Der `write`-Befehl schreibt den Wert der Variablen `a` in eine Datei „a.dat“.

Der `read`-Befehl liest eine Datei „a.dat“ und ordnet den ermittelten Wert der Variablen `a` zu.

Der 2. Operand „1“ ist speziell für Lese-/Schreib-Operationen aus/in Felder(n) vorgesehen, siehe hierzu unten. Man lasse ihn in erster Näherung weg.

Aus Betriebssystemgründen dürfen im Dateinamen nur Buchstaben, Ziffern, `_`, `(`, `)` und `$` stehen. Alle Buchstaben werden in Kleinbuchstaben umkodiert, Fremdzeichen, auch Umlaute in den Unterstrich.

4.4 Pointerbefehle und Felder

4.4.1 Pointer

Ein Ausschnitt aus der Symboltabelle habe folgendes Aussehen:

Symboladresse	Symbolname	Symbolwert
112	v1	6
113	v2	7
114	pV	-5

Im Normalfall wird ein Symbol über seinen Symbolnamen angesprochen. So wird z. B., um `V1` mit einer 10 zu beschreiben, der Befehl `mov V1 10` genutzt. Dass sich `V1` in der 112. Zeile der Symboltabelle befindet („auf Adresse 112“) ist hierbei uninteressant.

Der Befehl

```
adrof   p      v
```

trägt nun die *Adresse* des Symbols mit dem *Symbolnamen* `v` auf dem *Symbolwert* von „p“ ein. Damit wird „p“ zu einem *Pointer* auf die Variable `v`.

Nun sind Lese- und Schreiboperationen über „den Pointer“ möglich. Hierzu dienen die Befehle `put` und `get`:

```

put    p    q    v    ; Schreiben per Pointer
get    v    p    q    ; Lesen per Pointer

```

Derartige Zugriffe heißen auch indirekte Lese-/Schreibzugriffe. Die Variable „q“ heißt Offset wird zunächst noch nicht genutzt. Sie sei Null, wofür wir einstweilen das Leersymbol `.` notieren werden.

Der Befehl `put p q v` überträgt den **Wert** des Symbols mit dem **Namen** „v“ auf den **Wert** des Symbols mit der **Adresse** „p“. Man sagt: Er schreibt v *indirekt* nach p.

Der Befehl `get v p q` überträgt den **Wert** des Symbols mit der **Adresse** „p“ auf den **Wert** des Symbols mit dem **Namen** „a“: Er liest p *indirekt* nach v.

Unsere Symboltabelle

112	v1	6
113	v2	7
114	pV	-5

erhlt somit durch die Ausführung des Befehls

```

adrof  pV    v1    ; pV wird Pointer auf V1

```

folgendes Aussehen:

112	v1	6
113	v2	7
114	pV	112

Das Symbol pV ist damit zu einem Pointer auf die Variable V1 geworden. Nun führen wir den folgenden Befehl aus:

```

put    pV    .    10    ; V1 über Pointer beschreiben

```

Es bietet sich folgendes Bild:

112	v1	10
113	v2	7
114	pV	112

Wozu nützt nun indirektes Lesen und Schreiben? Neu ist, dass man Pointer vom Programm aus verändern und somit berechnen kann. Nach



```
add    pV    1                ; pV wird um 1 erhöht
```

zeigt pV nun nicht mehr auf die Variable V1, sondern auf die Variable V2:

112	v1	10
113	v2	7
114	pV	113

Der Pointer wurde also „um 1 weitergestellt“. Wenn man jetzt den gleichen Befehl wie oben

```
put    pV    .    10
```

nochmals ausführt, wird nicht mehr V1, sondern V2 beschrieben:

112	v1	10
113	v2	10
114	pV	113

Eine solche „Pointerweiterstellung“ ist mit dem gewöhnlichem Zugriff (z. B. mit `MOV`) nicht möglich.

Bleibt noch die Bedeutung der Offsetvariablen nachzutragen. Der Offset `q` wird lediglich zum Pointer `p` hinzuaddiert, bevor der Zugriff erfolgt. So lassen sich solche „Pointersprünge“ wie soeben von V1 nach V2 sehr elegant schreiben:

```
adrof   pV    V1
put     pV    0    10    ; V1 mit 10 beschreiben
put     pV    1    10    ; V2 mit 10 beschreiben
```

4.4.2 Felder

Der Pointerzugriff ist insbesondere beim Arbeiten mit Feldern sehr interessant. Was sind Felder? Hierzu nehmen wir den Pseudobefehl `_dim` vorweg. Der Befehl:

```
_dim F 3 ; Feld F mit 3 Elementen
```

sorgt dafür, dass sich beim Programmstart ein Symbol `F` in der Symboltabelle befindet. Auf dieses Symbol folgen in der Tabelle 4 weitere Symbole mit den Symbolnamen `F(0)`, `F(1)`, `F(2)` und `F(3)`. Schließlich bewirkt der `_dim`-Befehl noch, dass auf dem Symbolwert von `F` die Symboladresse von `F(0)` eingetragen wird.

Statt 3 sind natürlich auch beliebig andere Feldlängen möglich. Allerdings muss die Feldlänge immer mit einer Zahl angegeben werden. Eine Variable, z. B. in `dim F c` würde nicht funktionieren.

Die Symbole `F(0)`, `F(1)`, `F(2)`, `F(3)` heißen *Feldelemente*. Die Zahlen 0, 1, 2, 3 heißen *Feldindizes* der Feldelemente. Das Symbol `F` heißt *Feldpointer*. Ein Feld ist eine Datenstruktur, in

der mehrere Zahlen gespeichert werden können.

Ein Beispiel: Der Befehl `_dim Q 4` legt zum Programmstart z. B. folgendes Feld an:

Symboladresse	Symbolname	Symbolwert
36	Q	37
37	Q(0)	0
38	Q(1)	0
39	Q(2)	0
40	Q(3)	0
41	Q(4)	0

Nun stehe die Beispielaufgabe, das Feld Q mit Quadratzahlen zu füllen. Dies kann mit folgendem Code geschehen:

```
mov     Q(0)    0           ; 0 zum Quadrat
mov     Q(1)    1           ; 12
mov     Q(2)    4           ; 22
mov     Q(3)    9           ; 32
mov     Q(4)   16           ; 42
...     
```

Wenn man nun aber die Quadrate bis 1000 berechnen will, wird dies sehr aufwändig. Außerdem lautet die Aufgabe ja, dass das Programm die Quadrate *berechnen* soll und nicht, dass der Programmierer die Quadratzahlen in den Code *eintippen* soll.

