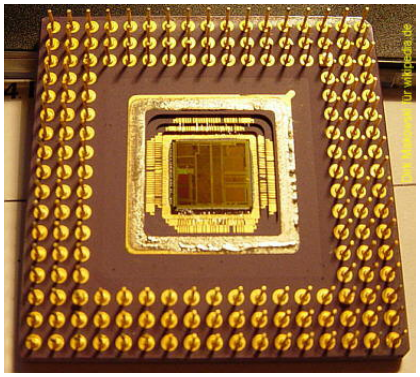
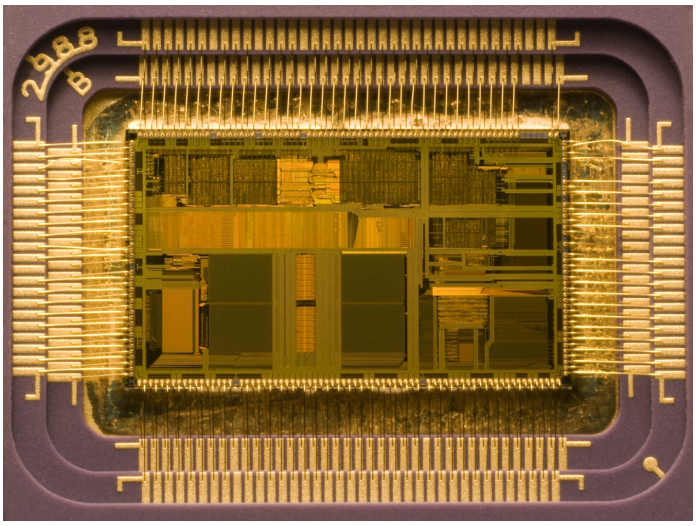
Von-Neumann-Rechner auf dem Bildschirm
CPU-Simulation mit dem Modellrechner DC[1]



Horst Gierhardt
Städtisches Gymnasium Bad Laasphe
www.gierhardt.de



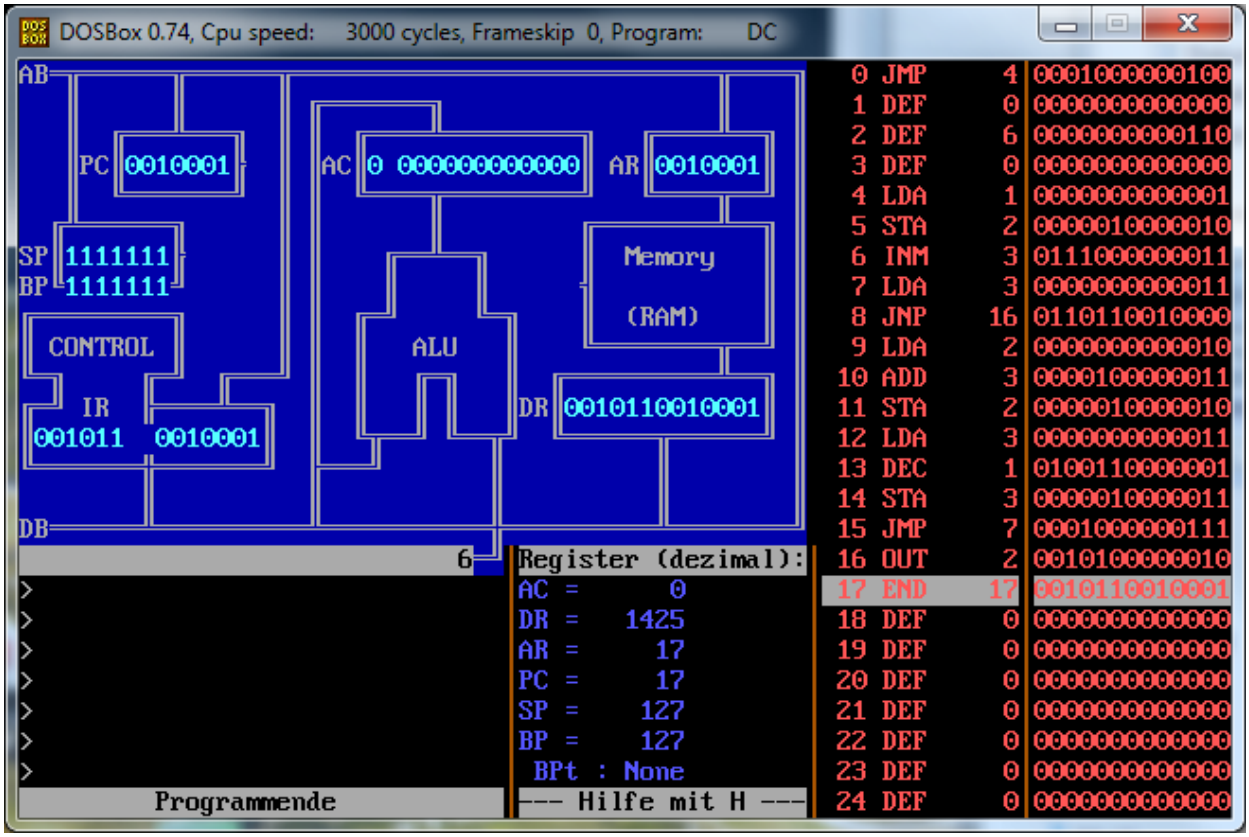
Version vom 11. März 2021

[1]Version 6.6

Inhaltsverzeichnis

John von Neumann (1903–1957) um 1940

1 „Hardware“-Beschreibung



1.1 Zentraleinheit

Das Programm DC simuliert auf dem Bildschirm eine einfache Zentraleinheit (CPU = **c**entral **p**rocessing **u**nit) eines Computers nach dem *Von-Neumann-Prinzip* mit

- Rechenwerk (ALU = **a**rithmetic **l**ogical **u**nit),
- Steuerwerk bzw. Kontrolleinheit,
- Speicher mit Schreib-Lese-Zugriff (RAM = **r**andom **a**ccess **m**emory), und
- Ein-Ausgabe-Einheit (I/O = Input/Output).

Die einzelnen Komponenten werden mit ihrem Inhalt auf dem Bildschirm dargestellt. Während des Programmlaufs sind alle Vorgänge und alle wichtigen Daten je nach Anwenderwunsch mehr oder weniger genau beobachtbar.
Die simulierte Zentraleinheit hat einen Arbeitsspeicher von 128 Wörtern zu je 13 Bit, in dem das Programm und die Daten einschließlich des Stacks untergebracht sind. Durch die Beschränkung der Wortlänge auf 13 Bit sind als Daten nur ganze Zahlen x mit $-4096 \leq x \leq 4095$ zulässig. Eine Speicherstelle kann eine ganze Zahl oder einen Befehl einschließlich einer Speicheradresse enthalten. Mit 6 der 13 Bit lässt sich ein Befehl darstellen, d.h. es sind 64 verschiedene Befehle möglich. Mit den restlichen 7 Bit ist eine Speicheradresse (0 – 127) darstellbar. In diesem Punkt weicht der Simulationsrechner von realen Rechnerarchitekturen ab, die meist den Befehl und den Operand an aufeinanderfolgenden Speicherstellen ablegen.

Diese Abweichung ermöglicht allerdings eine vereinfachte Darstellung der Vorgänge und ist keine Einschränkung in didaktischer Sicht.

1.2 Register

Im Mikroprozessor stehen verschiedene Register zur internen Zwischenspeicherung verschiedener Daten, Ablaufkontrolle und zum Rechnen zur Verfügung:

1.2.1 Befehlsregister (IR = instruction register)

Vor der Ausführung eines Befehls muss das entsprechende Befehlswort in das Befehlsregister geladen werden. Das Befehlswort (13 Bit) wird vom Steuerwerk (CONTROL) in den eigentlichen Befehl und den Operanden zerlegt. Am Bildschirm ist die Aufteilung in Befehlsteil und Adresse zu erkennen. Bei realen Mikroprozessoren wird an dieser Stelle der Operand mit einem weiteren Speicherzugriff geladen. Das Steuerwerk hat intern für jeden Befehl ein kleines Programm gespeichert (Mikrocode). Ein solches Programm wird nach der Decodierung des geladenen, binär codierten Befehls gestartet. Dies kann z.B. das Ablegen des PCs auf dem Stack, das Inkrementieren des PCs u.a. bewirken. Diverse Register werden von hier bedient bzw. aktiviert. Die Synchronisation aller Vorgänge durch einen Taktgenerator wird bei diesem Rechnermodell nicht thematisiert.

1.2.2 Befehlszählregister (PC = program counter)

In diesem Register steht immer die Speicheradresse des nächsten auszuführenden Befehls. Vor jeder Befehlsausführung wird dieses Register um 1 inkrementiert. Bei Sprungbefehlen wird dieses Register mit der Zieladresse, die vom Befehlsregister geliefert wird, geladen. Dieses Register kann im Direktmodus manipuliert werden.

1.2.3 Adressregister (AR)

Bei jedem Schreib- und Lesevorgang im Speicher muss in diesem Register die Adresse der anzusprechenden Speicherstelle stehen. Dadurch wird die entsprechende Speicherstelle zugänglich. Das Adressregister kann nicht direkt manipuliert werden, sondern wird jeweils vom Steuerwerk mit der benötigten Adresse bedient.

1.2.4 Datenregister (DR)

Das Datenregister nimmt den zu schreibenden oder den gelesenen Wert auf. Alle Werte vom und zum Speicher gehen über dieses Register. Es kann wie das Adressregister nicht direkt manipuliert werden.

1.2.5 Akkumulator (AC)

Der Akkumulator ist das zentrale Register im Mikroprozessor. In Verbindung mit der ALU (arithmetic logical unit) ist er für das Rechnen zuständig. Die ALU kann z.B. einen Wert aus einer Speicherstelle zu einem Wert im Akkumulator addieren. Das Ergebnis steht dann im Akkumulator. Im Akkumulator können also zwei Werte durch eine Rechenoperation ‘zusammengeführt’ werden. Die vorliegende ALU kennt allerdings nur Addition, Subtraktion

und Negation als Operationen. Das Inkrementieren und Dekrementieren um 1 kann direkt ohne Zugriff auf den Speicher geschehen. Reale Computer besitzen meist mehrere Register.

1.2.6 Stapelzeiger (SP = stack pointer)

Beim Einschalten des DC zeigt der Stapelzeiger auf das letzte Speicherwort mit Adresse 127, d.h. der Stapelzeiger enthält die Zahl 127. Wird ein Wert auf den Stack gelegt (ge'push't), so wird der Stapelzeiger um 1 erniedrigt. Er zeigt immer auf die erste freie Speicherstelle auf dem Stack. Der Stack wächst also im Speicher von oben nach unten.
Bei einem Prozeduraufruf wird die Rücksprungadresse auf dem Stack gesichert, damit nach Ausführung der Routine bei dem im Speicher dem Aufrufbefehl folgenden Befehl fortgefahren werden kann.
Werte im Akkumulator oder im Speicher können auf den Stack ge'push't und von diesem ge'pop't werden.
Besondere Befehle (siehe Befehlssatz) gestatten den Zugriff auf Adressen relativ zum Stapelzeiger. Damit sind echte lokale Variablen möglich. Rekursion mit einfacher Werteparameterübergabe ist kein Problem. Die Adressierung über den BP (siehe unten) sollte der Adressierung über den SP vorgezogen werden.

1.2.7 Basisregister (BP = base pointer)

Neben dem Stackpointer ist auch das Basisregister für Stackoperationen nutzbar. Es bietet sich z.B. an, wenn in einer Prozedur Zwischenergebnisse auf dem Stack abzulegen sind. Da sich bei jedem PUSH oder POP der Stackpointer ändert, wäre ohne ein zweites Register die Adressierung von lokalen Variablen schwierig bzw. umständlich. Man sollte sich beim Zugriff auf lokale Variablen grundsätzlich dieses Registers bedienen. Werte können zwischen dem SP und dem BP transferiert werden.

