

VON-NEUMANN-RECHNER auf dem Bildschirm
CPU-Simulation mit dem
Didaktischen Computer DC ¹

Horst Gierhardt
www.gierhardt.de

19. März 2003

¹Version 6.6

Inhaltsverzeichnis

1	'Hardware'-Beschreibung	3
1.1	Zentraleinheit	3
1.2	Register	3
1.2.1	Befehlsregister (IR = instruction register)	3
1.2.2	Befehlszählregister (PC = program counter)	3
1.2.3	Adressregister (AR)	4
1.2.4	Datenregister (DR)	4
1.2.5	Akkumulator (AC)	4
1.2.6	Stapelzeiger (SP = stack pointer)	4
1.2.7	Basisregister (BP = base pointer)	4
1.3	Datenübertragung	4
1.3.1	Adressbus	5
1.3.2	Datenbus	5
1.3.3	Steuerleitungen	5
2	Befehlssatz	5
2.1	Grundbefehle	6
2.2	Sprungbefehle	6
2.3	Stackoperationen (SP-orientiert)	7
2.4	Stackoperationen (BP-orientiert)	8
3	Syntax der Befehle	8
4	Programmsteuerung	9
5	Assembler	10
6	Beispiele	10
6.1	Beispiel ohne Assembler-Benutzung	10
6.2	Beispiel mit Assembler-Benutzung	11
6.3	Beispiel mit Stackadressierung ohne Parameter	11
6.4	Beispiel mit Rekursion und Parameterübergabe	12
6.5	Vom PASCAL- zum DC-Code	14
6.5.1	Das PASCAL-Programm	14
6.5.2	Das MINI-PASCAL-Programm	15
6.5.3	Das DCL-Programm	15
7	Unterrichtseinsatz	17
7.1	Idee zur Einführung in den Themenkreis	17
7.2	Erste Ergebnisse und Informationen zur Simulation	18
7.3	Der DC im Unterricht	19
7.4	Grenzen des Modells	20
8	Editor	20
9	Zur Entstehungsgeschichte	21
10	Bekannte Probleme	22

11 Sonstiges

22

1 ‘Hardware’-Beschreibung

1.1 Zentraleinheit

Das Programm DC simuliert auf dem Bildschirm eine einfache Zentraleinheit (CPU = **central processing unit**) eines Computers nach dem *Von-Neumann-Prinzip* mit

- Rechenwerk (ALU = **arithmetical logical unit**),
- Steuerwerk bzw. Kontrolleinheit,
- Speicher mit Schreib-Lese-Zugriff (RAM = **random access memory**), 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. Gängige Werte für Adressbusbreiten in realen Rechnern sind 8 Bit, 16 Bit, 20 Bit, 24 Bit, 32 Bit 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. Gängige Werte für Datenbusbreiten in realen Rechnern sind 8 Bit, 16 Bit und 32 Bit. Der Wang-PC mit Intel-8086-Mikroprozessor besitzt eine Adressbusbreite von 20 Bit und eine Datenbusbreite von 16 Bit. Intern verarbeitet er 16 Bit. Entsprechend wird er als 16-Bit-Computer klassifiziert. Rechner mit dem Intel-80386 haben eine Daten- und Adressbusbreite von je 32 Bit und können intern (z.B. im Akkumulator) 32 Bit mit einem Befehl verarbeiten.

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.

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.

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.

Mnemo	Intern	Bedeutung
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 $XXX \geq 0$ 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 ...

Mnemo	Intern	Bedeutung
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 $XXX \geq 0$ 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 ...
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 arbeiten. Die symbolischen Adressen werden ähnlich den Labels in Pascal benutzt, können aber auch als Bezeichner für Variablen benutzt werden. Der Doppelpunkt hinter den Labels ist nicht erforderlich, wird aber unterstützt, da man so beim Umsetzen eines Pascal-Quelltextes mit Labels in die Mini-Assembler-Form viele Textteile direkt übernehmen kann.

