TYPE STRUDEUT.WI

Ein Precompiler fuer effiziente

Assemblerprogrammierung

von Rolf-Dieter Klein

9.4.1983

Strukta ist ein Precompiler

geschrieben von Dipl. Ing. Rolf-Dieter Klein

Muenchen

Release 1.6

I N H A L T S V E R Z E I C H N I S

B I L D V E R Z E I C H N I S

```
0002'   DA 0006'        JP C,.L37
0005'   00              nop
                    ;1  endif
0006'               .L37:
```

..@F
Bild 3-22: IF Einfache Form
..@I IF,Einfache Form

Die Anweisungssequenz zwischen IF und ENDIF wird genau dann ausgefuehrt wenn die bei IF angegebene Bedingung erfuellt ist. Im anderen Fall wird hinter die ENDIF-Anweisung gesprungen. Nun gibt es auch noch eine andere wichtige Form dieser Anweisung, naemlich wenn im FALSE-Fall auch eine Sequenz von Anweisungen durchlaufen werden soll.

```
if a=0
 nop
else
 nop
endif
```

```
.pa
```

```
                    ;1  if a=0
0000'   B7              OR A
0001'   C2 0008'        JP NZ,.L38
0004'   00              nop
                    ;1  else
0005'   C3 0009'        JP .L39
0008'               .L38:
0008'   00              nop
                    ;1  endif
0009'               .L39:
```

..@F
Bild 3-23: IF mit ELSE-Teil
..@I IF,ELSE-Teil

Ist die Bedingung bei IF erfuellt, so werden die Anweisungen zwischen IF und ELSE ausgefuehrt, ist sie nicht erfuellt, so werden die Anweisungen zwischen ELSE und ENDIF ausgefuehrt.

Haeufig muss in Programmen eine Fallunterscheidung getroffen werden. Dazu wird in PASCAL z.B. die CASE-Anweisung verwendet. Eine andere Moeglichkeit ist die Verwendung von ELSEIF.

Beispiel:

WHILE-Schleifen koennen auch verschachtelt werden:

```
while a=0
 while b=0
  nop
 endwhile
 nop
endwhile
```

```
                        ;1  while a=0
  0000'                 .L33:
  0000'   B7                OR A
  0001'   C2 0011'          JP NZ,.L34
                        ;2  while b=0
  0004'                 .L35:
  0004'   78                LD A,b
  0005'   B7                OR A
  0006'   C2 000D'          JP NZ,.L36
  0009'   00                nop
                        ;2  endwhile
  000A'   C3 0004'          JP .L35
  000D'                 .L36:
  000D'   00                nop
                        ;1  endwhile
  000E'   C3 0000'          JP .L33
  0011'                 .L34:
```

```
..@F
Bild  3-21: WHILE Verschachtelung
..@I WHILE,Verschachtelung
```

```
.pa

..@B
3.6.  Die IF-Anweisung

..@M IF
..@M ELSE
..@M ELSEIF
..@M ENDIF
```

In keiner Sprache darf die bedingte Anweisung fehlen mit der Programmteile alternativ ausgefuehrt werden koennen.

Die IF-Anweisung kann mehrere Formen haben. Die einfachste zeigt das naechste Bild:

```
if b<=(hl)
 nop
endif
```

```
                        ;1  if b<=(hl)
  0000'   7E                LD A,(hl)
  0001'   B8                CP b
```

und beim STRU8080 12K + ca. 10K Variablenraum. Dieser Platz ist unabhaengig von der Laenge der Source und es gibt dafuer keine Begrenzung, da Strukta keine Symboltabelle verwalten muss.

Es gibt auch ein paar Optionen die dem Dateinamen durch ein Leerzeichen getrennt folgen koennen und durch das Zeichen "/" eingeleitet werden.

/C entfernen alle Kommentare bei Erzeugung der Assemblerdatei

/L Ausgabe des Assemblerlistings auf die Konsole

/P Ausgabe des Assemblerlistings auf den Drucker

Aufruf von Strukta

Beispiel

```
STRUZ80 TEST /C
```

TEST.STR ist die Quell-Datei. Es wird eine Datei TEST.MAC erzeugt in der zusaetzlich alle Kommentare, auch die automatisch erzeugten, entfernt sind.

STRUKTA laesst sich auch ohne Parameter aufrufen, dann meldet sich Strukta wie folgt:

```
B>c:struz80

------------------------------
Strukta   Compiler Version 1.6
by Rolf-Dieter Klein  (C) 1983
Munic              West Germany
Last change           10.2.1983
Z 8 0   V E R S I O N
------------------------------

Please enter Struktalines End = CTRL Z
```

Bild 2-1: Strukta Aufruf

Es lassen sich dann direkt ueber die Konsole Strukta-Befehle eingeben und die Uebersetzung wird auf der Konsole ausgegeben.

Damit ist es moeglich die Sprache zu lernen und die Codegenerierung verstehen zu lernen.

Beispiel:

```
B>c:struz80

------------------------------
Strukta   Compiler Version 1.6
by Rolf-Dieter Klein  (C) 1983
Munic              West Germany
Last change          10.2.1983
Z 8 0   V E R S I O N
------------------------------

Please enter Struktalines End = CTRL Z



if not[a=0 or not b=1 and c=0]
;1      if not[a=0 or not b=1 and c=0]
       OR A
       JP Z,.L1
       LD A,b
       CP 1
       JP Z,.L2
       LD A,c
       OR A
       JP Z,.L1
.L2:


endif
;1      endif
.L1:


4 line(s) were read
14 line(s) were generated
0 error(s) were detected
 End of Compilation
```

..@F
Bild 2-2: Direkt Eingabe im Interactiven Mode

Die fettgeschriebenen Teile wurden dabei von Hand eingegeben, der Rest wurde vom Compiler erzeugt. Mit CTRL-Z wird die Eingabe beendet.

..@M Handeingabe
..@M Interactiver Betrieb

.pa

..@A
3. Befehlssatz des Compilers

..@B
3.1. Arithmetische und logische Vergleiche

..@C
3.1.1. Flag-Abfrage

Strukta ermoeglicht eine flexible Angabe von arithmetischen und logischen Verknuepfungen in den Bedingungsausdruecken. Zunaechsteinmal koennen natuerlich alle Z80 (8080) Flags abgefragt werden.

Z, NZ, C, NC, P, M, PE, PO.

```
..@I Z
..@I NZ
..@I C
..@I NC
..@I P
..@I M
..@I PE
..@I PO
```

Ferner gibt es die beiden Bedingungen OV und NV, die aber vor allem im Compiler selbst verwendet werden und die eine Kombination von C und Z sind.

Beispiel eines einfachen Ausdrucks mit einer IF-Anweisung

```
IF Z
 ld a,c
ENDIF
```

bedeutet die Anweisung ld a,c (oder beim STRU8080 mit mov a,c) wird genau dann ausgefuehrt wenn das Flag Z (Zero-Flag) gesetzt war. Sonst wird die Anweisung umgangen.