1.3 Datenübertragung

Zur Übertragung von Daten sind Leitungen zwischen den einzelnen Komponenten des Mikroprozessors bzw. zwischen Mikroprozessor, Speicher und I/O-Einheit notwendig:

1.3.1 Adressbus

Über den Adressbus wird über das Adressregister im Speicher eine Speicherstelle 'geöffnet'. Im vorliegenden Modell wären also nur 7 Leitungen nötig, da mit 7 Bit 128 Speicheradressen angesprochen werden können. Der Adressbus hat hier also eine Busbreite von 7 Bit.
Typische Größen für den Adressbus sind 8, 20 (beim Intel 8088), 24 (beim Nachfolger Intel 80286), 32 (etwa Intel 80486), 36 (Pentium 2 bis Pentium 4), 40 (Athlon), 44 (Itanium) und 64 Bit.

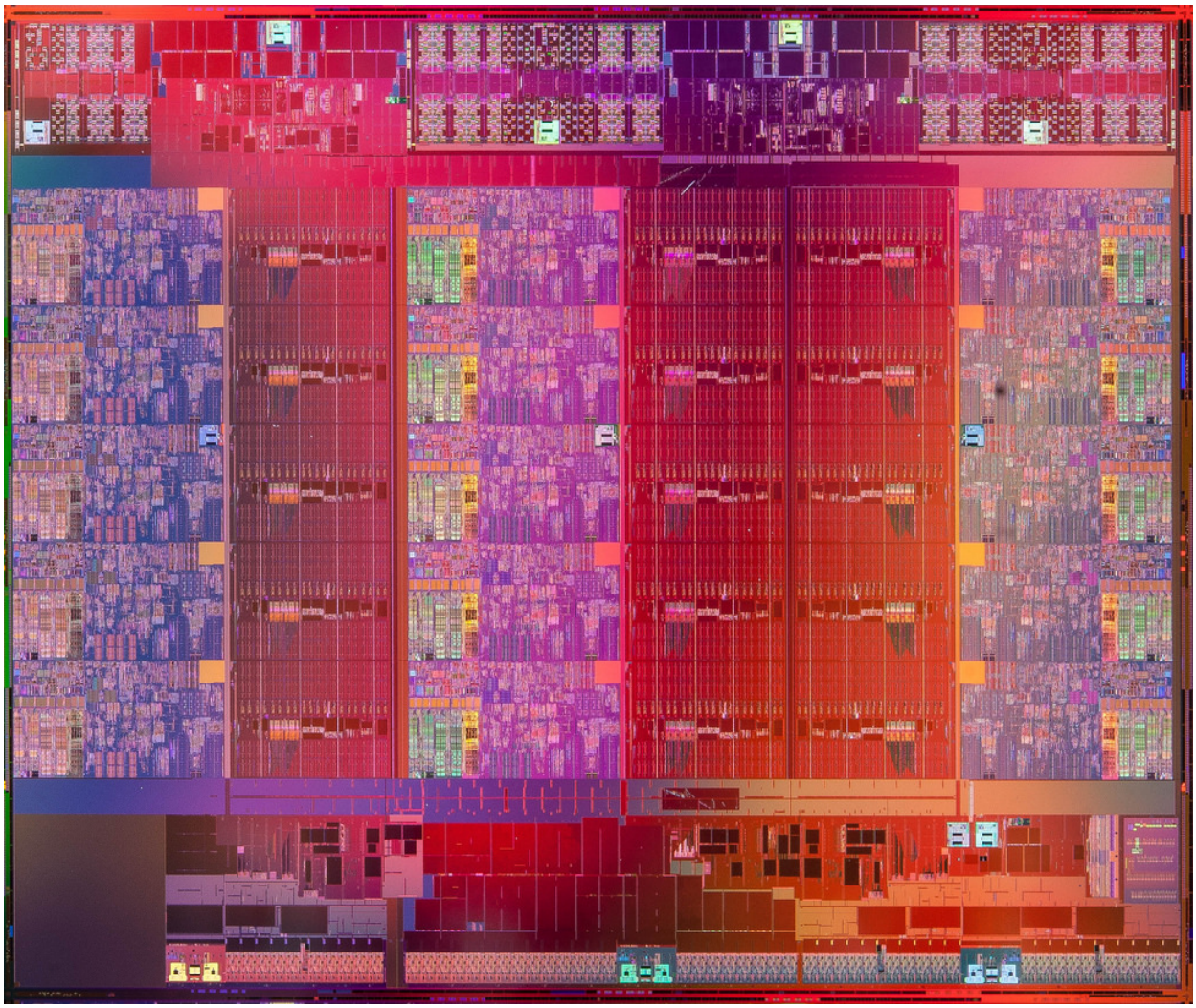
1.3.2 Datenbus

Über den Datenbus wird über das Datenregister ein Wert in eine 'geöffnete' Speicherstelle geschrieben oder daraus gelesen. Im vorliegenden Modell wären 13 Leitungen nötig, da die Speicherworte eine Breite von 13 Bit besitzen. Der Datenbus hat hier also eine Breite von 13 Bit. Typische Größen für den Datenbus sind 8-Bit-, 16-Bit-, 32-Bit- oder 64-Bit.

Über den Datenbus können auch Werte an die Ausgabeeinheit (Output) übergeben oder von der Eingabeeinheit (Input) übernommen werden.

1.3.3 Steuerleitungen

Zwischen Steuerwerk und Speicher liegt zusätzlich die Schreib/Leseleitung. Über sie wird die Information weitergegeben, ob bei einem Speicherzugriff geschrieben oder gelesen werden soll. Im vorliegenden Modell ist diese Leitung nicht dargestellt. Allerdings erscheint neben dem Speicher bei einem Speicherzugriff ein Symbol (‘rd’ oder ‘wr’), das die Art des Zugriffs (‘read’ oder ‘write’) darstellt. Das Steuerwerk (CONTROL) ist intern über Steuerleitungen mit den einzelnen Komponenten der CPU verbunden, um diese aktivieren zu können.



Intel Xeon E7v2-Die

2 Befehlssatz

In den folgenden Tabellen dargestellt ist

- der mnemonische Code (Mnemonic), der durch die Buchstabenfolge meist die Bedeutung des Befehls erkennen lässt,
- die interne Darstellung des Befehls in binärer Codierung. (Diese ist für die Programmierung uninteressant und nur der Vollständigkeit halber aufgeführt.)
- und eine Kurzbeschreibung der Bedeutung des Befehls.

2.1 Grundbefehle

Mnemo	**Intern**	**Bedeutung**
LDA	000000	LOAD INTO ACCUMULATOR — Lade den Wert der angegebenen Speicherstelle in den Akkumulator.
STA	000001	STORE ACCUMULATOR TO MEMORY — Speichere den Akkuinhalt an der angegebenen Speicherstelle ab.
ADD	000010	ADD TO ACCUMULATOR — Addiere den Wert der angegebenen Speicherstelle zum Akkuinhalt.
SUB	000011	SUBTRACT FROM ACCUMULATOR — Subtrahiere den Wert der angegebenen Speicherstelle vom Akkuinhalt.
NEG	010001	NEGATE ACCUMULATOR — Negiere Akkuinhalt.
INC	010010	INCREMENT ACCUMULATOR — Erhöhe Akkuinhalt um 1.
DEC	010011	DECREMENT ACCUMULATOR — Erniedrige Akkuinhalt um 1.
OUT	001010	OUTPUT MEMORY — Gib den Wert der angegebenen Speicherstelle an die Output-Einheit. Die auszugebende Zahl erscheint in einer Zeile oberhalb des Eingabe–Fensters.
INM	011100	INPUT TO MEMORY — Speichere die von der Input–Einheit gelesene Zahl an der angegebenen Adresse ab. Das Programm hält bei diesem Befehl an und wartet auf die Eingabe einer Zahl.
END	001011	ENDE — Programm beenden.
DEF	—	DEFINE word — Mit 34 DEF 3 erhält die Speicherstelle mit der Adresse 34 den Wert 3 zugewiesen. Dies ist keine vom Mikroprozessor ausführbare Anweisung, sondern dient nur der Wertbelegung von Speicherstellen beim Einlesen eines DC-Programmes oder im Direktmodus.

Mnemo	Intern	Bedeutung

2.2 Sprungbefehle

Mnemo	Intern	Bedeutung
JMP	000100	JUMP — Unbedingter Sprung. Springe zur angegeb. Speicherstelle und fahre mit dem dort stehenden Befehl fort.
JMS	000101	JUMP IF MINUS — Springe zur angegeb. Speicherstelle und fahre mit dem dort stehenden Befehl fort, wenn der Akkuinhalt negativ ist. Wenn nicht, dann fahre mit dem nächsten Befehl fort.
JPL	001000	JUMP IF PLUS — Sprung, wenn Akkuinhalt > 0.
JZE	001001	JUMP IF ZERO — Sprung, wenn Akkuinhalt $= 0$.
JNM	011010	JUMP IF NOT MINUS — Sprung, wenn Akkuinhalt ≥ 0.
JNP	011011	JUMP IF NOT PLUS — Sprung, wenn Akkuinhalt ≤ 0.
JNZ	010100	JUMP IF NOT ZERO — Sprung, wenn Akkuinhalt $\neq 0$.
JSR	000110	JUMP TO SUBROUTINE — Springe zum Unterprogramm an der angegebenen Adresse und fahre nach dem Rücksprung mit dem nächsten Befehl fort. Rücksprungadresse wird automatisch auf dem Stack abgelegt.
RTN	000111	RETURN FROM SUBROUTINE — Kehre vom Unterprogramm zurück zum Befehl nach der Aufrufstelle. Rücksprungadresse wird vom Stack geholt.

2.3 Stackoperationen (SP-orientiert)

Mnemo	Intern	Bedeutung
PSH	001100	PUSH ACCUMULATOR TO STACK — Lege den aktuellen Akkuinhalt auf dem Stack ab.
POP	001101	POP FROM STACK TO ACCUMULATOR — Hole Wert vom Stack in den Akkumulator.
PSHM	001110	PUSH MEMORY TO STACK — Lege den Inhalt der angegebenen Speicherstelle auf dem Stack ab.
POPM	001111	POP FROM STACK TO MEMORY — Hole Wert vom Stack und speichere ihn an der angegebenen Speicherstelle ab.
LDAS	010101	LOAD FROM STACK TO ACCUMULATOR — Lade den Wert an Adresse SP+XXX in den Akkumulator, wobei XXX ≥ 0 als Parameter anzugeben ist.
STAS	010110	STORE ACCUMULATOR TO STACK — Speichere den Akkumulator an Adresse SP+XXX.
ADDS	010111	ADD STACK TO ACCUMULATOR — Addiere den Wert an Adresse SP+XXX zum Akkuinhalt.
SUBS	011000	SUBTRACT STACK FROM ACCUMULATOR — Subtrahiere ...
OUTS	011001	OUT STACK — Ausgabe des Wertes an Adresse SP+XXX.
INS	011101	INPUT TO STACK — Speichere die von der Input–Einheit gelesene Zahl an der Adresse SP+XXX ab.

2.4 Stackoperationen (BP-orientiert)

Mnemo	Intern	Bedeutung
LDAB	011110	LOAD FROM STACK TO ACCUMULATOR — Lade den Wert an Adresse BP+XXX in den Akkumulator, wobei XXX ≥ 0 als Parameter anzugeben ist.
STAB	011111	STORE ACCUMULATOR TO STACK — Speichere den Akkumulator an Adresse BP+XXX.
ADDB	100000	ADD STACK TO ACCUMULATOR — Addiere den Wert an Adresse BP+XXX zum Akkuinhalt.
SUBB	100001	SUBTRACT STACK FROM ACCUMULATOR — Subtrahiere ...