Wie wäre es mit folgenden Code?

```
_dim    Q        1000      ; Feld F(0) ... F(4) erzeugen
clr     i                ; Index i löschen

$m1:
mov     r0       i        ; i nach Register r0
power   r0       2        ; r0 quadrieren
put     Q        i        r0 ; r0 nach Q(i) schaffen
add     i        1        ; i weiterstellen
cmple  i        1000     $m1 ; Solange i≤100 nach $m1
```

Der Code enthält eine Schleife. Diese beginnt bei der Marke \$m1. Weiterhin gibt es eine wichtige Variable i. Dies ist der Feldindex oder kurz Index. Dieses i wird zu Beginn der Rechnung mit einem `clr`-Befehl auf 0 gesetzt.

Der Index wird nun dreifach genutzt:

Erstens ist i die Zahl, die quadriert wird. Der `mov`-Befehl schafft i nach Register r0. Dort erfolgt das Quadrieren in dem anschließenden `power`-Befehl.

Zweitens dient i im `put`-Befehl als Offset. Bei einem i von 0 erreicht der `put` Feldelement F(0), i-Wert 1 „adressiert“ Feldelement F(1), i-Wert 2 Feldelement F(2) usw.

Drittens wird anhand des *i*-Wertes im `cmple`-Befehl geprüft, ob gesprungen werden soll. Dies erfolgt immer, sofern *i* den Wert 1000 noch nicht erreicht hat.

Der `add`-Befehl stellt *i* bei jedem Schleifendurchlauf jeweils um 1 weiter.

In diesem Code kann man nun einfach die „1000“ im `_dim` und im `cmple` gegen eine „10000“ ändern und schon werden die Quadratzahlen bis 10000 berechnet!

Anmerkung: Eine Notation „*Q(i)*“ als Variablenname (so wie etwa in höheren Programmiersprachen) ist in Assembler nicht zulässig. Mit etwa einem `mov Q(i) a` würde in RT lediglich eine Variable „*Q(i)*“ neu definiert und dann mit *a* beschrieben, keinesfalls aber das „*i*-te“ Feldelement von *Q*.

4.4.3 Die Befehle `read` und `write` in Felder

Der 2. Operand in den Befehlen `read` und `write` ist ein Längenparameter, der die Anzahl der Symbole angibt, die fortlaufend aus der Symboltabelle gelesen/geschrieben werden. Es wird jeweils *ein Symbol mehr geschrieben*, als angegeben ist. Mit dieser Regelung ist es leicht möglich, Felder in Dateien zu schreiben und zu lesen. Der Befehl

```
write    Q        4
```

erzeugt eine Datei „*q.dat*“. Diese enthält die 5 Textzeilen

```
37q
0q
1q
4q
9q
```

Die erste Zeile enthält den Feldpointer. Dieser ist uninteressant und braucht nicht weiter beachtet werden. Es folgen die Feldelemente *Q(0)*, *Q(1)*, *Q(2)*, *Q(3)*.

Das Gegenstück zum `write` bildet der `read`-Befehl, mit dem ein Feld aus einer Datei in die Symboltabelle eingelesen wird. Wichtig: Beim Einlesen eines Feldes aus einer Datei mit dem `read`-Befehl muss der Feldpointer unbedingt richtig eingestellt werden. Das muss mit einem dem `read` unmittelbar folgenden `adrof`-Befehl geschehen:

```
read     Q        4
adrof    Q        Q(0)
```

Ohne diesen `adrof` würde der Feldpointer auf einen falschen Bereich der Symboltabelle zeigen. Dies hätte dann zur Folge, dass nachfolgende `get` falsch lesen und nachfolgende `put` sogar die Symboltabelle zerstören können.

Merke:

- Ein Feld, dessen letztes Element *F(100)* ist, wird mit `_dim F 100` angelegt.

- Da die Zählung mit dem 0. Element beginnt, ist die Anzahl der Feldelemente immer um 1 höher, als der Index des letzten Feldelementes angibt. Unser Feld hat also **101** Elemente. Folglich muss das Feld mit dem `write F 101` geschrieben und dem `read F 101` gelesen werden.
- Tatsächlich kommt auch noch der Feldpointer hinzu. In der Symboltabelle belegt das Feld also (mit 0. Element und Feldpointer) **102** Speicherplätze. In der Datei belegt das Feld ebenfalls **102** Zeilen.

4.5 Verwaltungsbefehle

Der Befehl

init

bereitet die Maschine auf ein Programm vor. Der Befehl wird aber beim Assemblieren automatisch als erster Befehl in den Code eingefügt, daher braucht man ihn nicht weiter zu beachten.

Der Befehl

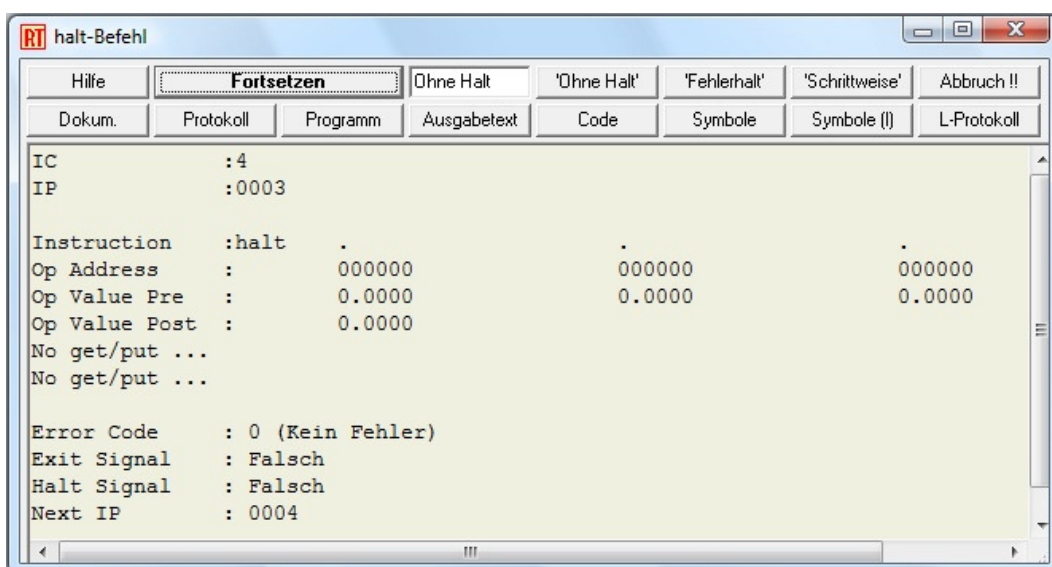
nop

– „no operation“ – macht überhaupt nichts (außer das er wenig Rechenzeit verbraucht). Eigentlich gibt es den `nop` nur, weil er ein Klassiker ist, den jedem Assembler-Codesatz enthält.