Als Assemblercode wird dabei eine Sequenz

```
      JP NZ,.L1
       ld a,c
:     .L1
```

beim STRUZ80 abgelegt oder beim STRU8080

```
      JNZ .L1
       mov a,c
      .L1:
```

wobei natuerlich im Quelltext auch mov a,c angegeben worden sein musste. Der Compiler copiert einfach allen Text, den er ~~Assemblersprache~~ und daher muss er die genaue Syntax des verwendeten Assemblers auch nicht kennen. Der Compiler erzeugt automatische Sprungmarken, die mit .L anfangen, gefolgt von einer Zahl.

..@C

3.1.2. Logische Verknuepfungen

Neben den Flags gibt es auch logische Verknuepfungen wie AND, OR und NOT.

..@I AND
..@I OR
..@I NOT

Um sie zu verschachteln wird die eckige Klammer verwendet,

Also z.B. Z AND C

oder komplizierter NOT[C OR NZ] AND PE OR M

Daraus generiert der Compiler eine Sequenz von bedingten Spruengen, wobei alle Negationen durch das De-Morgan-Gesetz auf die Bedingungen umgesetzt wird, z.B. ergibt sich intern

NOT[C OR NZ] AND PE OR M

entspricht

[NC AND Z AND PE] OR M

was direkt in eine Sprungsequenz umgesetzt werden kann. Die eckige Klammer kann uebrigends in der letzten Zeile entfallen, AND hat eine groessere Bindung als OR.

Codeerzeugung bei der Anweisung EXITIF die zum verlassen von LOOP-Schleifen oder Unterprogrammen verwendet werden kann. Das Beispiel ist mit dem STRU8080 erzeugt.

```
.pa

exitif not[c or nz] and pe or m
 ;1       exitif not[c or nz] and pe or m
          JC .L1
          JNZ .L1
          RPE
 .L1:
          RM

exitif nc and z and pe or m
 ;1       exitif nc and z and pe or m
          JC .L2
```

```
        JNZ .L2
        RPE
 .L2:
        RM
```

..@F
Bild 3-1: Uebersetzung von Flag-Bedingungen
..@I Codeerzeugung

..@C
3.1.3. Mengenabfragen

Ein Feature, das der Sprache PASCAL entlehnt wurd sind die Mengenabfragen. Sehr oft ist es noetig einen Bereich zu testen und je nachdem ob ein Wert in einem Bereichvorkommt eine Anweisungssequenz auszufuehren oder nicht. Dazu dient die Operation IN.

..@M IN
Die Syntax dafuer lautet

register IN [aufzaehlung]

wobei

aufzaehlung ::= bereich

einzelelement

aufzaehlung , aufzaehlung

sein kann

mit

bereich ::= wert1..wert2

einzelelement

::= wert

wert ::= registername

konstante

Nun in etwas verstaendlicher Form mit Beispielen:

a IN ['A'..'Z']

ist genau dann erfuellt wenn der Inhalt des Registers A ein Wert des Bereichs 'A' bis 'Z' annimmt.

oder

b IN [1,5,sieben]

ist genau dann wahr wenn der Inhalt von Register B den WErt 1 oder den Wert 5 oder den Wert 7 annimmmt.

oder

(hl) IN ['a'..'f','A'..'F','0'..'9']

ist genau dann wahr, wenn der Inhalt der durch HL adressierten Speicherzelle eine HEX-Ziffer ist.

Beispiel mit STRUZ80:

```
 if (hl) in ['A'..'F','0'..'9']
 ;2       if (hl) in ['A'..'F','0'..'9']
          LD A,(hl)
          CP 'F'+1
          JR NC,.L4
          CP 'A'+0
          JP NC,.L3
 .L4:
          CP '9'+1
          JP NC,.L2
          CP '0'+0
          JP C,.L2
 .L3:


 endif
 ;2       endif
 .L2:
 ;1
 .L1:
```

..@F
Bild 3-2: Uebersetzung von Mengenoperationen

..@C
3.1.4. Vergleiche

Vorher hatten wir schon kurz Registerangaben verwendet. Alle Z80 Register (bzw. 8080 Register beim STRU8080) lassen sich angeben:

A, B, C, D, E, H, L, (HL), (IX+wert), (IY+wert)

als Wert kann eine Konstante verwendet werden.

und bedingt auch Doppelregister

BC, DE, HL, IX, IY, SP

Ferner lassen sich in Vergleichen auch Konstanten angeben, entweder direkte Zahlen binaer, oktal, dezimal oder hex oder in ASCII-Darstellung , oder Namen, die z.B. mit EQU definiert wurden. Die Konstanten koennen auch mit + oder - verknuepft sein, duerfen aber kein einzelnen Vorzeichen tragen (Anstelle -6 muss man 0-6 schreiben).

..@I Vergleiche
..@I Register
..@I Zahlenbereich
Alle Vergleiche beziehen sich auf den positiven Zahlenbereich von 0 bis 255 bei Einzelregister, daher auch obige Einschraenkung um Fehlern vorzubeugen. Doppelregister koennen nur auf Gleich - oder Ungleichheit verglichen werden, dabei sind nur die Registerpaar BC, DE und HL erlaubt. Vergleiche mit mehreren Registern koennen ansonsten mit AND und OR formuliert werden. Bei allen Operationen die der Compiler erzeugt wird maximal der Akku vom Compiler veraendert.

Als Vergleiche sind folgende Operationen moeglich:

< <= = >= > und <>

Beispiel:

A < B (HL)='A' (IX+9)>=(IY+10) alpha=0 (HL)<=L

oder auch

A <= maxchar-1

..@M Vergleiche

.pa

Beispiel mit STRUZ80

```
 if a=0
 ;1       if a=0
          OR A
          JP NZ,.L1

 endif
 ;1       endif
 .L1:

 if (hl)<=(ix+konst)
 ;1       if (hl)<=(ix+konst)
          LD A,(ix+konst)
          CP (hl)
          JP C,.L2

 endif
 ;1       endif
 .L2:

 if (hl) >= a
 ;1       if (hl) >= a
          CP (hl)
          JR Z,.L4
          JP NC,.L3
 .L4:
```

```
 endif
 ;1       endif
 .L3:

..@F
```

Bild 3-3: Uebersetzung von Vergleichen

.pa

Alle Verknuepfungen lassen sich beliebig zum komplexen Ausdruecken kombinieren

Beispiel mit STRU8080

```
 if a=0 and not b in ['A'..'Z'] or c<=9 and not[hl=0 or de<>0]
;1       if a=0 and not b in ['A'..'Z'] or c<=9 and not[hl=0 or de<>0]
          ORA A
          JNZ .L3
          MOV A,b
          CPI 'Z'+1
          JNC .L2
          CPI 'A'+0
          JC .L2
 .L4:
 .L3:
          MVI A,9
          CMP c
          JC .L1
          MOV A,L
          ORA H
          JZ .L1
          MOV A,E
          ORA D
          JNZ .L1
 .L2:

 endif
 ;1       endif
 .L1:

..@F
```

Bild 3-4: Uebersetzung von zusammengesetzten Ausdruecken

.pa

..@B

3.2. Die DO-Anweisung

..@M DO-Anweisung

Ein sehr haeufige Konstruktion bei Programmen ist die einfache Zaehlschleife. In Strukta laesst sie sich mit DO und ENDDO bilden. DO erhaelt dazu die Angabe eines Registers oder eines

Registerpaares und optional eine Konstante fuer die Initialisierung der Zaehlschleife. Das Register wird dazu mit der Konstanten, falls sie angegeben wurde initialisiert und dann bei der Instruktion ENDDO um eins veringert. Die Schleife wird n mal durchlaufen, wenn n der Wert der Konstanten war.