Mnemo	Intern	Bedeutung
OUTB	100010	OUT STACK — Ausgabe des Wertes an Adresse BP+XXX.
INB	100011	INPUT to Stack — Speichere die von der Input–Einheit gelesene Zahl an der Adresse BP+XXX ab.
SPBP	100111	TRANSFER SP TO BP — Schreibe den Inhalt von SP in das Register BP.
BPSP	100110	TRANSFER BP TO SP — Schreibe den Inhalt von BP in das Register SP.
POPB	100100	POP BP — Hole Wert vom Stack in BP.
PSHB	100101	PUSH BP — Lege Inhalt von BP auf dem Stack ab.

3 Syntax der Befehle

Die meisten Befehle beziehen sich auf eine Speicheradresse. Entsprechend müssen sie diese als Parameter enthalten (z.B. LDA 02). Die Befehle NOP, END, RTN, PSH, POP, NEG, INC, DEC werden ohne Parameter geschrieben. Auf dem Bildschirm von DC kann zwar ein Parameter (z.B. 0) angezeigt werden, dieser hat aber bei den letztgenannten Befehlen keine Bedeutung.

4 Programmsteuerung

Eingabe	**Merkform**	**Bedeutung**
H	Help	Hilfetext anzeigen.
C	Clear	Alles löschen (DC-Reset). Der Speicherinhalt wird gelöscht, und alle Register werden auf ihren Startwert gesetzt.
W	Wait	Nach jeder Phase der Befehlsausführung auf Taste warten.
D	Delay	Nur eine kleine Pause nach jeder Phase.
N	No Wait	Keine Pausen, nicht auf Taste warten. Einzelschrittmodus ausschalten
R	Run	Programmausführung starten.
G x	Go x	Programm ab Adresse x ausführen.
CR	RETURN	Einzelausführung (CR = carriage return). Ein Befehl wird komplett ausgeführt.
ESC	ESCAPE	Programm anhalten.
V x	View x	Speicherseite mit Adresse x anzeigen. Damit kann z.B. der Stack kontrolliert werden.
ED	Edit	Editor (intern) aufrufen.
ASS	Assemble	Mini-Assembler aufrufen. Nach dem Aufruf ist zuerst eine Datei (mit Extension DCL) menügeführt auszuwählen. Der Mini-Assembler fragt dann nach, ob die evtl. bestehende Datei mit Extension DC überschrieben werden kann. Bei Verneinung stoppt der Assembler.
Q	Quit	DC verlassen.
L name	Load name	DC-Programm aus Datei 'NAME.DC' laden. Die Extension DC wird automatisch angefügt. Beim Einlesen wird der Programmtext in das interne DC-Format der Befehle konvertiert. Kommentare werden abgeschnitten. Wird kein Dateiname angegeben, erscheint ein Menü zur Auswahl der Datei. Verzeichnisse und Laufwerke können gewechselt werden.

Eingabe	Merkform	Bedeutung
B x	Break on	Breakpoint an Adresse x setzen. Das Programm hält an der angegebenen Adresse vor Ausführung des dortigen Befehls an. Nach Inspizierung der Registerinhalte kann das Programm mit R (Run) fortgesetzt werden.
B	Break off	Gesetzten Breakpoint löschen.
Adr. Code Argument		Befehl direkt in Speicher laden. Sinnvoll z.B. für das Testen von einzelnen Befehlen im Direktmodus.
Adr. DEF Konstante		Konstante direkt in den Speicher laden. Sinnvoll zum Verändern von Werten im Direktmodus, wenn man die Eingabemöglichkeit über die Input-Einheit nicht nutzt.
PC Adresse		PC-Register mit Adresse laden. Das PC-Register kann z.B. nach einem Programmlauf auf 0 zurückgesetzt werden, um einen nochmaligen Programmlauf zu ermöglichen.

Diese Befehle sind im DC-Eingabefenster links unten ohne Beachtung von Groß- und Kleinschreibung hinter dem '>'-Zeichen mit abschließendem RETURN einzugeben (Ausnahmen: CR und ESC).

5 Assembler

Benutzt man den Mini-Assembler, so kann man mit symbolischen Adressen (Labels) arbeiten. Die symbolischen Adressen können für Sprungziele, aber auch als Bezeichner für Variablen benutzt werden. Der Doppelpunkt hinter den Labels ist nicht erforderlich, wird aber unterstützt.
Der Mini-Assembler setzt die Labels in Speicheradressen um. Adress-Konstanten deklariert man am besten mit dem Schlüsselwort EQUAL. Beispiel: Null EQUAL 0.
Die Quelltextdatei mit Labels und EQUAL-Deklarationen versieht man mit der Dateiextension DCL (Default-Extension im Editor). Der Assembler übersetzt den Quelltext und schreibt die für den DC lesbare Form in eine Datei mit der Extension DC. Der Mini-Assembler kann von der DC-Ebene mit der Eingabe von ASS und der nachfolgenden Dateiauswahl gestartet werden. Praktischer ist allerdings der Aufruf von der Editor-Ebene, da man dann bei Fehlermeldungen sofort die entsprechende Zeile verbessern kann.

6 Beispiele

6.1 Beispiel ohne Assembler-Benutzung

Im folgenden Beispiel werden absolute Adressen benutzt. Es kann als Einstieg in die Arbeit mit dem DC dienen. Die Benutzung von absoluten Adressen hat aber den großen Nachteil, daß das Einfügen einer zusätzlichen Programmzeile die Änderung von mehreren Adressen zur Folge hat. Der Mini-Assembler löst dieses Problem.

```
; PROGRAMM Einfach.dc
; Alles hinter einem Semikolon wird als Kommentar gelesen.
; Die erste Zahl gibt die Adresse an, an der der Befehl stehen soll.
00 JMP 10   ; Zum Programmanfang an Adresse 10 springen.
            ; Datenbereich
01 DEF -3   ; int A = -3;
02 DEF 7    ; int B =  7;
03 DEF 00   ; int C =  0;
            ;
10 LDA 01   ; Lade den Inhalt von Adresse 01 in den Akku.
11 INC      ; Inkrementiere den Akkuinhalt um 1.
12 ADD 02   ; Addiere zum Akkuinhalt den Inhalt von Adresse 02.
13 STA 03   ; Speichere den Akkkuinhalt an Adresse 03.
14 OUT 03   ; uebergib den Inhalt von Adresse 03 an Outputeinheit.
15 END      ; Programmende.
```

6.2 Beispiel mit Assembler-Benutzung

Das Beispielprogramm von oben nimmt bei Benutzung des Mini-Assemblers die folgende Form an. Zusätzlich wird hier noch die Eingabe über die Input-Einheit demonstriert.

```
; PROGRAMM Einfach.dcl (* Bitte Endung DCL beachten ! *)
JMP Anfang
A:      DEF 000   ; int  A;
B:      DEF 000   ; int  B;
C:      DEF 000   ; int  C;
Anfang:           ; void action()
        INM A     ; {  A = In.readInt();
        INM B     ;    B = In.readInt();
        LDA A
        INC
        ADD B
        STA C     ;    C = (A+1) + B;
        OUT C     ;    Out.print(C);
        END       ; }
```

6.3 Beispiel mit Stackadressierung ohne Parameter

Das folgende Programm gibt die eingegebenen Zahlen in umgekehrter Reihenfolge wieder aus. Es zeigt die recht einfache Programmierung von Rekursionen mit dem DC. Die lokale Variable 'Zahl' wird hier über das Basisregister BP angesprochen. Eine Adressierung über den Stackpointer wäre ebenso möglich, ist aber eine etwas 'unsaubere' Programmiertechnik.

```
; PROGRAMM Umkehr.dcl
                          ; void action() // Hauptprogramm
        JSR Umkehr        ; {   umkehr;
        END               ; }

Umkehr                    ; void umkehr()
                          ; {  Ruecksprungadresse  BP+2
                          ;    int zahl;           BP+1
    Zahl EQUAL 1          ;                        BP = SP
        PSH               ;    //Beginn (Platz fuer Zahl schaffen)
        SPBP              ;    SP -> BP
        INB Zahl          ;    zahl = In.readInt();
        OUTB Zahl         ;    Out.print(zahl);
        LDAB Zahl         ;
        JZE EndIf         ;    if (zahl <> 0)
                          ;        {
        JSR Umkehr        ;        Umkehr rekursiv aufrufen.
        SPBP              ;        SP -> BP
        OUTB Zahl         ;        Out.print(zahl);
                          ;        }
EndIf
        POP               ;    // Platz fuer Zahl freigeben
        RTN               ; } // Ende von umkehr
```

6.4 Beispiel mit Rekursion und Parameterübergabe

Das folgende Beispiel zeigt die Möglichkeit zur Rekursion an einem etwas komplexeren Beispiel. Es demonstriert die Parameterübergabe an eine Methode mit PSH und den Zugriff auf die Parameter (lokale Variablen) über das BP-Register. Zu beachten ist immer die Position der Rücksprungadresse auf dem Stack. Es muss gewährleitet sein, daß bei einem RTN die richtige Adresse auf dem Stack verfügbar ist. Beim Eintritt in eine Methode stehen auf dem Stack von oben nach unten betrachtet zuerst die vorher gepushten Werte und als letzter Wert die mit JSR automatisch gesicherte Rücksprungadresse. Am Methodenende muss man meistens — wie auch hier demonstriert — die lokalen Variablen 'vernichten' und die Rücksprungadresse an die richtige Position setzen.

```
; ********************************************************
; ********** Tuerme von Hanoi fuer DC ********************
; ********************************************************
; Bei Anfangshoehe > 3 sollte bei OUTB Ziel ein
; Breakpoint gesetzt werden, um die Ausgabe verfolgen zu
; koennen. Erst ab ca. 16-17 Scheiben laeuft der Stack in den
; Codebereich.
; Historischer Vergleichswert: Fuer 6 Scheiben
; benoetigt ein 8-MHz-AT etwa 20s. Das ist lange her!
; Version fuer DC mit BasePointer BP und Mini-Assembler
```

```
; PROGRAMM Tuerme_von_Hanoi_rekursiv_mit_DC;

          JMP  Anfang
GlobalStart   DEF 01          ; final int GlobalStart  = 1;
GlobalAblage  DEF 02          ; final int GlobalAblage = 2;
GlobalZiel    DEF 03          ; final int GlobalZiel   = 3;
GlobalHoehe   DEF 00          ; int GlobalHoehe;
                              ; public void action()
                              ; { // Beginn des Hauptprogramms
Anfang
          INM  GlobalHoehe    ;   GlobalHoehe = In.readInt();
                              ;   (* Parameter auf Stack legen *)
          PSHM GlobalHoehe    ;   PUSH GlobalHoehe
          PSHM GlobalStart    ;   PUSH GlobalStart
          PSHM GlobalZiel     ;   PUSH GlobalZiel
          PSHM GlobalAblage   ;   PUSH GlobalAblage
                              ;   (* und Aufruf der Rekursion *)
          JSR  Transport      ;   Transportiere (GlobalHoehe, GlobalStart,
                              ;                      GlobalZiel, GlobalAblage)
                              ;    (* Von 1 nach 3 ueber 2 *)
          END                 ; } // Ende des Hauptprogramms

Transport
                              ; void Transportiere (Hoehe, Start,
                              ;                      Ziel, Ablage);
                              ; {
        Hoehe   EQUAL 5       ; BP + 5
        Start   EQUAL 4       ; BP + 4
        Ziel    EQUAL 3       ; BP + 3
        Ablage  EQUAL 2       ; BP + 2
        RSpAdr  EQUAL 1       ; BP + 1
          SPBP                ; SP --> BP. SP nach BP, da SP sich bei
                              ; den nachfolgenden PSH-Anweisungen ver-
                              ; veraendert. So bleibt Zugriff auf
                              ; die Parameter auf dem Stack gewaehrleistet.
          LDAB Hoehe
          DEC
                              ;   if (Hoehe-1) > 0
                              ;       {
          JNP  EndIf1         ;       (* Parameter auf Stack legen *)
          PSH                 ;       PUSH Hoehe-1
          LDAB Start
          PSH                 ;       PUSH Start
          LDAB Ablage
          PSH                 ;       PUSH Ablage
          LDAB Ziel
          PSH                 ;       PUSH Ziel
          JSR  Transport      ;       Transportiere (Hoehe-1, Start,
                              ;                      Ablage, Ziel);
                              ;       }
EndIf1    SPBP                ; SP --> BP (* neu setzen, da durch  *)
                              ;           (* Rekursion veraendert. *)
          OUTB Start          ;   Out.print('Lege Scheibe von Turm <Start> ');
          OUTB Ziel           ;   Out.print('                auf Turm <Ziel>.');

          LDAB Hoehe
```

```
            DEC
                                ;   if (Hoehe-1) > 0
                                ;       {
            JNP   EndIf2        ;       (* Parameter auf Stack legen *)
            PSH                 ;       PUSH Hoehe-1
            LDAB  Ablage
            PSH                 ;       PUSH Ablage
            LDAB  Ziel
            PSH                 ;       PUSH Ziel
            LDAB  Start
            PSH                 ;       PUSH Start
            JSR   Transport     ;       Transportiere (Hoehe-1, Ablage,
                                ;                      Ziel, Start);
                                ;       }
                                ;   Stack am Prozedurende bis auf
                                ;   Ruecksprungadresse vernichten
EndIf2      SPBP                ;   SP --> BP (* siehe oben *)
            LDAB  RSpAdr        ;   Ruecksprungadresse an die Spitze
            STAB  Hoehe
            POP                 ;   vier lokale Variablen vernichten
            POP
            POP
            POP
            RTN                 ; } // Ende der Methode Transport
```