Ein weiterer Klassiker ist der Befehl

halt

Der `halt` unterbricht die Programmabarbeitung und hält die Maschine an. Es öffnet sich ein Fenster:



HALT_Befehle waren früher wichtig, als es noch keine Debugger gab und man das Programm zur Fehlersuche „zu Fuß anhalten“ musste. Es ist für jeden Assembler-Befehlssatz guter Stil und

unbedingte Ehrenpflicht, über einen HALT-Befehl zu verfügen. Dessenungeachtet gilt aber: **In einem fertigen Programmen hat der HALT-Befehl nichts zu suchen.**

Der Befehl

```
mode a
```

erlaubt es, einen von 3 Testmodi einzustellen:

- a = 0: Modus ‚Ohne Halt‘: Das Programm läuft durch und hält nie an (außer beim HALT-Befehl)
- a = 1: Modus ‚Fehlerhalt‘: Das Programm hält in Fehlerfällen an.
- a = 2: Modus ‚Schrittweise‘: Das Programm hält nach jedem Befehl an. So kann man jeden einzelnen Befehl kontrollieren.

Ein ausgetestetes Programm sollte immer mit Modus 0 oder 1 arbeiten.

Eine andere Fehlerverfolgungsmöglichkeit bieten die beiden Fehlerbefehle

```
errcode a  
errjump m
```

Der `errcode` überträgt den aktuellen Fehlercode auf eine Variable `a`. Der `errjump` ist eigentlich ein Verzweigungsbefehl, der in Fehlerfällen zur Marke `m` springt. So lassen sich Fehlerbehandlungen organisieren, z. B.:

```
div a b  
errjump $m9  
...  
$m9: pause Division~durch~Null
```

Der letzte Verwaltungsbefehl ist der Wichtigste: Der `exit`-Befehl beendet die Programmabarbeitung.

```
exit
```

Die Abarbeitung eines Programmes sollte immer mit dem `exit`-Befehl enden.

4.6 Pseudobefehle

Pseudobefehle werden nicht in den Code übernommen und führen damit keine Operationen „auf der Maschine“ aus. Es handelt sich vielmehr um Anweisungen, die bei der Aufbereitung des Programmes („zur Assemblerzeit“, s. u.) bedeutsam sind und vor dem Programmstart ausgeführt werden.

```
_name Programmname
```

Mit dem Pseudobefehl `_name` kann dem Programm ein Programmname „Programmname“ gegeben werden.

RT ist (wie z. B. Fortran oder Basic) eine selbstdeklarierende Programmiersprache. Jedes neu auftauchende Symbol wird bei seinem ersten Auftreten selbsttätig in der Symboltabelle eingetragen. Mit dem Pseudobefehl

```
_var a
```

lässt sich die Eintagung eines Symbols aber auch gesondert deklarieren. So kann man z. B. eine bestimmte Anordnung der Symbole in der Symboltabelle bewirken. Auch machen Symboldeklarationen Programme leichter lesbar. Ein weiterer Wert einer derartigen Deklaration besteht darin, dass sich infolge von Tippfehlern verschriebene nichtdeklarierte Symbole „unten“ in der Symboltabelle ansammeln. So lassen sich Tippfehler leicht finden.

```
_dim F 99
```

Der `_dim`-Pseudobefehl ähnelt dem `_var` und wurde bereits oben behandelt. Er erzeugt ein Feld `F` einer bestimmten Länge, hier 99. Ein Feld erzeugen heißt, ein Symbol „`F`“ (Feldpointer) und $99+1$ Symbole „`F(0)`“, „`F(1)`“ ... „`F(99)`“ (Feldelemente) in die Symboltabelle einzutragen. Weiterhin trägt der `_dim` die Adresse von `F(0)` auf `F` ein. Dies bereitet spätere Zugriffe mit den Befehlen `put` und `get` vor.

Im Gegensatz zu den Variablen besteht bei Feldern Deklarationspflicht. Andernfalls wüsste das Programm ja nicht, wie lang das Feld sein soll. Zu beachten ist auch, dass die Feldlänge eine Zahl sein muss, also z. B. „10“ oder „99“. Eine Variable, z. B. „a“ würde als 0 aufgefasst und damit ein Feld erzeugen, welches lediglich aus einem Feldpointer und einem nullten Feldelement besteht.

Der Pseudobefehl

```
_lab m
```

erzeugt ein Symbol `m` und trägt auf diesem die aktuell vom Assembler erreichte Codeadresse ein. Damit wird `m` zu einer Marke, die z. B. von Sprungbefehlen angesprungen werden kann.

Für den Pseudobefehl `_lab m` gibt es die Alternativschreibweise `m:`. Diese wird für den gewöhnlichen Gebrauch eher empfohlen. So werden Programme leichter lesbar.

(Anmerkung: Die `m:`-Form ist tatsächlich nur eine andere Schreibweise, die den `_lab` „überdeckt“. Deshalb dürfen Markenzeilen (im Gegensatz zu anderen Assemblern) in RT keine (anderen) Befehle enthalten.)

Der Pseudobefehl

```
_config 1
```

ist dafür vorgesehen, der virtuelle Maschine einen Konfigurationsparameter, hier z. B. „1“, mitteilen zu können. Der Konfigurationsparameter muss immer eine Zahl sein, Variablen, z. B. „a“ würden als Null aufgefasst. Der Pseudobefehl ist lediglich reserviert; er wird nicht ausgewertet.

Schließlich zeigt der Pseudobefehl

```
_end
```

an, dass ein Programm zu Ende ist. Während `exit` das Ende des Programmlaufes anzeigt, zeigt `_end` die letzte Zeile einer Quelldatei an.

5. Kontrollstrukturen

Indem man Befehlsfolgen zusammensetzt, entstehen Abläufe, die komplexe Algorithmen realisieren. Dabei sind Sprünge, Marken, Test- und Vergleichsbefehle wichtig.