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 Dateierweiterung 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.

```

; PROGRAM 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 Programmstart an Adresse 10 springen.
        ; Datenbereich
01 DEF -3 ; VAR   A = -3.
02 DEF 7  ;       B = 7.
03 DEF 00 ;       C = 0.
        ; BEGIN
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 Akkuinhalt an Adresse 03.
14 OUT 03 ; uebergib den Inhalt von Adresse 03 an Output-Einheit.
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.

```

; PROGRAM Einfach.dcl (* Bitte Endung DCL beachten ! *)
JMP Anfang
A:   DEF 000 ; VAR  A,
B:   DEF 000 ;       B,
C:   DEF 000 ;       C : INTEGER;
Anfang: ; BEGIN
      INM A      ; Read (A);
      INM B      ; Read (B);
      LDA A
      INC
      ADD B
      STA C      ; C := (A+1) + B;
      OUT C      ; Write (C);
      END       ; 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' Programmierweise.

```

; PROGRAM Umkehr.dcl
                                ; BEGIN (* Hauptprogramm *)
      JSR Umkehr                ;   Umkehr;
      END                       ; END. (* Hauptprogramm *)

Umkehr                          ;PROCEDURE Umkehr;
                                ;   Ruecksprungadresse BP+2
      VAR Zahl : INTEGER;       ;BP+1
      Zahl EQUAL 1              ;   BP = SP
      PSH                       ;BEGIN (* Platz fuer Zahl schaffen *)
      SPBP                      ; SP -> BP
      INB Zahl                   ;   Read (Zahl);

```

```

        OUTB Zahl          ; Write (Zahl);
        LDAB Zahl          ;
        JZE EndIf          ; IF (Zahl <> 0) THEN
                        ; BEGIN
        JSR Umkehr          ; Umkehr rekursiv aufrufen.
        SPBP               ; SP -> BP
        OUTB Zahl          ; Write (Zahl);
                        ; END
EndIf
        POP                ; (* Platz fuer Zahl freigeben *)
        RTN                ;END; (* 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 Prozedur 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 gewährleistet sein, daß bei einem RTN die richtige Adresse auf dem Stack verfügbar ist. Beim Eintritt in eine Prozedur 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 Prozedurende muss man meistens — wie auch hier demonstriert — die lokalen Variablen ‘vernichten’ und die Rücksprungadresse an die richtige Position setzen.

```

; *****
; ***** Tuerme von Hanoi fuer DC (ab Version 6.1) *****
; *****
; 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. Das kann allerdings dauern. Fuer 6 Scheiben
; benoetigt ein 8--MHz--AT etwa 20s.
; Version fuer DC mit BasePointer BP und Mini-Assembler

; PROGRAM Tuerme_von_Hanoi_rekursiv_mit_DC;

        JMP Anfang

GlobalStart DEF 01          ; CONST GlobalStart = 1
GlobalAblage DEF 02         ;           GlobalAblage = 2
GlobalZiel DEF 03           ;           GlobalZiel = 3
GlobalHoehe DEF 00         ; VAR   GlobalHoehe

                        ; BEGIN (* Hauptprogramm *)
Anfang
        INM GlobalHoehe    ; Read (GlobalHoehe)
                        ; (* 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                ; END (* Hauptprogramm *)

```

Transport

```

; PROCEDURE Transportiere (Hoehe, Start,
;                           Ziel, Ablage);
; BEGIN (* Transportiere *)
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 THEN (* Rekursion *)
;   BEGIN
  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);
;   END; (* IF *)
EndIf1  SPBP      ; SP --> BP (* neu setzen, da durch *)
;                           (* Rekursion veraendert. *)
  OUTB Start      ;   Write ('Lege Scheibe von Turm <Start> ');
  OUTB Ziel       ;   Write ('           auf Turm <Ziel>.);

  LDAB Hoehe
  DEC

;   IF Hoehe-1 > 0 THEN (* Rekursion *)
;   BEGIN
  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);
;   END; (* IF *)
;   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

```