7 Unterrichtseinsatz

7.1 Idee zur Einführung in den Themenkreis

Als Einstieg in die Thematik „Rechneraufbau“ hat sich in meinem Unterricht die Simulation des Geschehens in einer CPU durch ein Rollenspiel bewährt. Die Schüler erhalten Angaben zu ihren Rollen mit der Aufgabe, ein „Programm“ ablaufen zu lassen.
Die Rollenverteilung für die CPU-Simulation ist in der Tabelle auf der folgenden Seite dargestellt. In dieser Form kann sie den Schülern als Arbeitsblatt vorgelegt werden.

Der Rechnerspeicher wird simuliert durch acht nummerierte geschlossene Schachteln. In den ‘Speicherzellen’ 1 - 5 befinden sich fünf Maschinenbefehle. In den Schachteln 6 - 8 können Zahlen gespeichert werden. Ein Speicherinhalt kann nur bei geöffneter Schachtel gelesen werden. Ein Wert darf nur in eine geöffnete Schachtel geschrieben werden. Die Rollenverteilung:

Adressierer : Kann auf Anweisung des CPU-Leiters eine Schachtel öffnen. Er kann den Inhalt nicht lesen. Die Nummer der zu öffnenden Schachtel schreibt ihm der CPU-Leiter oder der Schachtelzähler auf einen Zettel, den er immer bei sich trägt.

Datenbote : Kann eine Zahl auf einem Zettel zu einer geöffneten Schachtel bringen und die Zahl hineinschreiben oder eine Zahl aus einer geöffneten Schachtel lesen und weitertransportieren. Nach jedem Schreib/Lesevorgang ist die Schachtel zu schließen. Kann auch eine Zahl zum Bildschirm transportieren.

Bildschirm : Darf einen beschriebenen Zettel für alle Schüler sichtbar hochhalten.

Schachtelzähler : Merkt sich auf einem Zettel die Nummer der Schachtel mit dem nächsten Befehl. Kann vom CPU-Leiter Schachtelnummern übernehmen oder auf Anweisung seine Nummer um 1 erhöhen. Bei Programmbeginn steht auf seinem Zettel eine 1.

Rechenknecht : Kann vom Datenboten Zahlen auf einen Zettel übernehmen, dem Datenboten Zahlen übergeben oder eine Zahl zu der Zahl auf dem Zettel addieren.

CPU-Leiter : Gibt allen anderen die nötigen Anweisungen. Kann Befehle, die ihm der Datenbote aus den Schachteln besorgt, durch Nachsehen in der unten angegebenen Tabelle in die entsprechenden Anweisungen für seine Untergebenen übersetzen. Die Befehle in den Schachteln bestehen aus zweiziffrigen Zahlen. Die erste Ziffer ist als Befehlsnummer, die zweite als Adresse zu interpretieren. Er kennt nur fünf Befehle mit den Nummern 1 – 5 :

1. Lade den Inhalt der Schachtel mit der zweiten Ziffer als Nummer und übergib ihn dem Rechenknecht.
2. Speichere den Inhalt des Rechenknechtzettels in der Schachtel mit der zweiten Ziffer als Nummer.
3. Addiere den Inhalt der Schachtel mit der zweiten Ziffer als Nummer zum Inhalt des Rechenknechtzettels.
4. Übergib den Inhalt der Schachtel (2. Ziffer) dem Bildschirm.
5. Beende das Programm.

Die nummerierten Schachteln (Programm A):

Nr. der Schachtel	Inhalt der Schachtel	Bedeutung
Nr. 1	16	
Nr. 2	37	
Nr. 3	28	
Nr. 4	48	
Nr. 5	50	
Nr. 6	23	
Nr. 7	16	
Nr. 8	00	

Aufgabe 1: Simuliere den Programmablauf mit einem Rollenspiel.
Aufgabe 2: Was bewirkt das Programm in den folgenden Schachteln?
Die nummerierten Schachteln (Programm B):

Nr. der Schachtel	Inhalt der Schachtel	Bedeutung
Nr. 1	17	
Nr. 2	37	
Nr. 3	38	
Nr. 4	29	
Nr. 5	49	
Nr. 6	50	
Nr. 7	34	
Nr. 8	21	
Nr. 9	00	

7.1.1 Der Ablauf des Rollenspiels

Die nummerierten Schachteln (Programm A):

Nr. der Schachtel	Inhalt der Schachtel	Bedeutung
Nr. 1	16	Hole den Inhalt von Schachtel Nr. 6 und übergib ihn dem Rechenknecht.
Nr. 2	37	Hole den Inhalt von Schachtel Nr. 7 und übergib ihn dem Rechenknecht zum Addieren.
Nr. 3	28	Speichere den Inhalt des Rechenknechtzettels in Schachtel Nr. 8.
Nr. 4	48	Hole den Inhalt von Schachtel Nr. 8 und übergib ihn dem Bildschirm.
Nr. 5	50	Beende das Programm.
Nr. 6	23	
Nr. 7	16	
Nr. 8	00	

Die nummerierten Schachteln (Programm B:

Nr. der Schachtel	Inhalt der Schachtel	Bedeutung
Nr. 1	17	Hole den Inhalt von Schachtel Nr. 7 und übergib ihn dem Rechenknecht.
Nr. 2	37	Hole den Inhalt von Schachtel Nr. 7 und übergib ihn dem Rechenknecht zum Addieren.
Nr. 3	38	Hole den Inhalt von Schachtel Nr. 8 und übergib ihn dem Rechenknecht zum Addieren.
Nr. 4	29	Speichere den Inhalt des Rechenknechtzettels in Schachtel Nr. 9.
Nr. 5	49	Hole den Inhalt von Schachtel Nr. 9 und übergib ihn dem Bildschirm.
Nr. 6	50	Beende das Programm.
Nr. 7	34	
Nr. 8	21	
Nr. 9	00	

Der Ablauf im Detail:

- Start: Schachtelzähler schreibt 1 auf Zettel (bekommt er vom Betriebssystem).
- Schachtel Nr. 1 mit Inhalt 16: Hole den Inhalt von Schachtel Nr. 6 und übergib ihn dem Rechenknecht.

  CPU-Leiter

  - **an Adressierer:** Beim Schachtelzähler Nr. holen und entspr. Schachtel öffnen.
  - **an Datenbote:** Inhalt der offenen Schachtel holen.
  - **an Schachtelzähler:** Zahl um 1 erhöhen.

  CPU-Leiter interpretiert 16.

  CPU-Leiter

  - **an Adressierer:** Schachtel Nr. 6 öffnen.
  - **an Datenbote:** Inhalt der offenen Schachtel zum Rechenknecht bringen.
- Schachtel Nr. 2 mit Inhalt 37: Hole den Inhalt von Schachtel Nr. 7 und übergib ihn dem Rechenknecht zum Addieren.

  CPU-Leiter

  - **an Adressierer:** Beim Schachtelzähler Nr. holen und entspr. Schachtel öffnen.
  - **an Datenbote:** Inhalt der offenen Schachtel holen.
  - **an Schachtelzähler:** Zahl um 1 erhöhen.

  CPU-Leiter interpretiert 37.

  CPU-Leiter

  - **an Adressierer:** Schachtel Nr. 7 öffnen.
  - **an Datenbote:** Inhalt der offenen Schachtel zum Rechenknecht.
  - **an Rechenknecht:** Addieren!.
- Schachtel Nr. 3 mit Inhalt 28: Speichere den Inhalt des Rechenknechtzettels in Schachtel Nr. 8.

  CPU-Leiter

  - **an Adressierer:** Beim Schachtelzähler Nr. holen und entspr. Schachtel öffnen.
  - **an Datenbote:** Inhalt der offenen Schachtel holen.
  - **an Schachtelzähler:** Zahl um 1 erhöhen.

  CPU-Leiter interpretiert 28.

  CPU-Leiter

  - **an Adressierer:** Schachtel Nr. 8 öffnen.
  - **an Datenbote:** Rechenknechtzahl in offene Schachtel schreiben.

- Schachtel Nr. 4 mit Inhalt 48: Hole den Inhalt von Schachtel Nr. 8 und übergib ihn dem Bildschirm.

  CPU-Leiter

  - **an Adressierer:** Beim Schachtelzähler Nr. holen und entspr. Schachtel öffnen.
  - **an Datenbote:** Inhalt der offenen Schachtel holen.
  - **an Schachtelzähler:** Zahl um 1 erhöhen.

  CPU-Leiter interpretiert 48.

  CPU-Leiter

  - **an Adressierer:** Schachtel Nr. 8 öffnen.
  - **an Datenbote:** Inhalt der offenen Schachtel zum Bildschirm bringen.
  - **an Bildschirm:** Zeigen!.

- Schachtel Nr. 5 mit Inhalt 50: Beende das Programm.

  CPU-Leiter

  - **an Adressierer:** Beim Schachtelzähler Nr. holen und entspr. Schachtel öffnen.
  - **an Datenbote:** Inhalt der offenen Schachtel holen.
  - **an Schachtelzähler:** Zahl um 1 erhöhen.

  CPU-Leiter interpretiert 50.

  CPU-Leiter

  - **an Alle:** Ende!

7.1.2 Erste Ergebnisse und Informationen zur Simulation

1. Im *Speicher* befinden sich nebeneinander Befehle und Daten. Eine Unterscheidung ergibt sich erst durch den Programmlauf (Von-Neumann-Prinzip; John von Neumann 1945).

   - Erste Rechenmaschinen wurden von Pascal, Leibniz, u.a. im 17. Jahrhundert entwickelt.
   - Erste Pläne für eine „analytische Maschine“ stammen von Charles Babbage und Ada Lovelace aus dem 19. Jahrhundert.
   - 1941 wurde der erste programmgesteuerte Rechner von Konrad Zuse entwickelt, das Programm befand sich allerdings auf Lochstreifen.

2. Befehle liegen in codierter Form vor, sie müssen also von der Maschine vor ihrer Ausführung *decodiert* werden. Die Befehle bestehen aus einem Anweisungscode und *einer* Speicheradresse bzw. einem Operanden. Man spricht in diesem Zusammenhang von einem *Ein-Adress-Rechner*. Bestimmte Rechnerarchitekturen erlauben auch die Angabe von zwei oder drei Adressen *Zwei- und Drei-Adress-Rechner*.