Beispiel:

```
do b,5
 nop    ;beliebige Befehle des Assemblers
enddo
```

```
                    ;1  do b,5
  0000'   06 05         LD b,5
  0002'                 .L1:
  0002'   00            nop     ;beliebige Befehle des Assemblers
                    ;1  enddo
  0003'   05            DEC B
  0004'   C2 0002'      JP NZ,.L1
```

..@F
Bild 3-5: Zaehlschleife mit Einzelregister
..@I Zaehlschleife,Einzelregister

Die Schleife wird fuenf Mal durchlaufen. NOP steht fuer irgend eine Befehlssequenz, wobei natuerlich dort auch weitere geschachtelte DO-Schleifen stehen duerfen. Wird das Register B in der Schleife modifiziert, so muss es fuer den korrekten Ablauf mit einer Sequenz PUSH BC .. POP BC in der Schleife gerettet werden.

Auch Doppelregister koennen als Zaehler verwendet werden:

```
.pa

do hl,1000
 nop    ;beliebeige Befehle des Assemblers
enddo
```

```
                    ;1  do hl,1000
  0000'   21 03E8       LD hl,1000
  0003'                 .L2:
  0003'   00            nop     ;beliebeige Befehle des Assemblers
                    ;1  enddo
  0004'   2B            DEC HL
  0005'   7C            LD A,H
  0006'   B5            OR L
  0007'   C2 0003'      JP NZ,.L2
```

..@F
Bild 3-6: Zaehlschleife mit Doppelregister
..@I Zaehlschleife,Doppelregister

Alle Register A, B, C, D, E, H, L und die Doppelregister BC, DE, HL, IX und IY koennen als Zaehlvariable verwendet werden. SP ist nicht zulaessig, da dies zu instabilen Konstruktionen fuehren kann.

Wird die Konstante weggelassen, so wird das Register am Anfang der Schleife nicht initialisiert und es wird der folgende Code erzeugt:

```
do c
 nop
enddo
```

```
                    ;1  do c
  0000'             .L3:
  0000'   00            nop
                    ;1  enddo
  0001'   0D            DEC C
  0002'   C2 0000'      JP NZ,.L3
```

..@F
Bild 3-7: Zaehlschleife ohne Initialisierung
..@I Zaehlschleife,dynamisch

Damit lassen sich Schleifen mit programmierbarer Anzahl von Durchlaeufen verwirklichen.

Als Beispiel fuer das Verschachteln von DO-Anweisungen dient das folgende Programmstueck:

```
do de,1000
 nop
 do hl,5000
  nop
 enddo
 nop
enddo
```

```
                    ;1  do de,1000
  0000'   11 03E8       LD de,1000
  0003'             .L4:
  0003'   00            nop
                    ;2  do hl,5000
  0004'   21 1388       LD hl,5000
  0007'             .L5:
  0007'   00            nop
```

```
                    ;2  enddo
  0008'   2B            DEC HL
  0009'   7C            LD A,H
  000A'   B5            OR L
  000B'   C2 0007'      JP NZ,.L5
  000E'   00            nop
                    ;1  enddo
  000F'   1B            DEC DE
  0010'   7A            LD A,D
  0011'   B3            OR E
  0012'   C2 0003'      JP NZ,.L4
```

..@F
Bild 3-8: Zaehlschleife verschachtelt
..@I Zaehlschleife,verschachtelt

Mit der DO-Anweisung lassen sich auch Verzoegerungsschleifen realisieren:

.pa

```
; warteschleife

delay equ 1000

; 24 t-Zyclen pro Durchlauf
do bc,delay
enddo
;
                                ; warteschleife

  03E8                          delay equ 1000

                                ; 24 t-Zyclen pro Durchlauf
                    ;1  do bc,delay
  0000'   01 03E8       LD bc,delay
  0003'                 .L6:
                    ;1  enddo
  0003'   0B            DEC BC
  0004'   78            LD A,B
  0005'   B1            OR C
  0006'   C2 0003'      JP NZ,.L6
                                                    ;
```

..@F
Bild 3-9: Warteschleife
..@I Warteschleife

.pa

..@B
3.3. Die LOOP-Anweisung

Die LOOP-Anweisung wird immer dann fuer Schleifen verwendet,

wenn es mehrere Abbruchkriterien an unterschiedlichen Stellen in einer Schleife gibt. Bei der LOOP-Anweisung wird die Schleife mit dem EXIT, EXITIF-Befehl verlassen. Dabei wird um die Sequenz klein zu halten vom bedingten RETURN-Befehl Gebrauch gemacht. Dazu wird die Adresse die nach der ENDLOOP-Anweisung folgt auf den Stack gebracht. Ein Return verlaesst dann die Schleife. Beim Verschachteln dieser Anweisung wird durch den EXIT, EXITIF-Befehl immer die innerste Schleife verlassen.

```
..@M EXIT
..@M EXITIF
..@M LOOP
..@M ENDLOOP
```

Einfaches Beispiel:

```
loop
 nop
 exitif a=0
 nop
endloop
```

```
                        ;1  loop
  0000'   E5                    PUSH HL
  0001'   21 000C'              LD HL,.L7
  0004'   E3                    EX (SP),HL
  0005'                 .L8:
  0005'   00                    nop
                        ;2  exitif a=0
  0006'   B7                    OR A
  0007'   C8                    RET Z
  0008'   00                    nop
                        ;1  endloop
  0009'   C3 0005'              JP .L8
  000C'                 .L7:
```

```
..@F
Bild  3-10: LOOP mit einem Ausgang
..@I LOOP,ein Ausgang
```

Ein komplexeres Beispiel ist in der naechsten Abbildung dargestellt:

loop

```
 exitif hl=0 or bc=0
 nop
 exitif a=0 and l<=9
endloop
```

```
                    ;1  loop
  0000'   E5            PUSH HL
  0001'   21 0017'      LD HL,.L9
  0004'   E3            EX (SP),HL
  0005'               .L10:
                      ;2  exitif hl=0 or bc=0
  0005'   7D            LD A,L
  0006'   B4            OR H
  0007'   C8            RET Z
  0008'   79            LD A,C
  0009'   B0            OR B
  000A'   C8            RET Z
  000B'   00            nop
                      ;2  exitif a=0 and l<=9
  000C'   B7            OR A
  000D'   C2 0014'      JP NZ,.L11
  0010'   3E 09         LD A,9
  0012'   BD            CP 1
  0013'   D0            RET NC
  0014'               .L11:
                      ;1  endloop
  0014'   C3 0005'      JP .L10
  0017'               .L9:
```

