Inhaltsverzeichnis

Wir danken den Firmen Apple Computer Marketing GmbH, München, und GVR, Düsseldorf, für ihre freundliche Unterstützung bei der Herstellung dieses Buchs.

Kapitel 1

Die ersten Schritte

Wenn Sie schon einmal mit einem Computer gearbeitet haben, werden Ihnen die Begriffe Maschinensprache oder Assemblersprache vermutlich bereits begegnet sein. Maschinensprache ist ganz einfach jene Sprache, die der Mikroprozessor Ihres Computers, ein Z80, spricht.

Wir wollen uns anhand einer Addition der Zahlen 1 und 83 mit diesem Sachverhalt einmal näher vertraut machen.

In der Ihnen vertrauten Umgangssprache würden Sie die Aufgabe folgendermaßen formulieren: Addiere zur Zahl Dreiundachtzig eine Eins. Wie lautet das Ergebnis?

In der Programmiersprache BASIC könnten Sie das Problem mit dem nachfolgend angegebenen Programm lösen:

```
10 LET A = 83
20 LET A = A + 1
30 PRINT A
```

In Maschinensprache sieht das dann beispielsweise so aus:

```
3E   53
C6   01
CD   5A   BB
C9
```

Im Vergleich zum vorangegangenen BASIC-Programm ist das ziemlich unverständlich, oder? Genau das ist der Grund, weshalb wir uns zur Entwicklung von Maschinenprogrammen eines Assemblers bedienen, in dessen symbolischer Schreibweise das Programm dann so aussieht:

Programm 1.1

```
LD A,83          Lade den Akkumulator A mit der Zahl 83
ADD A,1          Addiere zum Inhalt von A die Zahl 1
CALL 47962       Gib den Inhalt von A auf dem Bildschirm aus
RET              Kehre in das aufrufende Programm zurück
```

In dieser Form ist das Programm offensichtlich leichter zu lesen und zu verstehen. Mit einem Assembler können Sie ein Maschinenprogramm in der Assemblersymbolik eingeben und leichter kontrollieren. Ein Assembler tut nichts anderes, als ein in Assemblersprache geschriebenes Programm in Maschinenbefehle umzusetzen. Bei der Übersetzung wird dann beispielsweise die Anweisung LD A in den Maschinencode 3E umgewandelt und das entsprechende binäre Datenwort im Speicher am richtigen Platz abgelegt.

Der Mikroprozessor Z80 besitzt eine Reihe von Registern, die nichts anderes sind als Speicherzellen, in denen Daten abgelegt werden können. Die meisten der für den Z80 gültigen Befehle machen von diesen Registern Gebrauch. Eines der am häufigsten verwendeten Register heißt Akkumulator. Der Akkumulator kann Zahlenwerte im Bereich zwischen 0 und 255 speichern.

Schauen wir uns einmal einen Befehl an, mit dessen Hilfe wir einen Wert in den Akkumulator übertragen können. Er lautet:

LD A,n Lade den Akkumulator unmittelbar mit dem Wert n.

Bitte beachten Sie, daß die für den Befehl verwendeten Buchstaben einen direkten Bezug zu dem haben, was der englischsprachige Begriff aussagt: LD = LoaD = lade. Befehle dieser symbolischen Form werden Mnemonics genannt, weil sie leicht im Gedächtnis zu behalten sind.

Es gibt eine Fülle von Möglichkeiten, Zahlen in den Akkumulator (oder in eine andere Speicherstelle des Computers) zu laden oder daraus abzurufen. Die unterschiedlichen Möglichkeiten werden als **Adresslerungsarten** bezeichnet. Die zuvor gezeigte Art der direkten Speicherung einer konstanten Zahl heißt unmittelbare Adressierung.

Der nächstfolgende Befehl im Programm lautet ADD:

ADD A,n Addiere den Wert n zum Inhalt des Akkumulators.

Bei diesem Befehl wird die unmittelbar hinter dem Komma angegebene Zahl (in unserem Beispiel eine 1) zum aktuellen Inhalt des Akkumulators (A) hinzuaddiert. Das Ergebnis dieser Operation wird wieder im Akkumulator abgelegt. Die ursprünglich vorhandene Zahl 83 wird bei diesem Vorgang gelöscht.

Der nächste Befehl in unserem Beispielprogramm sieht recht verwirrend aus:

CALL 47962

Dieser Befehl sagt dem Computer, daß er den Inhalt des Akkumulators, d.h. die Zahl 84, auf dem Bildschirm ausgeben soll. Obgleich diese Erläuterung formal durchaus richtig ist, wird sie dem wahren Sachverhalt noch nicht ganz gerecht.