3. Bei der Abarbeitung der Befehle ergibt sich immer die gleiche Struktur:

   (a) Befehlsholphase

   (b) Befehlsdecodierphase

   (c) Befehlsausführungsphase

4. Jeder Speicherzugriff lässt sich in zwei Phasen gliedern:

   (a) Adressierung („Öffnen“ der Speicherzelle).

   (b) Lesen oder Schreiben.

5. Die Adressierung und das Lesen/Schreiben geschieht in realen Rechnern über elektrische Leitungen (1 Bit — 1 Leitung).

6. Ein digitaler Speicher besteht aus *Speicherelementen*, den kleinsten Funktionseinheiten zum Aufbewahren von Daten. Sie können abhängig von einem äußeren Signal einen von zwei möglichen Zuständen annehmen. Ein Speicherelement speichert 1 *Bit*. Die kleinste adressierbare Einheit eines Speichers heißt *Speicherzelle*. Meist entspricht eine Speicherzelle einem *Byte* = 8 Bit. Die Zusammenfassung mehrerer Speicherzellen (meist 2, 4 oder 8) heißt *Speicherwort*. Bei 16-Bit-Computern besteht ein Speicherwort aus 2 Byte = 16 Bit.

7. Man unterscheidet Speicher mit

   - *wahlfreiem oder direktem Zugriff* (alle Speicherzellen sind mit gleichem zeitlichen Aufwand erreichbar),
   - *zyklischem Zugriff* (Speicherzellen sind nur zeitperiodisch erreichbar),
   - *sequentiellem Zugriff* (Speicherzellen sind nur durch Zugriff auf eine Sequenz von Speicherzellen erreichbar).

- *Schreib-Lese-Zugriff*,
- *Nur-Lese-Zugriff*.

Der Hauptspeicher eines Computers ist immer ein Speicher mit direktem Zugriff, wobei Schreib- und Lesezugriff erlaubt ist (RAM = random access memory). Teile des Betriebssystem befinden sich meist in einem nichtflüchtigen Teil (ROM = read only memory = Nurlesespeicher). Dieser Speicher ist allerdings ebenfalls im direkten Zugriff. Festplatten und Disketten sind Speicher mit zyklischem Schreib-Lese-Zugriff. Magnetbänder sind Speicher mit sequentiellem Zugriff. Bei CDs gibt es verschiedene Arten. Allen gemeinsam ist der zyklische Zugriff. Manche lassen sich allerdings nur einmal beschreiben, aber beliebig oft lesen (WORM = write once read multiple).

7.2 Der DC im Unterricht

Nach der Simulation mit dem Rollenspiel ist eine Einführung in die Arbeitsweise des DC kein Problem. Das oben vorgestellte Beispielprogramm kann als Einstiegsbeispiel dienen. Für einfache Probleme lassen sich recht schnell entsprechende DC-Programme schreiben. Anhand dieser Beispiele werden die internen Vorgänge bei jedem Befehl ausführlich studiert. Einfache Prozeduren sollten möglichst frühzeitig benutzt werden, um mit der Stackstruktur vertraut zu werden. Der Mini-Assembler sollte erst dann eingeführt werden, wenn sich eine gewisse Sicherheit beim Codieren eingestellt hat und das nachträgliche Verändern von Adressen als mühsam und umständlich angesehen wird.
Die internen Vorgänge im Simulationsrechner treten nach einiger Zeit immer mehr in den Hintergrund, wogegen immer mehr Gewicht auf die algorithmischen Probleme gelegt wird. Für die in PASCAL vorhanden Kontrollstrukturen sind Übersetzungsschablonen für die Übertragung in das DCL-Format zu entwickeln.
Die Parameterübergabe an Prozeduren sollte zumindest mit einfachen Beispielen thematisiert werden. Einfache Funktionen können die Funktionswertrückgabe über den Stack demonstrieren.
Ein tieferes Verständnis der überaus wichtigen Stackstruktur ist nur bei Behandlung von Rekursionen möglich. Mit dem DC kann man einen STACK-OVERFLOW *sehen!* Nach meiner Erfahrung wird die Rekursion erst durch das Studium des Stackauf- und -abbaus richtig verstanden. Abstrakte Begriffe wie „Inkarnation einer Prozedur“ helfen beim Verständnis von Rekursionen *vor* der Behandlung der rechnerinternen Vorgänge nicht weiter.

7.3 Java-Kontrollstrukturen im DC-Code

7.3.1 Aufgaben

1. Die Zahlen von 1 bis 10 sollen

   (a) absteigend mit einer `do while`-Schleife

   (b) aufsteigend mit einer `do while`-Schleife

   (c) absteigend mit einer `while`-Schleife

   (d) aufsteigend mit einer `while`-Schleife

   ausgegeben werden. Es ist dabei genau zu beachten, ob ein Jump-Befehl am Anfang oder am Ende der Schleife erfolgt.

2. Die Zahlen von 1 bis zu einer einzulesenden Zahl sollen aufaddiert und die Summe ausgegeben werden.

3. Die Struktur von `if else` ist in möglichst allgemeiner Form in eine Kombination von Jump-Befehlen zu übersetzen.

4. Eine Zahl wird eingelesen und festgestellt, ob sie gerade oder ungerade ist. Bei einer geraden Zahl soll 0, sonst 1 ausgegeben werden. Die Ermittlung des Restes bei der Division durch 2 soll in einem Unterprogramm gemacht werden.

5. Zwei Zahlen werden eingelesen und die größere der beiden Zahlen wird ausgegeben.

6. Ein Programm soll zwei positive Zahlen einlesen, in einem Unterprogramm `mult` das Produkt berechnen und das Ergebnis ausgeben. Zuerst ist dazu ein entsprechendes Java-Programm zu schreiben, wobei darin Multiplikation nicht erlaubt ist. Dieses Java-Programm ist dann in ein Maschinen-Programm „zu Fuß zu kompilieren“.

   Schwierigere Variante: Die beiden Zahlen können negativ oder nicht-negativ sein.

7. Die Multiplikation kann auch rekursiv über Additionen erfolgen. Schreibe dazu ein Java-Programm ohne Parameterübergabe und entwirf mit Hilfe dieses Programmes ein DCL-Maschinen-Programm.

   Warum kann diese rekursive Variante der Multiplikation nicht alle im DC erlaubten Faktoren verarbeiten?

   Tipp: Mit `v Speicheradresse` (v steht für *view*) kann man sich den Stack im DC ansehen.

7.3.2 Lösungen

1. Die Zahlen von 1 bis 10 sollen ausgegeben werden

   (a) absteigend mit einer `do while`-Schleife

```
        JMP Anfang
k       DEF 0
zehn    DEF 10
Anfang
        LDA zehn
        STA k               ; k = 10
Schleife                    ; do {
        LDA k
        OUT k               ;       print (k)
        DEC
        STA k               ;       k = k-1
        JPL Schleife        ; } while (k > 0)
        END
```

   (b) aufsteigend mit einer `do while`-Schleife

```
        JMP Anfang
k       DEF 00
eins    DEF 01
zehn    DEF 10
Anfang
        LDA eins
        STA k               ; k = 1
Schleife                    ; do {
        LDA k
        OUT k               ;       print (k)
        INC
        STA k               ;       k = k+1
        SUB zehn
        JNP Schleife        ; } while (k - 10 <= 0)
        END
```

   (c) absteigend mit einer `while`-Schleife

```
        JMP Anfang
k       DEF 0
zehn    DEF 10
Anfang
        LDA zehn
        STA k                   ; k = 10
SchleifenAnfang                 ; while (k > 0)
        JNP SchleifenEnde       ; {
        LDA k
        OUT k                   ;       print (k)
        DEC
        STA k                   ;       k = k-1
        JMP SchleifenAnfang     ;
SchleifenEnde                   ; }
        END
```

(d) aufsteigend mit einer `while`-Schleife

```
        JMP Anfang
k       DEF 0
eins    DEF 1
zehn    DEF 10
Anfang
         LDA eins
         STA k                ; k = 1
SchleifenAnfang               ; while (k-10 <= 0)
         SUB zehn
         JPL SchleifenEnde    ; {
         LDA k
         OUT k                ;       print (k)
         INC
         STA k                ;       k = k+1
         JMP SchleifenAnfang  ;
SchleifenEnde                 ; }
         END
```

2. Die Zahlen von 1 bis zu einer einzulesenden Zahl sollen aufaddiert und die Summe ausgegeben werden.

```
        JMP Anfang
null    DEF 000
summe   DEF 000
k       DEF 0
Anfang
         LDA null             ;
         STA summe            ;
         INM k                ; k = In.readInt();
WhileAnfang
         LDA k                ;
         JNP WhileEnde        ; while (k > 0)
         LDA summe            ; {
         ADD k                ;
         STA summe            ;       summe = summe + k
         LDA k                ;
         DEC                  ;
         STA k                ;       k = k - 1
         JMP WhileAnfang      ;
WhileEnde                     ; } //end while
         OUT summe
         END
```

```
        JMP Anfang
null    DEF 000
summe   DEF 000
k       DEF 0
Anfang
         INM k                ; void main() {
         JSR Addiere          ;     addiere()
         OUT summe            ;     print(summe)
         END                  ; }
addiere                       ; void addiere()
         LDA null             ; {
```

```
          STA summe          ;
WhileAnfang
          LDA k              ;
          JNP WhileEnde      ; while (k > 0)
          LDA summe          ;   {
          ADD k              ;
          STA summe          ;       summe = summe + k
          LDA k              ;
          DEC                ;
          STA k              ;       k = k - 1
          JMP WhileAnfang    ;
WhileEnde                    ;   } end while
          RTN                ; } end addiere
```

3. Die Struktur von `if else` ist in möglichst allgemeiner Form in eine Kombination von Jump-Befehlen zu übersetzen.

```
      LDA x          ; if (x > 0)
      JNP else       ;      {
      OUT x          ;        print(x)
      JMP endif      ;      }
else                 ; else {
      INC            ;
      STA x          ;        x++
      OUT x          ;        print(x)
endif                ;      }
      ...            ;
```

4. Eine Zahl wird eingelesen und festgestellt, ob sie gerade oder ungerade ist. Bei einer geraden Zahl soll 0, sonst 1 ausgegeben werden. Die Ermittlung des Restes bei der Division durch 2 soll in einem Unterprogramm gemacht werden.

```
     JMP Anfang
zahl DEF 0
zwei DEF 2
rest DEF 0

Anfang
    INM zahl
    JSR Rest_bilden
    OUT rest
    END

Rest_bilden
    LDA zahl
    STA rest
Schleife
    SUB zwei
    STA rest
    JPL Schleife    ; solange Rest > 0
                    ; jetzt ist Rest 0 oder -1
    NEG             ; jetzt ist Rest 0 oder 1
    STA rest
    RTN
```

5. Zwei Zahlen werden eingelesen und die größere der beiden Zahlen wird ausgegeben.

```
        JMP Anfang
X DEF 0
Y DEF 0

Anfang
        INM X
        INM Y

        LDA X
        SUB Y     ; if (x - y > 0)
        JNP else
        OUT X     ;       print(x)
        JMP endif
else
        OUT Y     ; else print(y)
endif
        END
```

6. Ein Programm soll zwei positive Zahlen einlesen, in einem Unterprogramm `mult` das Produkt berechnen und das Ergebnis ausgeben. Zuerst ist dazu ein entsprechendes Java-Programm zu schreiben, wobei darin Multiplikation nicht erlaubt ist. Dieses Java-Programm ist dann in ein Maschinen-Programm „zu Fuß zu kompilieren“.

```
class Vielfaches
{ static int zahl1,
             zahl2,
             ergebnis=0;

  static void multipliziere()
  {  while (zahl2>0)
     { ergebnis = ergebnis + zahl1;
       zahl2 = zahl2 - 1;
     }
  }

  public static void main(String args[]) // Hauptprogramm
  {  Out.println("Bitte ersten Faktor eingeben: ");
     zahl1=In.readInt();
     Out.println("Bitte zweiten Faktor eingeben: ");
     zahl2=In.readInt();
     multipliziere();
     Out.println("Das Ergebnis ist "+ergebnis);
  } // Ende von main
} // Ende von class Vielfaches
```

```
          JMP  Anfang
                          ;
Summand   DEF  04         ; int     summand;
Ergebnis  DEF  00         ; int     ergebnis;
WieOft    DEF  10         ; int     wieOft;

                          ; public void main(...)
Anfang                    ; {
          INM  Summand    ;   summand = In.readInt();
          INM  WieOft     ;   wieoft  = In.readInt();
          JSR  Addiere    ;   addiere();
          OUT  Ergebnis   ;   Out.print(ergebnis);
          END             ; }

Addiere                   ; public void  addiere()
                          ; {
DO
          LDA  Ergebnis   ;   do {
          ADD  Summand    ;        ergebnis = ergebnis + summand;
          STA  Ergebnis   ;
          LDA  WieOft     ;
          DEC             ;        wieOft = wieOft - 1;
          STA  WieOft     ;
          JPL  DO         ;      } while (wieOft > 0);
          RTN             ; }
```

Schwierigere Variante: Die beiden Zahlen können negativ oder nicht-negativ sein.