..@F
Bild 3-11: LOOP mit zwei Ausgaengen

..@I LOOP,zwei Ausgaenge

Ein praktisches Beispiel zeigt das naechste Beispiel. Es sollen solange Zeichen eingelesen werden bis das binaere Zeichen 00 als Eingabe gegeben wurde. Die Routine CI sei ein festes Monitorunterprogramm.

```
.pa

; definition an irgend einer Stelle des Programms
;

ci:
 jp 0f003h      ;z.B. Unterprogramm


; an anderer Stelle sei die Schleife

loop
 call ci
 exitif a=0
 ld (hl),a
 inc hl
endloop
```

```
0000'                ci:
0000'   C3 F003          jp 0f003h          ;z.B. Unterprogramm


                     ;1  loop
0000'   E5               PUSH HL
0001'   21 000F'         LD HL,.L12
0004'   E3               EX (SP),HL
0005'                .L13:
0005'   CD 0000'         call ci
                     ;2  exitif a=0
0008'   B7               OR A
0009'   C8               RET Z
000A'   77               ld (hl),a
000B'   23               inc hl
                     ;1  endloop
000C'   C3 0005'         JP .L13
000F'                .L12:
```

..@F
Bild 3-12: LOOP Zeicheneinleseprogramm
..@I LOOP,Einleseprogramm

Die EXIT und EXITIF-Befehle koennen auch zum Verlassen von Unterprogrammen verwendet werden.

.pa
..@B
3.4. Die REPEAT-Anweisung

Auch eine Konstruktion die in der Programmierpraxis sehr haeufig vorkommt. Eine Schleife wird sooft durchlaufen bis eine Bedingung erfuellt ist. Dabei wird beim REPEAT die Schleife mindestens einmal durchlaufen.

..@M REPEAT
..@M UNTIL

Beispiel, Die Schleife soll solange durchlaufen werden bis der Inhalt des Akkumulators 0 ist.

```
repeat
 nop
until a=0
```

```
                     ;1  repeat
0000'                .L14:
0000'   00               nop
                     ;1  until a=0
0001'   B7               OR A
0002'   C2 0000'         JP NZ,.L14
```

..@F
Bild 3-13: REPEAT einfaches Beispiel

Ein anderes Beispiel, es soll im Speicher ein Buchstabe A bis Z gefunden werden, oder die binaere 0.

```
repeat
 ld a,(hl)
 inc hl
until a in [0,'A'..'Z']
```

```
                    ;1  repeat
  0000'             .L15:
  0000'   7E            ld a,(hl)
  0001'   23            inc hl
                    ;1  until a in [0,'A'..'Z']
  0002'   FE 00         CP 0
  0004'   CA 0011'      JP Z,.L16
  0007'   FE 5B         CP 'Z'+1
  0009'   D2 0000'      JP NC,.L15
  000C'   FE 41         CP 'A'+0
  000E'   DA 0000'      JP C,.L15
  0011'             .L16:
```

..@F
Bild 3-14: REPEAT Suchschleife
..@I REPEAT,Suchschleife

Hier noch ein Beispiel mit einem Vergleich. Es werden solange Zeichen eingelesen bis ein Zeichen im Bereich 80h bis ffh liegt. Dies ist auch ein Beispiel um Vergleiche mit negativen Zahlen durchfuehren zu koennen, denn der Vergleich A<0 ist nie erfuellt, da der Compiler nur Code fuer positive Zahlen erzeugt. Mit dem Vergleich a>=80h wird aber dennoch auf <0 abgeprueft, wenn die Zahl als solche interpretiert werden kann.

```
repeat
 call ci
 ld (hl),a
until a>=80h
```

```
                    ;1  repeat
  0000'             .L17:
  0000'   CD 0000'      call ci
  0003'   77            ld (hl),a
```

```
                    ;1  until a>=80h
  0004'   FE 80         CP 80h
  0006'   DA 0000'      JP C,.L17


..@F
Bild  3-15: REPEAT Vergleich auf kleiner 0
..@I REPEAT,negative Zahl
```

Die Anweisung kann natuerlich auch geschachtelt werden:

```
repeat
 nop
 repeat
  nop
  repeat
   nop
  until a=0
 until b in [1,2,3]
 nop
until a<=1
.pa

                    ;1  repeat
  0000'             .L18:
  0000'   00            nop
                    ;2  repeat
  0001'             .L19:
  0001'   00            nop
                    ;3  repeat
  0002'             .L20:
  0002'   00            nop
                    ;3  until a=0
  0003'   B7            OR A
  0004'   C2 0002'      JP NZ,.L20
                    ;2  until b in [1,2,3]
  0007'   78            LD A,b
  0008'   FE 01         CP 1
  000A'   CA 0017'      JP Z,.L21
  000D'   FE 02         CP 2
  000F'   CA 0017'      JP Z,.L21
  0012'   FE 03         CP 3
  0014'   C2 0001'      JP NZ,.L19
  0017'             .L21:
  0017'   00            nop
                    ;1  until a<=1
  0018'   BD            CP 1
  0019'   28 03         JR Z,.L22
  001B'   D2 0000'      JP NC,.L18
  001E'             .L22:


..@F
Bild  3-16: REPEAT Verschachtelt
..@I REPEAT,Verschachtelt




.pa

..@B
3.5.  Die WHILE-Anweisung
```

```
..@M WHILE
..@M ENDWHILE
```
Die WHILE-Anweisung ist aehnlich zu der REPEAT-Konstruktion, nur das die Bedingung vor dem Durchlaufen der Schleife abgefragt wird. Ist die Bedingung nicht erfuellt, so wird die Schleife nicht durchlaufen.

Einfaches Beispiel:

```
while a=8
 nop
endwhile
```

```
                    ;1  while a=8
0000'               .L23:
  0000'   FE 08         CP 8
  0002'   C2 0009'      JP NZ,.L24
  0005'   00            nop
                    ;1  endwhile
  0006'   C3 0000'      JP .L23
  0009'               .L24:
```

```
..@F
```
Bild 3-17: WHILE Einfaches Beispiel
```
..@I WHILE,Einfaches Beispiel
```

Eine praktische Anwendung zeigt das naechste Bild. Zeichen sollen aus einem Textbuffer ausgegeben werden. Das Ende der Zeichenkette ist durch die binaere Null dargestellt. Im Registerpaar HL steht die Anfangsadresse der Zeichenkette. Dann erfuellt das folgende Programm die gestellte Aufgabe:

```
.pa
```

```
while (hl)<>0
 ld c,(hl)
 call 0f009h     ;co
 inc hl
endwhile
```

```
                    ;1  while (hl)<>0
0000'               .L25:
  0000'   7E            LD A,(hl)
  0001'   B7            OR A
  0002'   CA 000D'      JP Z,.L26
  0005'   4E            ld c,(hl)
  0006'   CD F009       call 0f009h     ;co
  0009'   23            inc hl
                    ;1  endwhile
  000A'   C3 0000'      JP .L25
  000D'               .L26:
```