```

POP
RTN                ; END (* PROCEDURE Transport *)

```

6.5 Vom PASCAL- zum DC-Code

Bei der Entwicklung eines DC-Programmes bietet sich bei nicht allzu trivialen Problemen die folgende Vorgehensweise an:

1. Codierung des Algorithmus' in PASCAL mit den üblichen Kontrollstrukturen (IF-THEN-ELSE, REPEAT-UNTIL, WHILE-DO,...)
2. Codierung des zugehörigen MINI-PASCAL-Programmes durch Anwendung von Übersetzungsschablonen. Dieses Programm enthält keine der oben angegebenen Kontrollstrukturen mehr, sondern nur noch Sprungbefehle und Labels. Die auftretenden Zuweisungen sind weitgehend in Richtung auf Registerarithmetik vereinfacht. Boolesche Ausdrücke mit AND und OR müssen in einfache IF-THEN-ELSE-Strukturen mit Sprungbefehlen aufgelöst werden.
3. Das MINI-PASCAL-Programm wird in das DCL-Format übertragen. Die Labels bleiben dabei weitgehend erhalten. Die IF-THEN-ELSE-Strukturen werden noch insofern vereinfacht, daß nur noch Abfragen nach dem Vorzeichen des Akkuinhaltes auftreten dürfen (siehe DC-Befehlssatz). Das Problem der Speicheraufteilung ist hierbei zu lösen: Wo stehen Daten? Wo steht das Hauptprogramm? Wo steht der Code der Prozeduren?
4. Nach dem Assemblieren mit dem MINI-Assembler kann das fertige DC-Programm im DC geladen und ausgeführt werden.

Als Beispiel für diese Vorgehensweise sind die drei Programme für die Ausgabe der Fibonaccizahlen unten aufgeführt. Das am Ende entstehende DC-Programm ist nicht aufgeführt, da es nur die absoluten Adressen und keinerlei Kommentare enthält.

6.5.1 Das PASCAL-Programm

```

PROGRAM Fibonacci;
VAR K          : INTEGER;
    EndWert    : INTEGER;
(* ----- *)
FUNCTION Fibo (N : INTEGER) : INTEGER;
BEGIN
  IF N = 0 THEN
    Fibo := 0
  ELSE BEGIN
    IF N = 1 THEN
      Fibo := 1
    ELSE BEGIN
      Fibo := Fibo (N - 1) + Fibo (N - 2);
    END;
  END;
END;
END; (* Fibo *)

BEGIN (* ----- *)
  ClrScr;
  Write ('Fibonacci-Folge bis zum Index : ');
  ReadLn (EndWert);
  FOR K := 1 TO EndWert DO BEGIN

```

```

        GotoXY (1, 1 + K); Write ('Fibo(', K : 2, ') = ', Fibo (K) : 6);
    END;
END.

```

6.5.2 Das MINI-PASCAL-Programm

```

PROGRAM Fibonacci_nur_mit_Labels;
CONST Null      = 0;
      Eins      = 1;
      Zwei      = 2;
VAR   K         : INTEGER;
      EndWert   : INTEGER;

Label WhileAnfang, WhileEnde;

(* ----- *)
FUNCTION Fibo (N : INTEGER) : INTEGER;
Label Else1, Else2, EndIf1, EndIf2;
VAR Ablage : INTEGER;
BEGIN
    IF N <> 0 THEN GOTO Else1;
        Fibo := Null;
        GOTO EndIf1;
    Else1:
        IF N <> Eins THEN GOTO Else2;
            Fibo := Eins;
            GOTO EndIf2;
        Else2:
            Ablage := Fibo (N - Eins);
            Fibo := Ablage + Fibo (N - Zwei);
        EndIf2:
    EndIf1:
END; (* Fibo *)