7. Die Multiplikation kann auch rekursiv über Additionen erfolgen. Schreibe dazu ein Java-Programm ohne Parameterübergabe und entwirf mit Hilfe dieses Programmes ein DCL-Maschinen-Programm.

```
class VielfachesRekursiv
{ static int zahl1, zahl2, ergebnis=0;

  static void addiere()
  {  ergebnis = ergebnis + zahl1;
     zahl2--;
     if (zahl2>0) addiere();
  }

  public static void main(String args[]) // Hauptprogramm
  {  Out.println("Bitte ersten Faktor eingeben: "); zahl1=In.readInt();
     Out.println("Bitte zweiten Faktor eingeben: "); zahl2=In.readInt();
     addiere();
     Out.println("Das Ergebnis ist "+ergebnis);
  } // Ende von main
} // Ende von class VielfachesRekursiv
```

```
; ***************************************************************
; *    Rekursion mit DC (einfache Addition bzw. Multiplikation)  *
; *    ohne Parameteruebergabe                                   *
; ***************************************************************
        JMP  Anfang
                          ;
Summand DEF  04           ; int    summand;
Ergebnis DEF 00           ; int    ergebnis;
WieOft  DEF  10           ; int    wieOft;

                          ; public void main(...)
Anfang                    ; {
        INM  Summand      ;   summand = In.readInt();
        INM  WieOft       ;   wieoft  = In.readInt();
        JSR  Addiere      ;   addiere();
        OUT  Ergebnis     ;   print(ergebnis)
        END               ; }

Addiere                   ; public void  addiere()
        LDA  Ergebnis     ; {
        ADD  Summand      ;     ergebnis = ergebnis + summand;
        STA  Ergebnis     ;
        LDA  WieOft       ;
        DEC               ;     wieOft = wieOft - 1;
        STA  WieOft       ;
        JNP  EndIf        ;     if (wieOft > 0)
        JSR  Addiere      ;         addiere();
EndIf
        RTN
                          ; } // addiere
```

Warum kann diese rekursive Variante der Multiplikation nicht alle im DC erlaubten Faktoren verarbeiten? *Bei zu großen Faktoren ergibt sich ein Stapelüberlauf (stack-overflow). Das kann man im DC sichtbar machen!*

7.4 Die Register

7.4.1 Aufgaben

Bei der Abarbeitung der Befehle ergibt sich immer die gleiche Struktur:

1. Befehlsholphase,
2. Decodierphase,
3. und Befehlsausführungsphase.

Die Befehlsholphase ist immer gleich:
PC $\longrightarrow$ AR; Lesen aus RAM; PC + 1 $\longrightarrow$ PC; DR $\longrightarrow$ IR

Der Ablauf der Befehlsausführungsphase soll hier für einige Befehle dargestellt werden:

Befehl	Ablauf
JMP 10	
LDA 01	
STA 03	
ADD 02	
INC	
DEC	
SUB 53	
JPL 66	
JSR 77	
RTN	
PSH	
POP	

Fragen:

1. Warum ist es günstig, schon bei der Befehlsholphase den PC um 1 zu erhöhen?
2. Was kann passieren, wenn ein Unterprogramm durch Rekursion zu oft aufgerufen wird?

7.4.2 Ergebnisse

Befehl	Ablauf
JMP 10	IR (Adr.) $\longrightarrow$ PC
LDA 01	IR (Adr.) $\longrightarrow$ AR, Lesen in DR, DR $\longrightarrow$ AC
STA 03	AC $\longrightarrow$ DR, IR (Adr.) $\longrightarrow$ AR, Schreiben
ADD 02	IR (Adr.) $\longrightarrow$ AR, Lesen in DR, AC und DR $\longrightarrow$ ALU(plus) $\longrightarrow$ AC
INC	AC + 1 $\longrightarrow$ AC
DEC	AC - 1 $\longrightarrow$ AC
SUB 53	IR (Adr.) $\longrightarrow$ AR, Lesen in DR, AC und DR $\longrightarrow$ ALU(minus) $\longrightarrow$ AC
JPL 66	Test: Akkuwert pos.?, Ja: IR (Adr.) $\longrightarrow$ PC
JSR 77	PC $\longrightarrow$ DR, SP $\longrightarrow$ AR, Schreiben, SP -1 $\longrightarrow$ SP, IR (Adr.) $\longrightarrow$ PC
RTN	SP + 1 $\longrightarrow$ SP, SP $\longrightarrow$ AR, Lesen in DR, DR $\longrightarrow$ PC
PSH	AC $\longrightarrow$ DR, SP $\longrightarrow$ AR, Schreiben, SP + 1 $\longrightarrow$ SP
POP	SP + 1 $\longrightarrow$ SP, SP $\longrightarrow$ AR, Lesen in DR, DR $\longrightarrow$ AC

Fragen:

1. Warum ist es günstig, schon bei der Befehlsholphase den PC um 1 zu erhöhen?
   Der um 1 erhöhte PC wird bei JSR auf den Stack geschrieben.

2. Was kann passieren, wenn ein Unterprogramm durch Rekursion zu oft aufgerufen wird?
   Durch zu häufiges „Pushen“ von Rücksprungadressen kann es zu einem Stapelüberlauf kommen.

7.5 Java-Methoden u.a. bzw. „Der Stack“

Der Code und die einfachen Daten eines Programmes beanspruchen einen festen Bereich im Rechnerspeicher. In einem Java-Programm sind die globalen Variablen zu deklarieren, damit der Compiler den nötigen Speicherplatz berechnen kann. Während des Programmlaufes können sich natürlich noch die Werte der Variablen ändern, nicht aber der dazu erforderliche Speicherplatz.
In vielen Situationen ist es aber notwendig und sinnvoll, einen dynamisch mit den Anforderungen wachsenden Datenbereich zu haben. Beispiele:

1. Beim Aufruf eines Unterprogrammes (Methode) muss irgendwo die **Rücksprungadresse** gespeichert werden. Bei Rekursion ist die Anzahl der nötigen Speicheradressen vorher nicht bekannt.

2. Unterprogramme bzw. Methoden können mit **lokalen Variablen** arbeiten, die nur während des Ablaufs des Unterprogrammes Speicherplatz beanspruchen. Nach Abarbeitung des Unterprogrammes wird der nötige Speicher wieder freigegeben.

3. Unterprogrammen können **Parameter** übergeben werden, die nur während des Ablaufs des Unterprogrammes benötigt werden. Bei Rekursion müssen die Parameter aller noch laufenden Unterprogramm-Instanzen gespeichert bleiben.

4. Funktionen müssen über irgendeine Speicheradresse ihre **Rückgabewerte** an das aufrufende Programm übergeben können.

Für all dies ist der sogenannte *Stack* (Stapel) zuständig.
Für weitere dynamische Datenstrukturen (z.B. Baumstrukturen, Listen u.a.) werden wir später den *Heap* (Haufen) kennen lernen.
Man kann sich den Stack als einen Stapel bestehend aus Zetteln vorstellen, wobei man immer nur den obersten Zettel lesen, einen neuen Zettel drauflegen oder einen Zettel wegnehmen kann.
Beim DC muss man sich diesen Stapel auf dem Kopf stehend vorstellen, denn der Stack wächst hierbei von der Speicherstelle 127 an abwärts. In dem SP (=StackPointer) genannten Register steht immer die Adresse der nächsten freien Speicherstelle auf dem Stack, zu Beginn eines Programmes steht darin also immer die Zahl 127.

1. **Rücksprungadresse:** Beim Aufruf eines Unterprogrammes wird durch den Befehl `JSR` der Wert der im Programm folgenden Speicheradresse auf dem Stack abgelegt und der StackPointer um 1 erniedrigt. Beim Auftreten von `RTN` wird der StackPointer zuerst um 1 erhöht und dann der Wert an dieser Stelle in den PC (Program Counter) übertragen. Damit kann das Programm an der Stelle weitermachen, die dem Aufruf des Unterprogrammes folgt.

2. **Lokale Variablen:** Nach dem Sprung zum Programmcode des Unterprogrammes kann durch `PSH`-Befehle auf dem Stack Platz für lokale Variable geschaffen werden. Dadurch ändert sich natürlich der SP. Durch spezielle Befehle, die Adressen relativ zum SP ansprechen können, kann mit diesen lokalen Variablen gearbeitet werden. Vor dem Befehl `RTN` müssen die zu Beginn des Unterprogrammes mit `PSH` angeforderten Speicherplätze durch die gleiche Anzahl an `POP`-Befehlen wieder freigegeben werden,

damit die Rücksprungadresse bereit steht. Achtung: Im Akkumulator steht nach den `POP`-Befehlen ein Wert einer lokalen Variablen.

3. **Parameter** für ein Unterprogramm müssen vor dem `JSR`-Befehl mit `PSH` auf den Stack gebracht werden. Sie stehen dann auf dem Stack oberhalb der Rücksprungadresse. Mit den speziellen Stack-Befehlen kann man auch auf diese Variablen zugreifen. Nach dem Rücksprung müss der benötigte Speicherplatz für die Parameter mit der entsprechenden Anzahl an `POP`-Befehlen wieder freigegeben werden. Achtung: Im Akkumulator steht nach den `POP`-Befehlen ein Wert einer lokalen Variablen, der aber nicht mehr benötigt wird.

4. **Rückgabewerte** kann man wie Parameter behandeln. Mit `PSH` macht man Platz für einen solchen Wert, wobei dabei der übergebene Wert unwichtig ist. Mit den speziellen Stackbefehlen kann die entsprechende Speicherstelle beschrieben werden. Der Wert, den man nach dem `RTN` durch `POP` im Akkumulator erhält, ist der Funktionswert des Unterprogrammes.

5. Bei rekursiven Aufrufen von Unterprogrammen mit Parametern ändert sich während des Laufes des Unterprogrammes der SP durch die notwendigen `PSH`-Befehle. Um trotzdem auf die lokalen Variablen und die Parameter mit fester Adresse zugreifen zu können, gibt es neben dem SP das Hilfsregister BP (=BasePointer), in das der SP zu Beginn eines Unterprogrammes kopiert werden kann. Es gibt außerdem spezielle Befehle, die die Adressierung des Stacks über den BP ermöglichen. Am Ende des Unterprogrammes kopiert man dann den BP zurück in den SP.