..@F
Bild 3-18: WHILE Zeichenausgabe
..@I WHILE,Zeichenausgabe

Nun sollen Zeichen solange eingelesen werden bis kein Leerzeichen mehr auftritt. Im Akku sei das erste Zeichen schon vorhanden:

```
while a=' '
 call ci
endwhile
```

```
                    ;1  while a=' '
  0000'             .L27:
  0000'   FE 20         CP ' '
  0002'   C2 000B'      JP NZ,.L28
  0005'   CD 0000'      call ci
                    ;1  endwhile
  0008'   C3 0000'      JP .L27
  000B'             .L28:
```

..@F
Bild 3-19: WHILE Leerzeichen ueberlesen
..@I WHILE,Leerzeichen ueberlesen

Es sollen Zeichen eingelesen werden, solange es HEX-Zahlen sind:

```
call ci
while a in ['A'..'F','0'..'9']
 call ci
endwhile
.pa
```

```
  0000'   CD 0000'      call ci
                    ;1  while a in ['A'..'F','0'..'9']
  0003'             .L29:
  0003'   FE 47         CP 'F'+1
  0005'   30 05         JR NC,.L32
  0007'   FE 41         CP 'A'+0
  0009'   D2 0016'      JP NC,.L31
  000C'             .L32:
  000C'   FE 3A         CP '9'+1
  000E'   D2 001C'      JP NC,.L30
  0011'   FE 30         CP '0'+0
  0013'   DA 001C'      JP C,.L30
  0016'             .L31:
  0016'   CD 0000'      call ci
                    ;1  endwhile
  0019'   C3 0003'      JP .L29
  001C'             .L30:
```

..@F
Bild 3-20: WHILE HEX-Zahlen einlesen
..@I WHILE,Hex-Zahlen

Zeilen pro Minute und ist damit schneller als der nachfolgende Assembler-Lauf.

Ablauf einer Uebersetzung:

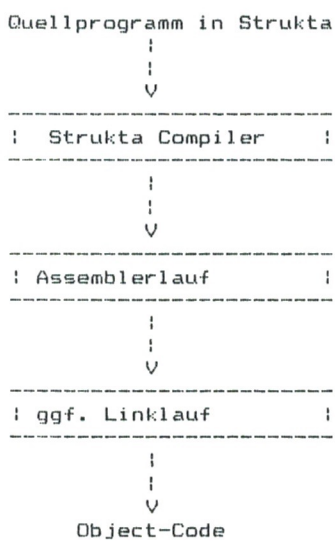


Bild 1-1: Uebersetzungsschema

Strukta kann durch dieses Verfahren auch mit anderen Programmen aus hoeheren Programmiersprachen gelinkt werden und umgekehrt.

2. Aufruf des Compilers

Der Compiler wird mit

STRUZ80 dateiname

oder STRU8080 dateiname

gestartet. Er liest eine Datei mit der Extension .STR und erzeugt eine Datei die mit .MAC endet.

STRUZ80 ist nur auf einem Z80 ablauffaehig, STRU8080 kann jedoch auch auf einem 8080-System uebersetzen. Der Speicherbedarf ist beim STRUZ80 10K + ca. 10K Variablenraum

..@A
1. Strukta ein Pre-Compiler

Fuer die Entwicklung von Mikrocomputersoftware gibt es zwei unterschiedliche Programmiersysteme, zum einen Assemblerpro gramme zum anderen Programme in einer hoeheren Programmier sprache. Das programmieren in einer hoeheren Sprache ist sehr vorteilhaft, jedoch ist der erzeugte Code, geradee bei der 8-Bit-Generation inefficient und lang, so dass oft zum Assembler gegriffen werden muss wenn es um zeitkritische Aufgaben oder um die Entwicklung von Software fuer Single-Board-Computern geht. Strukta ermoeglicht die Verwendung von struktutierten Konstruktionen aus den hoeheren Programmier sprachen und soll diesen Mangel beheben. Strukta wird als Precompiler gestartet und erzeugt eine Assembler-Source-Datei die dann mit einem konvetionellen Assembler (z.B. M80, MAC etc) uebersetzt werden kann. Strukta achtet nur auf die eigenen Schluesselworte der struktuierten Sprache und kopiert alle anderen Anweisungen einfach, somit lassen sich alle Faehigkeiten des nachfolgenden Assemblers wie linken, Makros etc. ausnuetzen.

Strukta nimmt dem Programmierer die muehevolle Aufgabe sich Labels auszudenken und bedingte Spruenge zu generieren ab, da Strukta ueber Kontrollstrukturen wie IF, ELSE, ELSEIF, ENDIF, REPEAT, UNTIL, DO, ENDDO, LOOP, ENDLOOP besitzt, die auch komplexe Ausdruecke mit Registern und Vergleichen sowie Und, ODER, NICHT und Mengenoperationen auswerten koennen. STRUKTA wurde fuer eine optimierte Code-Erzeugung konstruiert und ist in zwei Versionen, STRU8080 und STRUZ80 verfuegbar. Strukta ist selbst in Strukta geschrieben und wird inclusive Source-Code geliefert. Die Source ist auch noetig um ggf. auf unter schiedliche Assembler angepasst werden zu koennen.

Strukta uebersetzt mit einer Geschwindigkeit von ca. 3000

```
if a='A'
 nop
elseif a='B'
 nop
elseif a in ['C'..'G']
 nop
elseif a='H'
 nop
else
 nop
endif

.pa

                        ;1  if a='A'
  0000'   FE 41             CP 'A'
  0002'   C2 0009'          JP NZ,.L40
  0005'   00                nop
                        ;1  elseif a='B'
  0006'   C3 002A'          JP .L41
  0009'                 .L40:
  0009'   FE 42             CP 'B'
  000B'   C2 0012'          JP NZ,.L42
  000E'   00                nop
                        ;1  elseif a in ['C'..'G']
  000F'   C3 002A'          JP .L41
  0012'                 .L42:
  0012'   FE 48             CP 'G'+1
  0014'   D2 0020'          JP NC,.L43
  0017'   FE 43             CP 'C'+0
  0019'   DA 0020'          JP C,.L43
  001C'                 .L44:
  001C'   00                nop
                        ;1  elseif a='H'
  001D'   C3 002A'          JP .L41
  0020'                 .L43:
  0020'   FE 48             CP 'H'
  0022'   C2 0029'          JP NZ,.L45
  0025'   00                nop
                        ;1  else
  0026'   C3 002A'          JP .L46
  0029'                 .L45:
  0029'   00                nop
        ;1  endif
  002A'                 .L46:
  002A'                 .L41:

..@F
Bild  3-24: IF mit ELSEIF
..@I IF,ELSEIF
```