5.1 Die Ja-Nein-Entscheidung („if-Anweisung“)

Es wird nur dann `b` mit 10 multipliziert, wenn `a` nicht 0 ist ...

```
        tsteq    a        $m1        ; Wenn a=0 Sprung nach $m3
        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 eingegengesetzt sind.

5.2 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.

5.3 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:

```

;
;0 Wenn c≠0 bei $m1 weiter
;0 Wenn c=1 .. Addieren
;0 Wegspringen zu $m9
$m1:
    cmpne    c        0        $m1
    add     a        b
    jump    $m9
;1 Wenn c≠1 bei $m2 weiter
;1 Wenn c=2 .. Subtrahieren
;1 Wegspringen zu $m9
$m2:
    cmpne    c        1        $m2
    sub     a        b
    jump    $m9
;2 Wenn c≠2 bei $m3 weiter
;2 Wenn c=3 .. Multiplizieren
;2 Wegspringen zu $m9
$m3:
    cmpne    c        2        $m3
    mul     a        b
    jump    $m9
;3 Wenn c≠4 bei $m4 weiter
;3 Wenn c=4 .. Dividieren
;3 Kann man auch weglassen
$m9:
    cmpne    c        3        $m9
    div     a        b
    jump    $m9
...
;

```

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

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

Dies ist eine besonders schnelle Auswahlentscheidung und zugleich eine Assembler-Rafinesse, die es in anderen Programmiersprechen kaum gibt. Sie 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.

Wir notieren:

```

; d ist jetzt 0, 1, 2 oder 3
; d ist jetzt 0, 2, 4 oder 6
; d ist jetzt $m0, $m1, $m2 oder $m3!!
; nun Sprung nach $m0, $m1, $m2, $m3
$m0:
    mov     d        c
    mul     d        2
    add     d        $m0
    jump    d
;0
;0
$m1:
    sub     a        b
    jump    $m9
;1
;1
$m2:
    mul     a        b
    jump    $m9
;2
;2
$m3:
    div     a        b
    jump    $m9
;3
;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 (1.) der jeweilige `add/sub/mul/div` zwischen ihnen 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 sog. *relativen Codeadresse*.

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

4. Nun kommt der Clou: 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.

Übrigens: Die Marken \$m1, \$m2, \$m3 kann man im Programmtext sogar weglassen!

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

Weil das Wissen um derartige Codierungen allmählich ausstirbt, nutzen Virenprogrammierer und andere zwielichtige Gestalten solche Codes mitunter, um die wahren Absichten ihrer Programme abzutarnen. Wenn in einem lichtscheuen Code „Daten“ in den „Befehlszähler“ geraten, sollte man als „Softwarepiratenjäger“ hellhörig werden.

5.5 Die Solange-Bedingungsschleife („do ... while-Anweisung“)

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

```
$m1:      mov     a      100000      ; Variable laden
          ...
          ...                ;# 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 irgendwann 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.

5.6 Die Bis-Bedingungsschleife („do ... until-Anweisung“)

Eine andere Variante hat folgenden Code:

```
$m1:    mov     a      100000      ;
        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 $a \leq 1$ irgendwann 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.

5.7 Die Zählschleife („for-Anweisung“)

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

```
$m1:    mov     c      10          ; Zähler i mit 10 füllen
        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` und nicht mit Addition oder Subtraktion.

5.8 Der Unterprogramm sprung und -rück sprung („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.

RT verfügt über keine speziellen Unterprogrammbe fehle. Das hätte den Sprachentwurf zu sehr aufgebläht. 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:
    output  r2      Mittelwert
    ...

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

.subgeo:
    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.

Für die *Parameterübergabe* an das Unterprogramm und zurück ist es assemblertypisch, Register zu benutzen. Hierfür eignen sich die vordefinierten Variablen r0 bis r7.

Eine andere Möglichkeit besteht darin, zur Parameterübergabe bestimmte vom Programm definierte Variablen zu benutzen.

Beim Schreiben von Unterprogrammen in RT muss darauf geachtet werden, dass Variablen im Unterprogramm und im Hauptprogramm immer unterschiedliche Namen haben. RT kennt nämlich

immer nur eine einzige Symboltabelle („einen Namensraum“). Wenn das Hauptprogramm einen Wert auf einer Variablen „a“ speichert, und das Unterprogramm ebenfalls „a“ beschreibt, wird damit der Hauptprogramm-Wert zerstört.

6. Fehlersuche

Zur Fehlersuche gibt es drei Modi der Programmabarbeitung, vgl. hierzu auch den Befehl `mode`:

- Modus *„Ohne Halt“*: Das Programm läuft durch und hält auch in Fehlerfällen nicht an.
- Modus *„Fehlerhalt“*: Das Programm hält in Fehlerfällen an.
- Modus *„Schrittweise“*: Es wird nach jedem Befehl angehalten. So lässt sich jeder Befehl kontrollieren.

Weiterhin gibt es zwei Signale, mit denen man das laufende Programm beeinflussen kann.

- Das *Haltsignal* wirkt wie der `halt`-Befehl und unterbricht das Programm.
- Das *Abbruchsignal* bewirkt eine sofortige Programmbeendigung. Es wirkt wie der `exit`-Befehl.

Zum Setzen dieser Signale gibt es entsprechende Schalflächen der Bedienoberfläche. In der Vimage-Implementation setzt die Eingabe von `<Strg/Z>` das Abbruchsignal.

In Fehlerfällen wird ein Fehlercode erzeugt, auf den mit den Befehlen `errcode` und `errjump` reagiert werden kann. Die Fehlercode 116 bis 120 sind Assemblerfehler. Hier ist das Programm selbst fehlerhaft und muss korrigiert werden. Die Fehler 101 bis 113 sind Laufzeitfehler, d. h. diese werden bei der Abarbeitung durch fehlerhafte Daten verursacht. Es gibt folgende Fehlercodes:

101	Überlauf. Betrag > 9E99
102	Division durch 0
103	Potenz 0 hoch 0
104	Rationale Potenz von Zahl < 0
105	Rationale oder gerade Wurzel aus Zahl < 0
106	Wurzelexponent 0
107	Logarithmus aus Zahl < 0
108	Logarithmus aus 0
109	Logarithmenbasis < 0
110	Logarithmenbasis 0
111	Logarithmenbasis 1
112	Funktionswert nicht definiert
113	Dateifehler
116	Unbekannter Befehl
117	Symbol nicht definiert
118	Symbol bereits definiert
119	Symboltabelle voll
120	Symbolname länger als 1024 Zeichen

7. Programmbeispiele

Hallo World

Mit Hallo World fängt jede Programmiersprache an ...