7.5.1 Beispiel

Das folgende Programm berechnet die Summe der ganzen Zahlen von 1 bis einer einzulesenden Zahl k. Die Berechnung erfolgt mit einer `while`-Schleife in einer Methode `int addiere(int zahl)` mit Parameterübergabe und Rückgabewert.

```
; PROGRAMM SumJSRSt.DCL
; Die ganzen Zahlen von 1 bis k werden addiert
; mit einer WHILE-Schleife in einer Methode Addiere
; inklusive Parameteruebergabe und Rueckgabewert

        JMP Anfang

null    DEF 000
summe   DEF 000
k       DEF 5

Anfang                  ; public void main(...)
        INM k           ; {  In.read(k);
        LDA k
        PSH             ;    // Platz fuer Funktionserg. (Returnwert),
                        ;    // Wert unwichtig
        PSH             ;    // k auf Stack, in zahl kopieren
        JSR addiere     ;    // Methodenaufruf
        POP             ;    // lokale Variable bzw. Parameter zahl vernichten
        POP             ;    // Funktionsergebnis vom Stack nehmen in Akku
        STA summe       ;    summe = addiere(k);
        OUT summe       ;    Out.print(summe);
        END             ; }

addiere                   ; int addiere(int zahl)
        PSH               ; {  int ergebnis (Platz fuer lokale Variable)
        returnwert EQUAL 4 ;    //  SP + 4
        zahl       EQUAL 3 ;    //  SP + 3
        RSprAdr    EQUAL 2 ;    //  SP + 2
        ergebnis   EQUAL 1 ;    //  SP + 1
        LDA null          ;
        STAS ergebnis     ;      ergebnis = 0;
WhileAnfang
        LDAS zahl         ;      while (zahl > 0)
        JNP WhileEnde     ;      {
        LDAS ergebnis     ;
        ADDS zahl         ;
        STAS ergebnis     ;        ergebnis = ergebnis + zahl;
        LDAS zahl
        DEC
        STAS zahl         ;        zahl = zahl - 1;
        JMP WhileAnfang   ;      } // end von while
WhileEnde                 ;
        LDAS ergebnis
        STAS returnwert   ;      return ergebnis;
        POP               ;      lok. Variable ergebnis vernichten
        RTN               ;    } end von addiere
```

7.5.2 Aufgaben

1. Das oben dargestellte Programm ist zu analysieren. Es wird z.B. die Zahl 3 eingelesen. Zeichne den Stack mit vollständigem Inhalt,
   (a) nachdem `JSR addiere` ausgeführt wurde.
   (b) wenn zum ersten Mal im Programm `WhileAnfang` erreicht wird.
   (c) bevor `RTN` ausgeführt wird.
   (d) bevor `OUT summe` ausgeführt wird.
2. Schreibe ein Programm, das das Quadrat einer beliebigen Zahl berechnet:
   (a) mit einem Unterprogramm `quadrat` ohne Parameter und Rückgabewert.
   (b) mit einem Unterprogramm `quadrat`, das die zu quadrierende Zahl als Parameter über den Stack geliefert bekommt.
   (c) mit einem Unterprogramm `quadrat`, das die zu quadrierende Zahl als Parameter über den Stack geliefert bekommt und das Ergebnis als Rückgabewert bzw. Funktionswert zurück liefert.
3. Was bewirkt das folgende Programm?

```
        JSR WasIstDas
        END
WasIstDas                   ; void WasIstDas()
                            ; {   Ruecksprungadr.  SP+2
                            ;   int zahl;          SP+1
        zahl EQUAL 1        ;                      SP
        PSH                 ;   // Platz fuer zahl schaffen
        INS zahl            ;   zahl = In.readInt();
        OUS zahl            ;   Out.print(Zahl);
        LDS zahl            ;
        JZE EndIf           ;   if (    )
                            ;       {
        JSR WasIstDas       ;        WasistDas();
        OUS zahl            ;        Out.print(zahl);
                            ;       }
EndIf
        POP                 ;   // Platz fuer zahl freigeben
        RTN                 ; } // Ende von WasIstDas
```

4. Das Programm mit der Methode `WasistDas` ist so umzuformulieren, dass statt des SP der BP für die Adressierung verwendet wird.
5. Schwierig! Die Fibonacci-Zahlen sollen rekursiv ausgegeben werden.
6. Schwierig! Das Problem der „Türme von Hanoi“ soll mit DC gelöst werden.

7.5.3 Lösungen der Aufgaben

1. Das oben dargestellte Programm ist zu analysieren. Es wird z.B. die Zahl 3 eingelesen. Zeichne den Stack mit vollständigem Inhalt,

(a) nachdem `JSR addiere` ausgeführt wurde.

Nach PSH, PSH, und JSR sieht der Stack so aus (er wächst von oben nach unten):

leer (beschrieben mit 3, aber unwichtig)
Parameter k (3)
Rücksprungadresse

(b) wenn zum ersten Mal im Programm `WhileAnfang` erreicht wird.

Nach PSH und STAS ergebnis:

leer (beschrieben mit 3, aber unwichtig)
Parameter k (3)
Rücksprungadresse
Lokale Variable ergebnis (0)

(c) bevor `RTN` ausgeführt wird.

Nach POP („Vernichtung“ der lokalen Variablen ergebnis):

Nun mit Funktionswert 6 beschrieben.
Parameter k (0)
Rücksprungadresse

(d) bevor `OUT summe` ausgeführt wird.

Nach POP und POP ist das Funktionsergebnis 6 im Akkumulator und der Platz für den Parameter zahl „vernichtet“. Der Stack ist leer.

2. Schreibe ein Programm, das das Quadrat einer beliebigen Zahl berechnet:

   (a) mit einem Unterprogramm `quadrat` ohne Parameter und Rückgabewert.

```
; PROGRAMM Quadrat1.dcl
; Das Quadrat einer DC-Zahl X mit X < 45 ist zu berechnen. Negative
; Zahlen seien dabei zugelassen. Zu Beginn soll die Basis bei
; negativem Vorzeichen durch ihren Betrag ersetzt wird.
; Auch ein zweiter Programmlauf mit anderem X sollte moeglich sein.
; Hier: Simulation einer Methode quadrat() ohne Parameteruebergabe

          JMP  Anfang
null      DEF  00        ; final int null = 0;
x         DEF  00        ; int   x;       // nur globale Variablen
quad      DEF  00        ; int   quad;
zaehler   DEF  00        ; int   zaehler;

Anfang    INM  x         ; x = In.readInt();
          LDA  x
          JNM  EndIf1    ; if (x < 0)
          NEG            ;     x = -x;
          STA  x         ;
EndIf1
          JSR  Quadrat
          OUT  quad      ; print (quad)
          END            ; Ende des Hauptprogrammes

quadrat                  ; quadrat()
          LDA  Null      ; { // Beginn der Methode
          STA  quad      ; quad = 0;
          LDA  x         ; Basis x laden
          STA  zaehler   ; zaehler = x;
WhileAnf
          JNP  EndWhile  ; while (zaehler > 0)
          LDA  quad      ; {
          ADD  x
          STA  quad      ;     quad = quad + x
          LDA  zaehler
          DEC            ;     zaehler  = zaehler - 1;
          STA  zaehler
          JMP  WhileAnf  ; } // Ende von while
EndWhile
          RTN            ; } // Ende der Methode quadrat
```

(b) mit einem Unterprogramm `quadrat`, das die zu quadrierende Zahl als Parameter über den Stack geliefert bekommt.

```
; PROGRAMM Quadrat2.dcl
; Hier: Simulation einer Methode quadrat(int zahl)
; mit Parameteruebergabe

          JMP Anfang
null      DEF 00        ; final int null = 0;
x         DEF 00        ; int   x;
quad      DEF 00        ; int   quad;
zaehler   DEF 00        ; int zaehler; // Hilfvariable
                        ;              // beim Zaehlen in addiere


Anfang    INM x         ; x = In.readInt();
          LDA x
          JNM EndIf1    ; if (x < 0)
          NEG           ;     x = -x;
          STA x         ;
EndIf1
          PSH           ; x ueber Stack an Methode uebergeben.
          JSR Quadrat
          POP           ; x vernichten
          OUT quad      ; print (quad)
          END           ; Ende des Hauptprogrammes


quadrat                 ; quadrat (int Zahl)
                        ; Lokale Variablen auf Stack von oben:
zahl      EQUAL 2       ;  zahl                <- SP + 2
                        ;  ruecksprungadresse  <- SP + 1
                        ;                      <- SP
                        ; { // Beginn der Methode
          LDA Null
          STA quad      ; quad = 0;
          LDAS zahl     ; zahl vom Stack
          STA zaehler   ; zaehler = zahl
WhileAnf
          JNP EndWhile  ; while (zaehler > 0)
          LDA quad      ; {
          ADDS zahl
          STA quad      ;     quad = quad + zahl (auf Stack)
          LDA zaehler
          DEC           ;     zaehler  = zaehler - 1;
          STA zaehler
          JMP WhileAnf  ; } // Ende von while
EndWhile
          RTN           ; } // Ende der Methode quadrat
```

(c) mit einem Unterprogramm `quadrat`, das die zu quadrierende Zahl als Parameter über den Stack geliefert bekommt und das Ergebnis als Rückgabewert bzw. Funktionswert zurück liefert.

```
; PROGRAMM Quadrat3.dcl
; Hier: Simulation einer Methode int quadrat(int zahl) mit
; Parameteruebergabe. Auch das Funktionsergebnis wird
; ueber den Stack zurueckgeliefert.

          JMP Anfang
null      DEF 00          ; final int null = 0;
x         DEF 00          ; int   x;
quad      DEF 00          ; int   quad;

Anfang    INM x           ; x = In.readInt();
          LDA x
          JNM EndIf1      ; if (x < 0)
          NEG             ;     x = -x;
          STA x           ;
EndIf1
          PSH             ; Platz fuer Funktionsergebnis schaffen
          PSH             ; x ueber Stack an Function uebergeben.
          JSR Quadrat
          POP             ; x vernichten
          POP             ; Funktionsergebnis vom Stack holen
          STA quad        ; und speichern
                          ; quad = quadrat(x);
          OUT quad        ; Out.print (quad);
          END             ; end (* HAUPTPROGRAMM *)


quadrat                   ; int quadrat (int Zahl)
                          ; Lokale Variablen auf Stack von oben:
quaderg   EQUAL 5         ;  quaderg              <- SP + 5
zahl      EQUAL 4         ;  zahl                 <- SP + 4
                          ;  ruecksprungadresse   <- SP + 3
zaehler   EQUAL 2         ;  zaehler              <- SP + 2
ergebnis  EQUAL 1         ;  ergebnis             <- SP + 1
                          ;                       <- SP
                          ; { // Beginn der Funktion
          PSH             ; Platz fuer Zaehler    (lokale Variable)
          PSH             ; Platz fuer Ergebnis   (lokale Variable)
          LDA Null
          STAS ergebnis   ; ergebnis = 0;
          LDAS zahl       ; zahl vom Stack
          STAS zaehler    ; zaehler = zahl
WhileAnf
          JNP EndWhile    ; while (zaehler > 0)
          LDAS ergebnis   ; {
          ADDS zahl
          STAS ergebnis   ;     ergebnis = ergebnis + zahl (auf Stack)
          LDAS zaehler
          DEC             ;     zaehler  = zaehler - 1
          STAS zaehler
          JMP WhileAnf    ; } // Ende von while
EndWhile
```

```
        LDAS ergebnis
        STAS quaderg   ; Ergebnis auf Stack zurueckliefern
        POP            ; Ergebnis vernichten (Platz freigeben)
        POP            ; Zaehler  vernichten
        RTN            ; } // Ende der Funktion Quadrat
```

3. Was bewirkt das folgende Programm?

```
        JSR WasIstDas
        END
WasIstDas                   ; void WasIstDas()
                            ; {   Ruecksprungadr.  SP+2
                            ;   int zahl;          SP+1
        zahl EQUAL 1        ;                      SP
        PSH                 ;   // Platz fuer zahl schaffen
        INS zahl            ;   zahl = In.readInt();
        OUS zahl            ;   Out.print(Zahl);
        LDS zahl            ;
        JZE EndIf           ;   if (    )
                            ;       {
        JSR WasIstDas       ;        WasistDas();
        OUS zahl            ;        Out.print(zahl);
                            ;       }
EndIf
        POP                 ;   // Platz fuer zahl freigeben
        RTN                 ; } // Ende von WasIstDas
```

Die eingegebenen Zahlen werden durch die Rekursion am Ende in umgekehrter Reihenfolge ausgegeben.