Die IF-Anweisung kann wie alle anderen Anweisungen auch geschachtelt werden:

```
if a=0
 if b=9
  nop
 else
  nop
 endif
 nop
endif
```

```
.pa

                        ;1  if a=0
  0000'   B7                OR A
  0001'   C2 0010'          JP NZ,.L47
                        ;2  if b=9
  0004'   78                LD A,b
  0005'   FE 09             CP 9
  0007'   C2 000E'          JP NZ,.L48
  000A'   00                nop
                        ;2  else
  000B'   C3 000F'          JP .L49
  000E'                 .L48:
  000E'   00                nop
                        ;2  endif
  000F'                 .L49:
  000F'   00                nop
                        ;1  endif
  0010'                 .L47:


..@F
Bild  3-25: IF Geschachtelte Konstruktion
..@I IF,Schachtelung


Natuerlich   koennen   alle   Konstruktionen   auch   gemischt

verschachtelt werden:


repeat
 while a=0
  nop
  if a=0
   nop
   if b=0
    nop
   endif
  else
   nop
  endif
 endwhile
 nop
until (hl)<=(ix+0)

.pa


                        ;1  repeat
  0010'                 .L50:
                        ;2  while a=0
  0010'                 .L51:
  0010'   B7                OR A
  0011'   C2 0027'          JP NZ,.L52
  0014'   00                nop
                        ;3  if a=0
  0015'   B7                OR A
  0016'   C2 0023'          JP NZ,.L53
  0019'   00                nop
                       ;4   if b=0
  001A'   78                LD A,b
  001B'   B7                OR A
```

```
001C'   C2 0020'        JP NZ,.L54
001F'   00              nop
                    ;4  endif
0020'                   .L54:
                    ;3  else
0020'   C3 0024'        JP .L55
0023'                   .L53:
0023'   00              nop
                    ;3  endif
0024'                   .L55:
                    ;2  endwhile
0024'   C3 0010'        JP .L51
0027'                   .L52:
0027'   00              nop
                    ;1  until (hl)<=(ix+0)
0028'   DD 7E 00        LD A,(ix+0)
002B'   BE              CP (hl)
002C'   DA 0010'        JP C,.L50
```

..@F
Bild 3-26: Verschachteln von verschiedenen Anweisungen

.pa

..@A
4. Wirkungsweise des Compilers

In diesem Kapitel sei noch kurz etwas ueber die Entstehungsgeschichte des Compilers gesagt. STRUZ80 ist in STRUZ80 geschrieben und STRU8080 in STRU8080.

Man wird sich dann sofort fragen, wie das Moeglich ist einen Compiler in der Sprache zu schreiben, die er uebersetzten soll, da zuerst der Compiler ja noch nicht da war. Da gibt es zwei Moeglichkeiten, zum einen kann die erste Uebersetzung von Hand vorgenommen werden, was aber sehr Fehlertraechtig ist, vom ianderen wieddeden Compiler zu aehnlichen Sprache geschrieben. So wurde es auch hier gemacht. Als erstes existierte eine STRUZ80 Version in PASCAL (MT+). Mit dieser Version konnte dann die STRUKTA-Version uebersetzt werden. Dabei wurde in der PASCAL nurd as noetigste implementiert und auf eine Fehlerbehandlung weitgehenst verzichtet. Diese Version war uebersetzt 31K lang und uebersetzte sehr langsam. Die entgueltige Version STRUZ80 ist nun 10K gross und die STRU8080 Version 12K, da sie auch nur mit 8080-Mnemonics geschrieben wurde. Die STRU8080-Version wurde nach der STRUZ80 in zwei Phasen gebaut, den hier bestand

natuerlich ein aehnliches Problem.

Der Compiler bestitzt um die Diskettenzugriffe klein zu halten zwei 4K Buffer, die fuer die Quelle und den Objekt verwendet werden. Nur dadurch ist es auch moeglich eine Uebersetzungsgeschwindigkeit von 3000 Zeilen/Minute zu erreichen.

Soll STRUKTA fuer einen Assembler neu angepasst werden, z.B. um kollidierende Bedingungsnamen IF fuer STRUKTA und IF im Assembler, so muss STRUKTA neu uebersetzt werden. Dazu wird zunaechst

STRUZ80 STRUZ80

aufgerufen, dann

M80 =STRUZ80

und mit

L80 STRUZ80,STRUZ80/N/E

ist der neue Compiler erzeugt. Doch VORSICHT. Wenn die Schluesselworte veraendert wurden kann der neue Compiler sicht zunaechst nicht neu uebersetzten. Dazu muessen in der Quelle auch alle Schluesselwoerter geaendert werden.

Dann kann ggf. ein neuer Durchlauf versucht werden. Der Original-Compiler muss in jedem Fall aufgehoben werden.

Nach dem zweiten Lauf kann nun der neue Compiler Object-Code mit dem alten verglichen werden, dieser muss, wenn nur die Schluesselworte geaendert wurden gleich sein.

Der Compiler arbeitet Recursiv als RECURSIV-DECEND-Compiler.

Die Syntax ist nicht kontextfrei, da Registernamen und Konventionen nicht immer sofort als solche erkennbar sind. Die skommt durch die Mischung von Assembler und hoeherer Programmiersprache.

Im Compiler gibt es deshalb einen kleinen Scanner, der vorausschauend die Syntax erkennt und danach wird erst nochmals die Syntax gescannt, dann aber mit dem Parserlauf zur

Codeerzeugung.

Durch das Verfahren der Precompilierung koennen Strukta-Sprachelemente nicht in Makroexpansionen verwendet werden, da zuerst der Struktateil expandiert wird und danach erst im Assemblerlauf der Makro expandiert wird. Dadurch gibt es Probleme bei den verwendeten Marken im Makro. Manche Assembler legen automatisch lokale Marken an, dann geht das Verfahren.

Im anderen Fall muesste ein getrennter Makroprozessor verwendet werden, der vor dem Struktalauf alle Makros aufloest und dann Strukta.
Ansonsten ist dies keine grosse Einschraenkung in den Makros kein Strukta zu verwenden, wie es z.B. im Source-Code des Compiler selbst auch getan wurde.
Bedingte Anweisungen fuer bedingte Assemblierung kollidieren manchmal mit denen Anweisungen in Strukta. Da gibt es neben der Loesung durch neucompilieren des Strukta-Compilers auch noch die Moeglichkeit z.B. alle Assemblerbedingungen mit einem Punkt zu kennzeichnen und anschliessend aus der erzeugen Assemblersourcefile wieder zu entfernen.