```
;
; HALLO WORLD!
; =====
;
;           _name      Hallo~World
;           pause      Hallo~World!
;           exit
;           _end
```

Das Programm besteht aus lediglich 2 Befehlen: Der Befehl `pause` gibt „Hallo World!“ aus und der Befehl `exit` beendet das Programm. Hinzu kommt der Pseudobefehl `_name`, der dem Programm den Namen „Hallo World“ gibt und `_end`, das die letzte Zeile kennzeichnet. Der Rest ist Kommentar.

Mittelwert zweier Zahlen

Es werden zwei Zahlenwerte im Dialog (`input`-Befehl) abgefragt, deren Mittelwert berechnet und das Ergebnis angezeigt (`output`-Befehl).

```
           _name      Mittelwertberechnung      ; Programmnamen
           _var       z1                          ; Variable z1 deklarieren
           _var       z2                          ; Variable z2 deklarieren
           _var       m                            ; Variable für Mittelwert dekl.
           input      z1      Erste~Zahl         ; Erste Zahl abfragen
           input      z2      Zweite~Zahl        ; Zweie Zahl abfragen
           mov        m        z1                ; z1 nach m transportieren
           add        m        z2                ; z2 zu m addieren
           div        m        2                 ; Division durch 2
           output     m        Mittelwert        ; Mittelwert ausgeben
           exit
           _end                                     ; Das ist die letzte Zeile
```

Berechnung der Fakultäten von 1 bis 20

Unter der Fakultät einer (natürlichen) Zahl versteht man das Produkt aller natürlichen Zahlen von 1 bis zu dieser Zahl. Das Fakultätszeichen ist in der Mathematik das Ausrufungszeichen. Der Wert von $4!$ ist also mit $1 \cdot 2 \cdot 3 \cdot 4$, die Zahl 24. Fakultäten werden ganz schön schnell ganz schön groß. Wie groß? In RT kann man das wie folgt programmieren:

```
;
; FAKULTÄT
; =====
;
; DEKLARATIONEN
;
;           _name      Fakultät
;           _var       factor                          ; Laufender Faktor
;           _var       f                            ; Fakultät
;
; INITIALISIERUNGEN
;
;           pause      Das~Programm~berechnet~Fakultäten~von~1~bis~20.
;           mov        factor      1                  ; laufenden Faktor auf 1 setzen
```

```

        mov     f         1           ; die Fakultät auf 1 setzen
        cls                     ; Ausgabertext löschen
;
; SCHLEIFE
$loop:
        add     factor    1           ; faktor eins weiter
        mul     f         factor      ; fakultät aufmultiplizieren
        cmpgt  factor    20         $ende ; Bei factor 20 abbrechen
        printn factor     2         0   ; factor mit 2 Stellen ausgeben
        prints !:~              ; !, : und Leerzeichen ausgeben
        printn f         19         0   ; Fakultät mit 19 St. ausgeben
        prints  \                ; Zeilenvorschub ausgeben
        jump   $loop              ; nächste Schleife
;
; ENDE
$ende:
        save    fakultäten
        pause   Das~Ergebnis~wurde~in~die~Datei~fakultäten.txt~geschrieben.
        exit
        _end

```

Außer der eigentlichen Berechnung gibt es hier etwas Ein-/Ausgabe. Die `pause`-Befehle teilen dem Anwender mit, was das Programm macht und die Befehle `cls`, `printn`, `prints` und `save` schreiben das Rechenergebnis in den globalen Ausgabertext und in eine Datei „fakultäten.txt“.

Berechnung der Zahl e mit einer Reihenentwicklung

Reihenentwicklungen gehören schon zu den anspruchsvolleren Aufgaben. So kompliziert, wie man denkt, ist eine Reihenentwicklung nun auch wieder nicht. Um die Eulersche Zahl e zu berechnen, gibt es z. B. die Formel

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

Und so codet man das Ganze in RT ab:

```

;
; EULERSCHE ZAHL e BERECHNEN
; =====
;
; DEKLARATIONEN
;
        _name    Euler
        _var     e                ; e
        _var     n                ; Zykluszähler
        _var     f                ; Fakultät
        _var     restgl           ; Restglied
;
; INITIALISIERUNGEN
;
        pause    Das~Programm~berechnet~die~Zahl~e~mit~einer~Reihenentwicklung.
        mov     e         1         ; e ist erst einmal 1
        clr     n                ; Zykluszähler löschen
        mov     f         1         ; die Fakultät von 0 ist 1
        clr     restgl         ; Restglied löschen
        cls                     ; Ausgabertext löschen
;
; HAUPTSCHLEIFE
$loop:
        add     n         1         ; Zykluszähler inkrementieren
        mov     restgl    1         ; Zähler des Bruches: 1
        mul     f         n         ; Fakultät weitermultiplizieren
        div     restgl    f         ; Bruch berechnen
        add     e         restgl     ; Auf e aufsummieren
        printn n         2         0   ; n mit 2 Vorkommastellen

```