4. Das Programm mit der Methode `WasistDas` ist so umzuformulieren, dass statt des SP der BP für die Adressierung verwendet wird.

   Lösung: siehe Beispielprogramm Umkehr.dcl oben

5. Schwierig! Die Fibonacci-Zahlen sollen rekursiv ausgegeben werden.

```
; Fibonacci-Zahlen rekursiv berechnen fuer DC
; Parameter, Zwischen- und Funktionsergebnis auf Stack

            JMP Anfang
Null        DEF 000             ; final int Null = 0;
Eins        DEF 001             ; final int Eins = 1;
Zwei        DEF 002             ; final int Zwei = 2;
EndWert     DEF 005             ; int   EndWert;
K           DEF 000             ; int   K;
FiboResult  DEF 000             ; int   FiboResult;

            ; public void action(){  // Hauptprogramm
Anfang
            INM EndWert         ; Endwert = In.readInt();
            LDA Eins
            STA K
WhileAnfang
            LDA K               ; while (K <= EndWert)
            SUB EndWert         ; {
```

```
            JPL  WhileEnde
            PSHM K                  ;       Parameter K auf Stack
            JSR  Fibo               ;       Aufruf
            POPM FiboResult         ;       FiboResult = Fibo(K)
            OUT  FiboResult         ;       Out.Print(FiboResult);
            LDA  K
            INC
            STA  K                  ;        K = K + 1;
            JMP  WhileAnfang        ; } // Ende von while
WhileEnde
            END                     ; } // Ende von action (Hauptprogramm)

Fibo                                ; int Fibo(int N)
            N       EQUAL 3         ;         N         SP + 3
                                    ;         RSA       SP + 2
            Ablage EQUAL 1          ;         Ablage    SP + 1
                                    ; {
            PSH                     ;   Platz fuer Ablage schaffen
            LDAS N
            JNZ  Else1              ; if (N == 0)
            LDA  Null               ;       {
            STAS N                  ;       return 0;
            JMP  EndIf1             ;       }
Else1                               ; else {
            LDAS N
            SUB  Eins
            JNZ  Else2              ;       if (N == 1)
            LDA  Eins               ;            {
            STAS N                  ;            return 1;
            JMP  EndIf2             ;            }
Else2
            LDAS N                  ;       else {
            DEC                     ;            Berechne  N - 1
            PSH
            JSR  Fibo               ;            Aufruf
            POP
            STAS Ablage             ;            Ablage = Fibo(N-1)
            LDAS N
            SUB  Zwei               ;            Berechne N - 2
            PSH
            JSR  Fibo               ;            Aufruf
            POP                     ;            Accu := Fibo (N-2)
            ADDS Ablage             ;            Accu := Accu + Ablage
            STAS N                  ;            Fibo := Fibo (N - Eins) +
                                    ;                    Fibo (N - Zwei);
        EndIf2                      ;            }
EndIf1                              ;       }
            POP                     ;   // Platz fuer Ablage wieder frei
            RTN                     ; } // Ende von Fibo
```

6. Schwierig! Das Problem der „Türme von Hanoi“ soll mit DC gelöst werden.

   Lösung: siehe oben

7.6 Grenzen des Modells

Wie jedes Modell hat auch dieses seine Grenzen. Verschiedene anspruchsvolle Adressierungstechniken sind nicht möglich, ebensowenig die Arbeit mit Pointern, obwohl die Adressierung mit dem BP solche Denkweisen fördert. Die Veranschaulichung von Objekten ist dementsprechend praktisch unmöglich. Da die Schüler aber nicht zu Maschinensprache-Programmmierern ausgebildet werden sollen, kann auf eine eingehende Behandlung dieser Themen verzichtet werden. Ein Ausblick auf die Möglichkeiten eines realen Prozessors im Rahmen eines Referates sollte genügen.
An eine Erweiterung des Rechnermodells durch weitere Befehle ist z.Zt. nicht gedacht, da es sich im Unterricht in dieser Form bewährt hat. Gegenüber konstruktiven Verbesserungsvorschlägen möchte ich allerdings offen bleiben.

8 Editor

Zum Editor sind nur wenige Anmerkungen zu machen. Es ist ein einfacher ASCII-Editor, der keine Textformatierungen ermöglicht. Mehrere Textfenster sind ebensowenig möglich wie Makro-Programmierung o.ä. Allerdings beschränkt er die Textgröße nicht wie bei den Turbo-Pascal-Editoren üblich auf 64 kB. Die Zeilenlänge ist nicht auf 125 Zeichen beschränkt. Für das Schreiben von DCL-Quelltexten wird man dies aber wohl kaum ausnutzen.
Die Tastenbelegung:

Zeichen nach links	←
Zeichen nach rechts	→
Wort nach links	CTRL-←
Wort nach rechts	CTRL-→
Zum Zeilenanfang	Home bzw. Pos1
Zum Zeilenende	End
Zeile nach oben	↑
Zeile nach unten	↓
Seite nach oben	PgUp
Seite nach unten	PgDn
Zum Dateianfang	CTRL-PgUp
Zum Dateiende	CTRL-PgDn
Aufwärts rollen	CTRL-Home
Abwärts rollen	CTRL-End
Zum Blockanfang	SHIFT-F3
Zum Blockende	SHIFT-F4
Blockanfang markieren	F3
Blockende markieren	F4
Block kopieren	F5
Block löschen	F6
Block verschieben	F7
Wort markieren	F8
Block lesen	F9
Block schreiben	F10
Block verdecken/zeigen	ALT-F3
Block nach rechts	SHIFT-F6 oder CTRL-KF
Block nach links	SHIFT-F5 oder CTRL-KA
Zeile einfügen	F1
Zeile löschen	F2
Zeile ab Cursor löschen	SHIFT-F2
Suchen	SHIFT-F9
Suchen und Ersetzen	SHIFT-F10
Wiederholung der Suche	ALT-F10

Aut. Tabulierung an/aus	SHIFT-Tab
Insert/Overwrite	Ins bzw. Einfg
Kontrollzeichen-Präfix	CTRL-P
Operation unterbrechen	CTRL-U
Paragraphenzeichen (IBM)	CTRL-O

Über den Befehlsumfang kann man sich auch im Editor in einem Hilfefenster informieren. Die Funktionstastenbelegung weicht etwas vom Turbo-Standard (F7 und F8) ab. Die wichtigsten Wordstar-Tastenkombinationen, die sich bei vielen anderen Editoren wiederfinden, sind neben den angegebenen Tastenkombinationen implementiert. Für die sonstige Bedienung muss man eigentlich nur wissen, daß mit ESC das Hauptmenü erscheint. Alles andere erklärt sich weitgehend von selbst. Für das Dateiauswahlmenü sollte man noch wissen, daß mit CONTROL + Laufwerksbuchstabe das Laufwerk gewechselt werden kann. Die Default-Datei-Extension ist DCL.

9 DC in Zeiten von Windows-7 und -8

DC.EXE ist ein 16-Bit-Programm, das für die DOS-Konsole geschrieben wurde und keine Mausbedienung vorsieht. Bis Windows-XP konnte es noch ohne Probleme eingesetzt werden. Windows-7 lehnt aber sogar ältere Windows-Programme ab, sodass ein direktes Ausführen von DC.EXE nicht möglich ist. Man kann aber unter Windows-7 eine DOS-Emulation starten und darin DC.EXE ohne Probleme ausführen.

9.1 Einrichten der DOS-Emulation

- Auf der Seite `http://www.dosbox.com` findet man unter *Downloads* für Windows den Win32Installer `DOSBox0.74-win32-installer.exe`.
- Das Programm herunterladen und mit Administratorrechten einrichten.

9.2 Arbeiten mit der DOS-Emulation und DC

- Es muss später angegeben werden, in welchem Verzeichnis man arbeiten möchte. Dieses Verzeichnis wird dann als ein virtuelles Laufwerk eingebunden. Erzeuge dazu **auf der Windows-Ebene** z.B. auf Laufwerk `Q:` das Verzeichnis `Q2If1` und darin das Verzeichnis `DC`. Es entsteht das Verzeichnis `Q:\Q2If1\DC`. Das soll das Arbeitsverzeichnis sein.
- Im Verzeichnis `C:\Informatik\DC` findest Du das Programm DC.EXE. Kopiere es in das vorher angelegte Arbeitsverzeichnis.
- Starte die Emulation durch Anklicken des Links *DOSBox*.
- Mit Alt-Enter wechselt man in den Vollbildmodus und wieder zurück.
- Mit dem Befehl `MOUNT X Q:\Q2If1\DC` wird das Arbeitsverzeichnis unter dem Laufwerksnamen X erreichbar.
- Mit `X:` und Enter wechselt man in das Arbeitsverzeichnis.
- Kontrolle: Mit dem Befehl `DIR` und Enter müsste `DC.EXE` erscheinen.
- Durch die Eingabe `DC.EXE` und Enter wird DC gestartet.
- DC (bzw. der Assembler) kann mit den Dateien im Arbeitsverzeichnis arbeiten. Wenn man aber durch ein Windows-Programm eine Datei im Arbeitsverzeichnis erzeugt, muss man in der DOSBox durch STRG-F4 die Dateiliste aktualisieren.
- DC verlässt man durch Eingabe von `Q` für Quit.
- Die DOSBox verlässt man durch Eingabe von `EXIT`.
- **Dateinamen:** Unter DOS können nicht beliebige Dateinamen gewählt werden. Möglich sind maximal acht Zeichen vor und drei Zeichen hinter dem Punkt. Für den Assembler ist z.B. `ABCDEFGH.DCL` ein gültiger Dateiname. Leerzeichen und die meisten Sonderzeichen (außer dem Unterstrich) sind nicht erlaubt.

10 Zur Entstehungsgeschichte

- Zbigniew Szkaradnik
  - Version 1.1 für DEC LSI-11 — 25.04.1985.
  - Version 1.2 für JOYCE — 10.02.1987.
- Michael Ceol
  - Version 1.3 für Turbo-Pascal — in Zeitschrift 'PASCAL', Heft 12/87
- Horst Gierhardt
  - komplette Neuformulierung und -gestaltung in MODULA-2 (LOGITECH MODULA 2/86, Version 3.40) mit Anpassung an WANG und IBM-Kompatible.
  - Verbesserung der Benutzeroberfläche und umfangreiche Befehlssatzerweiterungen.
  - Integration eines Editors und eines Mini-Assemblers.
  - Modula 2 Version 4 (1993);
  - Laufwerke A: bis Z: erreichbar (2001/02)

11 Bekannte Probleme

Beim Einsatz in Netzwerken kann es kleinere Probleme geben. Der Editor lädt beispielweise eine Datei nicht vollständig. Beim zweiten Laden ist die Datei aber vollständig. Bisher konnte ich den Fehler noch nicht lokalisieren. Es empfiehlt sich aber sowieso, auf den internen Editor zu verzichten, da er in der Bedienung veraltet ist. Es reicht, mit Notepad oder einem anderen beliebigen Texteditor zu arbeiten.

12 Sonstiges

Bei Rückfragen bzw. Verbesserungsvorschlägen wenden Sie sich bitte an:

Horst Gierhardt
http://www.gierhardt.de
Horst@Gierhardt.de