Sie müssen vielmehr wissen, daß der Schneider CPC 464 eine Vielzahl von Maschinenprogrammen (manchmal auch Routinen genannt) besitzt, die unlöschbar im Festwertspeicher (ROM) abgelegt sind. Ohne diese Routinen würde es beispielsweise nicht möglich sein, mit dem BASIC-Interpreter zu arbeiten. Eine dieser Routinen hat die Aufgabe, im Akkumulator stehende Werte auf dem Bildschirm des Monitors darzustellen.

Der Aufruf einer derartigen Routine erfolgt mittels des Befehls CALL. Jede der Routinen beginnt im Speicher an einer ganz bestimmten Stelle, die durch eine Adresse im Bereich zwischen 0 und 65535 gekennzeichnet ist. Die zuvor im CALL-Befehl vereinbarte Zahl 47962 verweist auf diejenige Speicherzelle, in der der erste Befehl der Maschinenroutine für die Ausgabe eines Zeichens auf dem Bildschirm abgespeichert ist.

So weit, so gut! Jetzt wollen wir uns damit beschäftigen, wie wir ein Maschinenprogramm starten und ablaufen lassen können.

Zu diesem Zweck zunächst einige Bemerkungen zum Assembler selbst: Der diesem Lernkurs beigefügte Assembler erlaubt es Ihnen, Ihre Maschinenprogramme auf einfache Art und Weise von der BASIC-Ebene aus zu editieren, wie Sie dies ja bereits gewöhnt sind, wenn Sie BASIC-Programme entwickeln. Wir benutzen dazu dieselben formalen Vereinbarungen, d.h. wir kennzeichnen jede einzelne Zeile durch eine Zeilennummer.

Bevor Sie mit dem Entwurf eines Programms beginnen, müssen Sie dem Assembler mitteilen, **wo** Sie das Programm im Speicher ablegen wollen. An dieser Stelle wollen wir die Entscheidung darüber noch dem Assembler selbst überlassen. Sie können dies dadurch erreichen, daß Sie als erstes die Anweisung ENT eingeben:

```
10  ENT
    .
    .
    .
Programm
    .
    .
```

Vom Assembler übersetzte (assemblierte) Programme können entweder von der BASIC-Ebene aus (durch einen CALL-Befehl) oder direkt unter Assemblerkontrolle mit dem Kommando C aufgerufen und gestartet werden.

In jedem Fall müssen Sie dem Computer mitteilen, daß Sie nach Ausführung des Maschinenprogramms in das jeweils aufrufende Programm zurückkehren müssen. Hierzu benutzen Sie im Programm den Befehl RET:

RET RETurn = Kehre aus dem Unterprogramm zurück.

Für die Umsetzung unseres aktuellen Programms müssen wir also

1. dem Assembler mitteilen, an welcher Stelle im Speicher wir unser Programm ablegen wollen,

2. den Wert 83 in den Akkumulator laden (LD A,83),

3. die Zahl 1 zum Inhalt des Akkumulators hinzuaddieren (ADD A,1),

4. die im Festwertspeicher (ROM) enthaltene Maschinenroutine zur Ausgabe des errechneten und im Akkumulator abgespeicherten Ergebniswertes auf dem Bildschirm aufrufen (CALL 47962),

5. aus dem Maschinenprogramm in den BASIC-Editor zurückspringen (RET) und

6. dem Assembler (nicht dem Z80) mitteilen, daß die Eingabe des Programmtextes beendet ist. Sie erreichen dies durch Betätigen der mit einem "Klammeraffen" (@) bezeichneten Taste.

Das sieht dann so aus (bitte noch nicht eingeben!):

Programm 1.1

```
10 ENT
20 LD A,83
30 ADD A,1
40 CALL 47962
50 RET
```

Bevor Sie diese Programmsequenz eingeben, sollten Sie den Assembler von der Kassette entsprechend den nachfolgenden Anweisungen in den Computer laden und dann weiter den Anweisungen folgen:

1. Lösen Sie einen Kaltstart des Computers dadurch aus, daß Sie gleichzeitig die Tasten SHIFT, CTRL und ESC betätigen und diese gemeinsam wieder loslassen. Auf dem Bildschirm wird der Grußtext ausgegeben, wie Sie dies beim Einschalten des Systems gewohnt sind.

2. Spulen Sie die dem Kurs beiliegende Kassette (Seite A) auf den Anfang zurück.

3. Betätigen Sie gleichzeitig die mit CTRL und ENTER (kleine ENTER-Taste rechts unten am Zahlenfeld) bezeichneten Tasten, und lassen Sie diese anschließend wieder los. Auf dem Bildschirm sollte nun der Text

RUN"
Press PLAY then any key:

ausgegeben werden. Ist dies nicht der Fall, dann wiederholen Sie bitte die zuvor erläuterte Prozedur.