.pa

..@A
5. Fehlerbehandlung

Der Compiler gibt eine Meldung aus, sobald er einen Fehler erkennt. Dieser Fehler wird dann auch in der Assemblerdatei markiert. Erkennt der Compiler keinen Fehler, z.B. wenn schliessende Klammern vergessen wurden oder bei manchen Vergleichsausdruecken, so erkennt sie im allgemeinen der nachfolgende Assemblerlauf.
Nachfolgend ein paar Beispiele:

```
if a
;1      if a
```

```
ERROR WRONG EXPRESSION
AT LINE 2
ERROR WRONG EXPRESSION
```

Die Fehlermeldung erfolgt hier doppelt, da ansonsten eine Meldung auf der Konsole erscheint und eine in die Datei geschrieben wird.

```
if
;4      if
ERROR WRONG EXPRESSION
AT LINE 9
ERROR WRONG EXPRESSION
```

```
if hl<=de
;5      if hl<=de
registerpairs not implemented
AT LINE 11
registerpairs not implemented
```

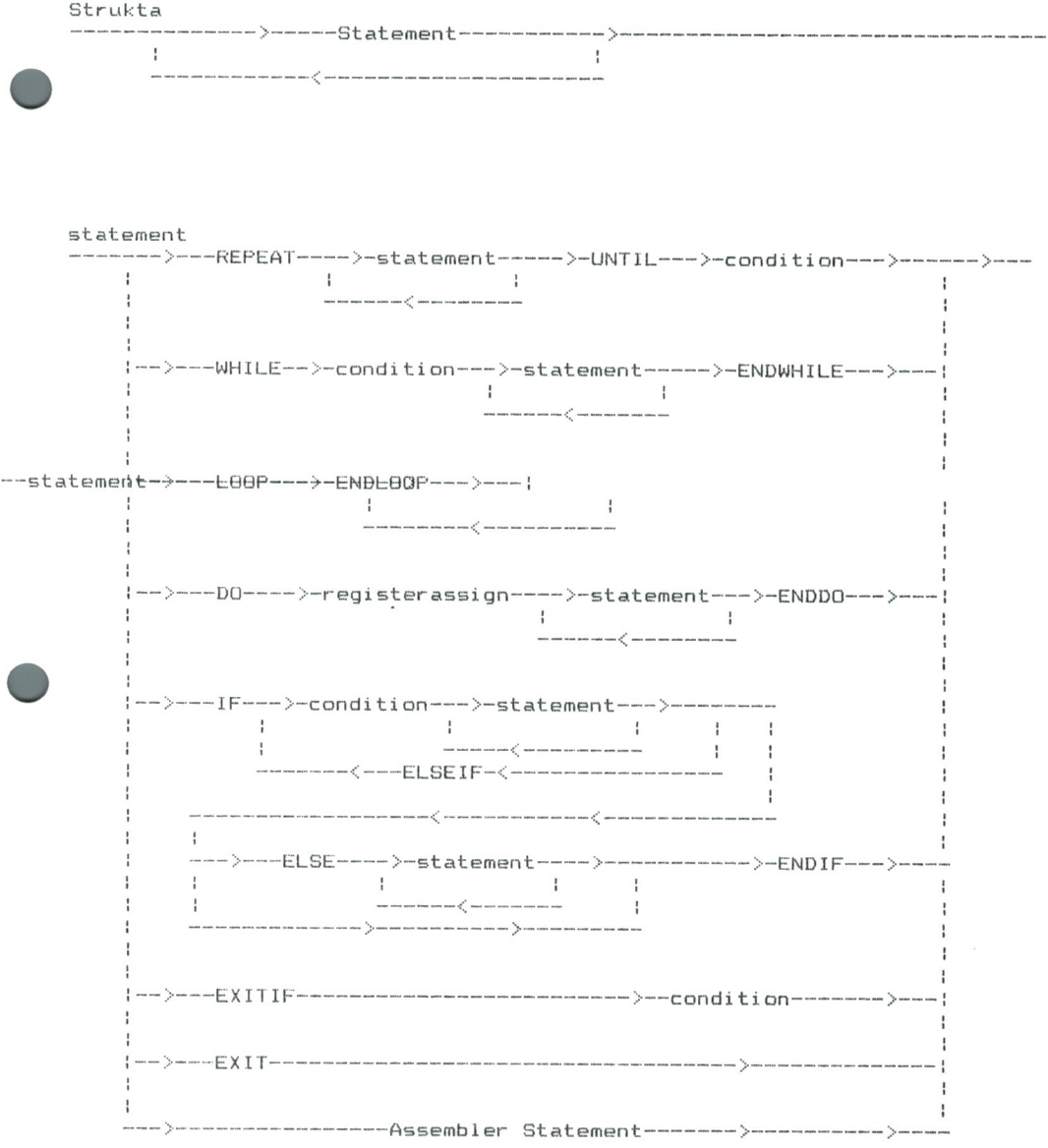
```
.pa

if hl=0
;6      if hl=0
        LD A,L
        OR H
        JP NZ,.L7

else
;6      else
        JP .L8
.L7:

endwhile
;7      endwhile
ERROR ENDWHILE WITHOUT WHILE
AT LINE 22
ERROR ENDWHILE WITHOUT WHILE
```

Werden schliessende Anweisungen wie ENDIF, UNTIL etc. vergessen und kann der Compiler sonst keinen Fehler entdecken, der darauf hinweist, so erkennt es erst der nachfolgende Assemblerlauf, da dann Marken vom Typ .L undefiniert sind. Anhand des Assemblerlistings laesst sich dann der Fehler schnell einkreisen.