```

prints      :~                ; Doppelpunkt und Leerzeichen
printn     e      1      14    ; e mit 14 Stellen ausgeben
prints     \                ; Zeilenvorschub ausgeben
mov        r0      restgl     ; restgl auf Hilfszelle r0
abs        r0                ; Absolutbetrag bilden
cmpgt     r0      1e-16      $loop ; restgl<1e-16 weitermachen
;
; ENDE
$ende:
save      e                ; Text in Datei e.txt schreiben
pause    Jetzt~steht~e~in~der~Datei~e.txt ; Abschlussmitteilung
exit
_end

```

Auch hier erfolgt zusätzlich zur eigentlichen Berechnung wieder eine Ausgabe in den globalen Ausgabertext und in eine Datei, hier mit Namen „e.txt“.

8. Anmerkungen

8.1 Wie RT intern funktioniert

Computersprachen erscheinen häufig als etwas sehr Kompliziertes und Undurchschaubares. Assembler im Allgemeinen und RT im Besonderen funktionieren aber ganz einfach.

Wie bei allen Programmiersprachen gibt es zwei Abarbeitungsphasen. In einer *Assemblerzeit* wird das RT-Quellprogramm von einem Programm – *dem RTA-Assembler* – eingelesen und aufbereitet.

Dann folgt eine *Laufzeit*, in der die eigentliche Programmabarbeitung erfolgt. Klassische Assemblerprogramme werden auf einer CPU ausgeführt, die die Befehle hardwaremäßig umsetzt. Die Assemblerbefehle sind dann zugleich Maschinenbefehle, die als Operationen auf einer festen elektronischen Schaltung ausgeführt werden.

Für RT gibt es keine derartige Hardware und auch keine Maschinenbefehle. RT ist (wie z. B. auch Java oder MIXAL) eine synthetische Sprache, die ohne Rücksicht auf eine bestimmte Hardware entworfen wurde. Eine Hardware kann hier also keine Berechnungen ausführen. Folglich muss auch dies eine Software besorgen. Der Java-Bytecode läuft auf der Java Virtual Machine. Im Fall von RT gibt es ebenfalls eine derartige virtuelle Maschine. Dies ist der *RT-Prozessor* oder die *RT-Maschine*.

Was passiert nun im Einzelnen?

Zur Assemblerzeit: Der *RT-Assembler* übersetzt das Quellprogramm. Übersetzen heißt, dass 2 Tabellen erzeugt werden: eine Codetabelle und eine Symboltabelle. Jeder Befehlszeile des Programms wird in eine Zeile der Codetabelle übersetzt. Der Befehlscode wird übernommen und den Operanden werden 3 Symboladressen zugeordnet. Nicht vorhandene oder nicht benötigte Symbole erhalten die Symboladresse Null. (Dies ist die Adresse des Leersymbols.) Pseudobefehle erzeugen keinen Code. Gleichzeitig wird die Symboltabelle aufgebaut. Ist ein Symbol bereits in der Symboltabelle vorhanden, so gibt es schon eine Symboladresse. Diese braucht nur noch entnommen zu werden. Noch nicht verzeichnete Symbole werden neu eingetragen und erhalten eine Symboladresse am Tabellenende zugewiesen. Wenn der Symbolname eine Zahl ist, so wird diese Zahl auf dem Symbolwert eingetragen, ansonsten Null. Wenn der Pseudobefehl `_end` erreicht ist, beendet der Assembler seine Arbeit.

Das ist alles!

Das Assemblieren erfolgt (jit; just in time) immer unmittelbar vor dem Programmstart. Weil es nur einen Sekundenbruchteil dauert, merkt man gewöhnlich gar nicht, das assembliert wird.

Laufzeit: Nun kann die Programmabarbeitung folgen. Der *RT-Prozessor* erhält Code- und Symboltabelle übergeben. Er beginnt er die Programmabarbeitung mit der Ausführung des ersten Befehls der Codetabelle. Je nach Befehl werden die Symbolwerte in der Symboltabelle gelesen, berechnet und wieder in die Tabelle zurückgeschrieben. Nach den meisten Befehlen wird mit der nächsten Zeile der Codetabelle fortgesetzt. Mit Sprungbefehlen kann aber auch zwischen Zeilen der Codetabelle gesprungen werden. Der Befehl `exit` am Programmende bewirkt die Programmbeendigung.

Das ist alles!

Ein Beispiel

Zur Illustration lassen wir das folgende Beispielprogramm „Mittelwertberechnung“ vom Assembler übersetzen:

Quellprogramm:

```

_name      Mittelwertberechnung      ; Programmnamen
_var       z1                        ; Variable z1 deklarieren
_var       z2                        ; Variable z2 deklarieren
_var       m                         ; Variable für Mittelwert dekl.
input      z1      Erste~Zahl        ; Erste Zahl abfragen
input      z2      Zweite~Zahl       ; Zweie Zahl abfragen
mov        m       z1                ; z1 nach m transportieren
add        m       z2                ; z2 zu m addieren
div        m       2                 ; Division durch 2
output     m       Mittelwert        ; Mittelwert ausgeben
exit                                     ; Abarbeitung beenden
_end                                     ; Das ist die letzte Zeile

```

Der Assembler übersetzt das Quellprogramm in folgende Symboltabelle

Symboladresse	Symbolname	Symbolwert
0 [Das Leersymbol]	.	0
...
36	z1	0
37	z2	0
38	m	0
39	Erste~Zahl	0
40	Zweite~Zahl	0
41	2	2
42	Mittelwert	0

und folgende Codetabelle:

Programmzeile	Code	Symboladresse		
		1.	2.	3.
0	init	0	0	0
1	input	36	39	0

2	input	37	40	0
3	mov	38	36	0
4	add	38	37	0
5	div	38	41	0
6	output	38	42	0
7	exit	0	0	0

Während der Laufzeit werden nun unter Steuerung des RT-Prozessors die folgenden Operationen ausgeführt:

Der **init** macht im Prinzip nichts.

Der **input** von Zeile 1 fragt mit Text Nr. **39** („**Erste Zahl**“) eine Zahl ab und speichert sie auf Symbolwert Nr. **36**.

Der **input** von Zeile 2 fragt mit Text Nr. **40** („**Zweite Zahl**“) eine Zahl ab und speichert sie auf Symbolwert Nr. **37**.

Der **mov** schafft Symbolwert Nr. **36** nach Symbolwert Nr. **38**.

Der **add** addiert Symbolwert Nr. **37** auf Symbolwert in Nr. **38**.

Der **div** teilt Symbolwert Nr. **38** nun durch Symbolwert Nr. **41** (Dies ist immer eine **2**). Damit steht in Symbol Nr. **38** der fertig berechnete Mittelwert.

Der **output** gibt nun Symbolwert Nr. **38** mit Text Nr. **42** („**Mittelwert**“) aus.

Der **exit** beendet das Programm.

Das ist alles. Ist Assembler nicht etwas wunderbar Einfaches?

8.2 Der Zeichensatz von RT

Der RT-Zeichensatz enthält alle 8-Bit-Zeichen von 32 bis 127 und 160 bis 255. Dies ist zugleich der 8-Bit-Unicode-Zeichensatz. Feste Bedeutungen haben folgende Sonderzeichen:

+ - . E e	Vorzeichen, Dezimalpunkt und Exponentzeichen in Zahlen
~	Leerzeichen-Stellvertreter in Zeichenketten
\	Zeilenumbruch-Stellvertreter in Zeichenketten
()	Klammern der Feldindizes
.	Zeichen in Leersymbol „.“ und Befehlszeiger „.“
° (Zeichen für Grad- und Bogenmaß in den Symbolen „°(“ und „°(“. (Code ° = 176)
/	Divisionszeichen in den Symbolen „pi/2“ und „pi/4“
®	Zeichen „Erde“ in „®“ und „®f“ (betrifft nur Vimage-Implementation/Code = 174)
() \$ _	In Dateinamen zulässige Sonderzeichen
:	Abschließendes Kennzeichen der Markentoken
 	Führendes Kennzeichen der Pseudobefehle
;	Kommentar-Kennzeichen
¶	Zeilenfortsetzungs-Zeichen. (Zeichencode 0182)

8.3 RT und andere Assembler

Was ist in RT „typisch Assembler“ und was unterscheidet RT von anderen Assemblersprachen?
Vor allem eines: RT ist Minimalist.

• **Die „Befehls-Operationsrichtung“:** kann von Assembler zu Assembler wechseln und ist oft heiß umstrittene Glaubensfrage. RT operiert wie Intel- oder AMD-Assembler: `mov a 10`. Bei AT&T und DEC-Assemblern würde man eine 10 per `mov 10 a` nach a befördern. Grundregel ist, dass ein Operand, der geschrieben und somit geändert wird, immer der 1. Operand sein muss. Der 2. und 3. Operand wird immer nur gelesen.

• **Variablen deklarieren oder nicht:** Einen ebensolchen Glaubenskrieg gibt im Bereich höherer Programmiersprachen zwischen selbstdeklarierenden Sprachen und Sprachen, deren Variablen zu deklarieren sind. Assembler machen grundsätzlich nichts von allein. Folglich muss der Programmierer den Variablen Speicherplatz persönlich zuteilen, also – deklarieren. — RT hingegen ähnelt hier Fortran, Basic, Python oder Ruby und trägt – assembleruntypisch – alles, was es nicht kennt, automatisch in die Symboltabelle ein.

• **Bezeichner:** RT kennt keine Einschränkungen bei der Vergabe von Bezeichnern. Außer gewohnten Symbolnamen wie `m12`, `pp` oder `alpha` auch `a'`, `b2`, `M<22>`, ja sogar `,??;½¶@`, `!?!<` oder `,`, ausdrücklich zulässig. Alle Zeichencodes 33 ... 126 und 160 ... 255 können vergeben werden. (Ausgenommen ist allein das Semikolon, das Kommentar einleitet und das Paragraph-Zeichen am Zeilenende.) — Fast alle anderen Programmiersprachen, Assembler eingeschlossen, beschränken Bezeichner auf Buchstaben und Ziffern, oft auch Unterstrich und Dollarzeichen, mehr aber nicht.

• **Bitte keine Kommas:** Assembler nutzen oft das Komma als Tokentrenner. — Das klappt in RT nicht: Ein `mov eax, 4` setzt nicht etwa „eax“ auf 4, sondern deklariert eine Variable „eax,4“ neu und lädt diese mit dem Wert Null. Eine Variable „eax“ hingegen bleibt unverändert.

• **Register:** Physische CPU, wie X86 oder die AMD-64-Architektur verfügen über eine sehr große Anzahl von Registern mit klangvollen Namen, wie Pointer-, Status-, Stack-, Segment-Register etc. In diesen laufen alle Operationen ab. — RT ist hier viel einfacher strukturiert: Man kann rechnen, ohne erst Hauptspeicherplätze in spezielle Register befördert zu haben. Für den Assembler-Stylisten gibt es allerdings die 8 vordefinierten Register-Variablen „r0“ bis „r7“. Ein Rechenzeitvorteil verbindet sich damit allerdings in RT nicht.

• **Adressierungsmodi:** Die hohe Schule klassischen Assembler-Programmierens ist es, zahlreiche „Adressierungsmodi“, mit klangvollen Namen wie „Direkt“, „Indirekt“, „Absolut mit Offset“, „Autoinkrement“ oder „Indirekt-Autoinkrement“ kunstvoll einzusetzen. — RT kennt derartige Adressierungsmodi nicht. Zu kompliziert. Kenner werden eine ausreichende Andeutung derartiger Funktionalitäten allein in den Befehlen `put` und `get` erkennen.

• **Bedingte Sprünge:** Fast alle Assembler arbeiten bedingten Sprünge zweiteilig ab. Zuerst testet z. B. ein `CMP` („compare“) die Bedingung und setzt gewisse Flagbits. Anschließend erfolgt anhand dieser die bedingte Verzweigung mit einem folgenden Befehlen, wie z. B. `JLE` („jump if less equal“). — Weil RT als Minimal keine Flagbits kennt, gibt es hier die „Doppelbefehle“ des `cmple`-Typs, die beide Teilschritte in einem Befehl abarbeiten.

• **Datentypen** – kennt RT nicht. RT-intern ist ein Symbolwert immer eine 8-Byte-Double-Gleitkommazahl. — Andere Assembler rechnen hingegen zunächst vorzugsweise in Integer. Dieses differenzieren sie in zahlreiche Untervarietäten, wie signed/unsigned, 8 Bit/16 Bit/32 Bit/64 Bit etc.

• **add statt inc?** Der Nicht-Assemblerkundige wird sich natürlich fragen, wozu die Befehle `inc`, `dec` oder `neg` nützlich sind – kann man doch auch mit `add a 1` inkrementieren und mit