BEGIN
    Write ('Fibonacci-Folge bis zum Index : ');
    ReadLn (EndWert);
    K := Eins;
    WhileAnfang:
    IF K > EndWert THEN GOTO WhileEnde;
        WriteLn (Fibo (K));
        K := K + 1;
        GOTO WhileAnfang;
    WhileEnde:
END.

```

6.5.3 Das DCL-Programm

```

; Fibonacci-Zahlen rekursiv berechnen fuer DC
; Parameter, Zwischen- und Funktionsergebnis auf Stack
; Version mit BasePointer BP (ab Version 6.1)
                JMP Anfang
Null            DEF 000                ; CONST Null = 0
Eins            DEF 001                ;      Eins = 1
Zwei           DEF 002                ;      Zwei = 2
EndWert        DEF 005                ; VAR   EndWert,

```

```

K          DEF 000          ;          K,
FiboResult DEF 000          ;          FiboResult : DCINTEGER;

; BEGIN (* Hauptprogramm *)
Anfang
INM EndWert          ; Read (Endwert);
LDA Eins
STA K
WhileAnfang
LDA K                ; WHILE (K <= EndWert) DO BEGIN
SUB EndWert
JPL WhileEnde
PSHM K              ;          Parameter K auf Stack
JSR Fibo            ;          Aufruf
POPM FiboResult     ;          FiboResult := Fibo (K)
OUT FiboResult      ;          Write (FiboResult)
LDA K
INC
STA K                ;          K := K + 1;
JMP WhileAnfang    ; END; (* WHILE *)
WhileEnde
END                  ; End. (* Hauptprogramm *)

; (* ----- *)
Fibo
N          EQUAL 3    ;          N          BP + 3
;          RSA       BP + 2
Ablage    EQUAL 1    ;          Ablage    BP + 1
; BEGIN
PSH
SPBP
LDAB N
JNZ Else1          ; IF N = 0 THEN
LDA Null          ;          BEGIN
STAB N            ;          Fibo := 0;
JMP EndIf1        ;          END
Else1
LDAB N
SUB Eins
JNZ Else2          ;          IF (N = 1) THEN
LDA Eins          ;          BEGIN
STAB N            ;          Fibo := 1;
JMP EndIf2        ;          END
Else2
LDAB N            ;          ELSE BEGIN
DEC              ;          Berechne N - 1
PSH
JSR Fibo          ;          Aufruf
POP
SPBP
STAB Ablage      ;          Ablage := Fibo (N-1)
LDAB N
SUB Zwei          ;          Berechne N - 2
PSH
JSR Fibo          ;          Aufruf
POP              ;          Accu := Fibo (N-2)

```

```

                SPBP
                ADDB Ablage           ;           Accu := Accu + Ablage
                STAB N                 ;           Fibo := Fibo (N - Eins) +
                                        ;           Fibo (N - Zwei);
        EndIf2           ;           END
EndIf1
                POP                   ;   (* Platz fuer Ablage wieder frei *)
                RTN                   ;   END; (* Fibo *)

```

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.

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 für die CPU-Simulation ist in der Tabelle auf der folgenden Seite dargestellt. In dieser Form kann sie den Schülern als Arbeitsblatt vorgelegt werden.

- 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:

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

7.2 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),
 - *sequentiellen 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 Betriebssystems 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 sequentiellen 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).

7.3 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 vorhandenen 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.

7.4 Grenzen des Modells

Wie jedes Modell hat auch dieses Rechnermodell seine Grenzen. Verschiedene anspruchsvolle Adressierungstechniken sind nicht möglich, ebensowenig die Arbeit mit Pointern, obwohl die Adressierung mit dem BP solche Denkweisen fördert. Da die Schüler aber nicht zu Maschinensprache-Programmierern 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, da die Informatik sehr viele interessantere Themenbereiche zu bieten hat. 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. Pos 1
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 einfgn	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 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)

10 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 zu arbeiten.

11 Sonstiges

Bei Rückfragen bzw. Verbesserungsvorschlägen wenden Sie sich bitte an:

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