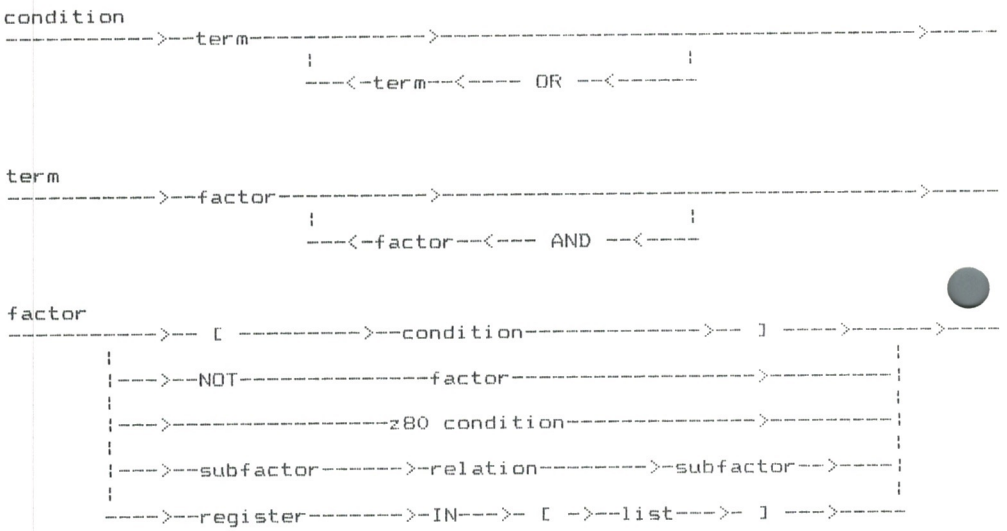
.pa
..@X
..@A
A. Syntaxdiagramme
.po 0

Syntaxdiagrams of STRUKTA by Rolf-Dieter Klein 830109

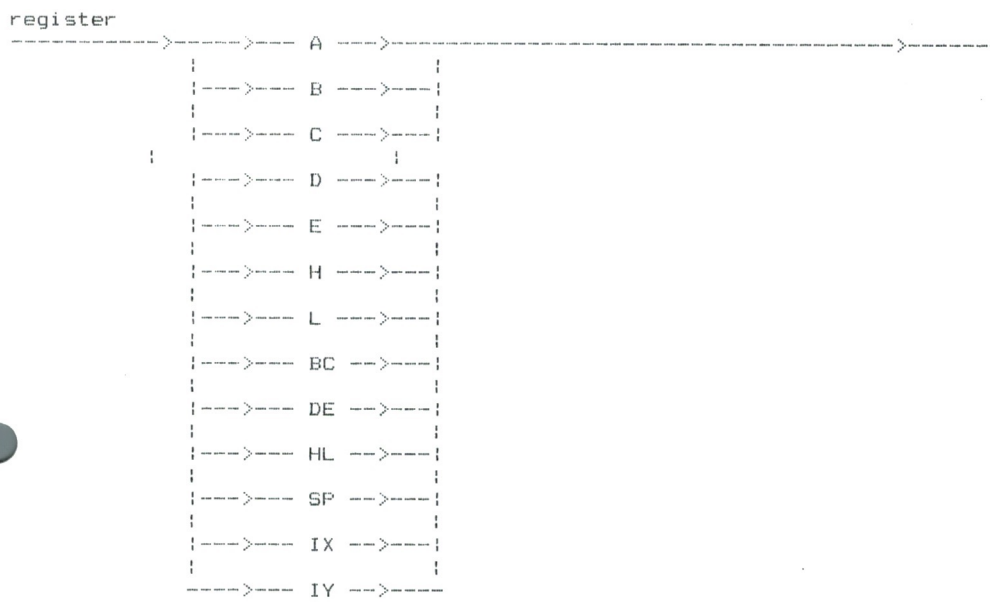
********** -1- **********

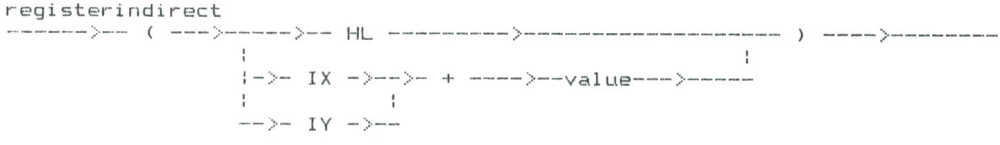


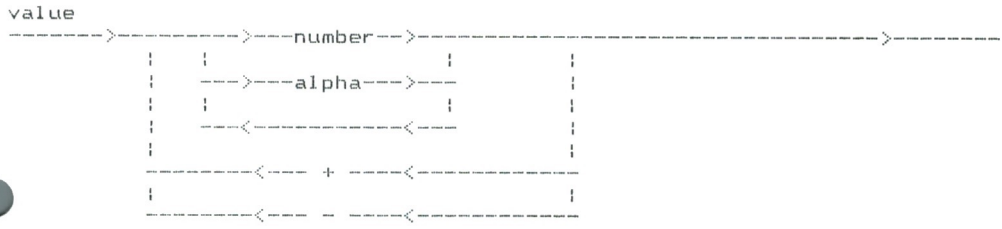
********** -2- **********


condition
term
OR
term
factor
AND
factor
[
condition
]
NOT
factor
z80 condition
subfactor
relation
subfactor
register
IN
[
list
]
list
subfactor
,
..
registerassign
register
,
value
subfactor
value
text
register
registerindirect

********** -3- **********


register
A
B
C
D
E
H
L
BC
DE
HL
SP
IX
IY

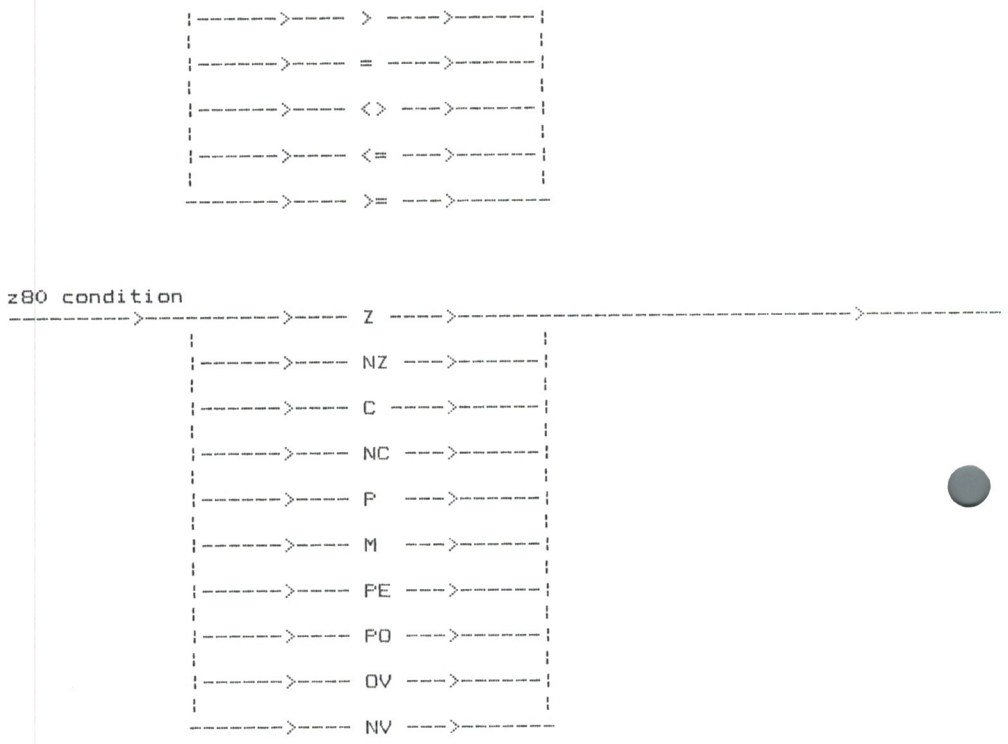

registerindirect
(
HL
IX
IY
+
value


value
number
alpha
+
-


text
'
ascii
"
ascii

********** -4- **********


relation
<



.pa

.po 8

..@A
B. Literaturverzeichnis

/1/ Wirth, Niklaus. Compilerbau. Teubner 1977

/2/ Alfred v. Aho u. Jeffrey D.Ullman.

Principles of Compiler Design. ADDISON-WESLEY 1979.

/3/ Gries, David. Compiler Construction for Digital

Computers. John Wiley & Sons, Inc. 1971

/4/ Kernighan ,Brian W. Plauger, P.J.

Software Tools. Addison-Wesley. 1976

.pa

..@A
C. Stichwortverzeichnis

.cp5
/

.cp5
A

.cp5
C

.cp5
D

.cp5
E

B>