`mul z -1` negieren. Klare Antwort: Diese Befehle gehören zu einem Assembler-Befehlssatz einfach dazu. Es wäre verdammt unstylish, eine Programmiersprache Assembler zu nennen, die keinen Inkrementbefehl kennt. Einen Shift-Befehl wird man allerdings in RT vergeblich suchen: RT macht das sonst übliche Durcheinander von logischen und bitweisen Operationen nicht mit.

• **Bei Feldern bitte besonders aufpassen:** Wie alle Assembler bietet auch RT grundsätzlich vollen Zugang zum gesamten Hauptspeicher – sprich zur Symboltabelle. So kann man sich allerdings auch viel kaputt machen. Dies gilt auch für RT. Mit unqualifizierten Schreibzugriffen kann man sich sehr schnell die Symboltabelle zerstören. Hier noch einmal eine kurze Checkliste für den korrekten Umgang mit Feldern:

1. Felder müssen mit `_dim` deklariert werden.
2. Die Feldlänge im `_dim`-Befehl ist als Zahl anzugeben. Variablen gelten als 0.
3. Symbolnamen wie „a(64)“ sollen ausschließlich für Feldelemente genutzt werden.
4. Symbolnamen wie „a(b)“ sollten besser nicht verwendet werden.
5. Beim `put`-Befehl in Felder ist auf korrekten Pointer und Offset achten.
6. Beim `read`-Befehl in Felder sollen niemals zu große Feldlängen angegeben werden.
7. Nach einem `read`-Befehl in Felder ist immer der Feldpointer per `adrof` richtig einzustellen.

• **Höhere Mathematik:** RT kennt höhere Rechenoperationen, wie Winkelfunktionen oder Logarithmen. — Dies ist eine Nutzerfreundlichkeit, über die ältere Assembler niemals verfügen. Es ist direkt ein „klassischer“ Hauptnachteil des Assemblerprogrammierens, dass man z. B. für Winkelfunktionen schlimmstenfalls eine Reihenentwicklungen schreiben muss.

Orthogonalitätspuristen werden bemerken, dass die Befehle `exp`, `exp10` und `exp2` infolge des `expx` überflüssig sind, ähnliches gilt für die Logarithmusfunktionen. Insbesondere könnte `e` als Standardwert des `expx` (`logx`) den Befehlssatz reduzieren. Bitte nicht, denn dann ergäben sich eine numerisch sehr bedenkliche Verschleierung von Exponenten-Nulldurchgängen.

• **Ein-/Ausgabe** erfolgt in Assembler im Allgemeinen über sehr umständliche Betriebssystemrufe. — Hier vereinfachen und erleichtern die Ein-/Ausgabebefehle des RT-Assemblers sehr stark.

• **Keine Literale:** Alle Programmiersprachen und auch Assembler trennen Operanden in Literale (z. B. „2“) und Variablen (z. B. „a“). Literale können nur gelesen werden und niemals Werte zugewiesen bekommen. — RT ist Minimalist und macht hier keinen Unterschied. Es gibt keine Trennung zwischen „rvalues“ und „lvalues“. Alles ist immer „nur“ Symbol. Eine feine Nebenfolge ist freilich: Ein Befehl, wie `add 4 5` ist sprachlich korrekt. Dieser schreibt nun aber auf den Symbolwert des Symbolen mit dem Namen „4“ eine 9! Ein folgender Befehl `mul 10 4` würde aus der 10 eine 90 machen! So sollte man allerdings nicht programmieren!

• **Nur rudimentäre Zeichenketten:** Assembler waren wohl noch nie besonders gute Zeichenketten-Verarbeitungsspezialisten. RT folgt insofern dieser Tradition, als das Zeichenketten in Form der Symbolnamen Konstanten sind und nicht beschrieben werden können. Bei der Konzeption von RT wurde vorrangig auf die Umsetzung mathematischer Formeln Wert gelegt. Dabei werden Zeichenketten eher selten benötigt.

• **Zeichenkettennotation:** Auch will die Zeichenkettennotation mit `~` als Leerzeichen und `\` als Zeilenumbruchzeichen auf RT als Spartaner hinweisen. Mit `abc~def~ghi` bleibt die Tokenstruktur offensichtlicher, als bei der üblicheren Notationsformen a la "abc def ghi". Für den

Schreibaufwand, den Tilden bereiten, wird um Verständnis gebeten. Was bleibt ist ein Gewinn an Ästhetik und Stil.

- **1024 Zeichen:** Die „harte“ Begrenzung der Zeichenkettenlänge (=Symbolnamenlänge) auf 1024 Zeichen, die man grob als „16 Zeilen mit 64 Zeichen“ lesen kann, soll ebenfalls das *less is more-Konzept* von RT unterstreichen. Fasse dich kurz! Wir erinnern damit gleichzeitig an Zeiten in denen einige Dutzend kByte Zeichenkettenspeicher sehr sehr viel waren. (Hintergrund ist: Irgendwann werde ich einmal einen RT-Compiler schreiben. Und der soll nicht unbedingt einen 16-Gigabyte-Zeichenkettenpool mitschleppen.)
- **Kein Maschinencode:** Assembler übersetzt klassischerweise Quellcode in Maschinencode. RT kennt aber keine Maschinenbefehle. Diese sind nicht erforderlich, weil RT keinen Hardware-Prozessorhintergrund hat, sondern auf einer virtuellen Maschine läuft.

Und nun ?

Assembler – so gut wie ausgestorben?

Das stimmt nicht ganz. Jede Prozessorarchitektur, jede CPU hat ihren Assembler. Allerdings werden die Instruction Sets der CPU seit 60 Jahren immer umfassender, komplexer und komplizierter. Die Dokumentation des Befehlssatzes der AMD-64-Architektur umfasst mittlerweile 6 Bände:

Vol. 1. Application Programming	336 Seiten
Vol. 2. System Programming	488 Seiten
Vol. 3. General-Purpose and System Instructions	466 Seiten
Vol. 4. 128-Bit Media Instructions	426 Seiten
Vol. 5. 64-Bit Media and x87 Instructions	360 Seiten

Zusammen (mit Stand 2008) also 2076 Seiten. Befehle, Befehle, Befehle, Befehle, Befehle ...

Assembler, so eine weit verbreitete Annahme, wird nur noch selten benutzt, weil es höhere Programmiersprachen gibt, die leistungsfähiger sind.

Meine eigene Vermutung ist allerdings, dass das Assemblerprogrammieren ein Opfer des instruktionellen Overkill der Prozessorarchitekten geworden ist. CVTTPD2DQ: „Convert Packed Double-Precision Floating-Point to Packed Doubleword Integers, Truncated“: Dann doch lieber zu Java, PHP, C++, C#, Python, Perl, VB.NET oder Brainfuck greifen.

Dennoch gibt es alte Bekannte in dem gigantischen Instruktion Set des AMD-64:

Auf Seite 204 der General-Purpose Instruction Reference des AMD-64 lesen wir:

NOP No Operation

Does nothing. This one-byte instruction increments the rIP to point to next instruction in the instruction stream, but does not affect the machine state in any other way.

Mnemonic	Opcode	Description
NOP	90	Performs no operation.

Den guten alten `nop` gibt es also immer noch!

Der lief 1965 auf der PDP-8, 1980 auf der VAX, 1985 auf dem 80386, 1990 auf dem 80486 und – auf dem Athlon läuft er immer noch.

Ob es in 100 Jahren noch PDF noch gibt, wird zwar von Archivaren und Bibliothekaren gewünscht. Sicher ist das aber nicht. Immerhin ist die Dokumentation über 700 Seiten lang. Damit sind bei künftigen Implementationen vielfältige Inkompatibilitäten vorprogrammiert.

Eine RT-Maschine mit ihren 97 Befehlen hingegen wird man sich auch in 100 Jahren schnell einmal selber schreiben können. Wünschen wir uns, dass es dann noch Computer gibt. Ansonsten bliebe nur – Rechenschieber, Logarithmentafel und das Fahren mit dem Rheingold-Express.