4. Betätigen Sie die PLAY-Taste am Recorder und anschließend die Leertaste.

   Nach einiger Zeit wird nun eine Nachricht auf dem Bildschirm ausgegeben, die Sie darüber informiert, daß der Assembler geladen ist.

5. Betätigen Sie nun die mit CAPS LOCK bezeichnete Taste. Dadurch ist sichergestellt, daß nur Großbuchstaben eingegeben werden können. Drücken Sie dann die Taste I. Wie Sie dem Menütext entnehmen können, wird der Assembler durch dieses Kurzkommando für die Eingabe eines neuen Programms vorbereitet. Auf die Nachricht hin "Bitte Startzeile und Schrittweite eingeben" tippen Sie die Zahl 10 ein und schließen Ihre Eingabe, wie üblich, durch die ENTER-Taste ab.

   Auf dem Bildschirm wird nun eine 10 ausgegeben.

6. Geben Sie anschließend ENT ein, und bestätigen Sie Ihre Eingabe wieder durch ENTER.

   Hinweis: Denken Sie daran, daß Sie alle Ihre Eingaben grundsätzlich durch Betätigen der Eingabetaste ENTER abschließen müssen. Wir wollen nachfolgend nicht immer wieder darauf hinweisen.

   Unmittelbar nach dem unter C. angegebenen Schritt wird die Zahl 20 auf dem Bildschirm ausgegeben.

7. Geben Sie nun die nächstfolgende Zeile des Programms mit der Anweisung LD A,83 ein. Lassen Sie dabei das Leerzeichen zwischen LD und dem A **nicht** weg! Fügen Sie außerdem keine Leerzeichen zwischen dem Komma und der Zahl 83 oder sonstwo ein!

8. Jetzt wird die nächstfolgende Zeilennummer 30 ausgegeben, bei der Sie den Befehl ADD A,1 einzugeben haben. Achten Sie bitte auch hier wieder darauf, daß zwischen dem Mnemonic ADD und dem Operanden ein Leerzeichen stehen muß.

9. In Zeile 40 geben Sie CALL 47962 ein (Sie denken an das Leerzeichen?).

10. Schließlich tippen Sie in der Zeile 50 noch den Rücksprungbefehl RET ein.

11. Da Sie die Eingabe des Programms nun beendet haben, können Sie den durch I zu Beginn aktivierten Einfügemodus wieder verlassen und durch Betätigen der Taste @ (und ENTER!) in den Befehlsmodus zurückkehren.

12. Drücken Sie jetzt die Taste F mit der nachfolgenden Zahl 10 als Startzeile, um eine Ausgabe des Programmtextes (Listing) auf dem Bildschirm zu erhalten. (Sie können die Startzeile auch weglassen, wenn Sie den Programmtext wie im vorliegenden Fall von Beginn an ausgeben wollen).

    Das Programm sollte nun so aussehen

    Programm 1.1

```
10  ENT
20  LD   A,83
30  ADD  A,1
40  CALL 47962
50  RET
```

    Sollte dies wider Erwarten nicht der Fall sein, dann rufen Sie über die Taste R den Korrekturmodus auf und ersetzen die Zeile(n), die fehlerhaft ist (sind). Falls nach den Änderungen das Programm so aussieht, wie zuvor angegeben, dann geben Sie das Kurzkommando A ein. Wählen Sie unter den anschließend angegebenen Wahlmöglichkeiten die Nr. 2 aus. Der Assembler wird nun das in Assemblersprache formulierte Programm in den Maschinencode übersetzen.

13. Wenn der Assembler den Übersetzungsvorgang beendet hat, werden Sie aufgefordert, eine Taste zu drücken. Wenn Sie dies tun, gelangen Sie wieder in den Befehlsmodus, und das Menü erscheint erneut auf dem Bildschirm. Enthält das Programm Fehler, dann gibt der Assembler eine Fehlermeldung aus. Benutzen Sie dann entweder R oder I, um in den Editiermodus zur Korrektur zurückzukehren.

14. Um das übersetzte Maschinenprogramm ablaufen zu lassen, geben Sie den Kurzbefehl C ein.

Falls Sie sich nun darüber wundern, daß anstatt des erwarteten Rechenergebnisses 84 der Großbuchstabe T auf dem Bildschirm ausgegeben wird, dann sollten Sie einen Blick in den Anhang III des Handbuchs zum Schneider CPC 464 werfen. Sie finden dort eine Tabelle mit dem ASCII-Zeichensatz. Wie Sie sehen, entspricht dem ASCII-Code 84 in dezimaler Schreibweise der Buchstabe groß T.

ASCII ist eine Abkürzung für "American Standard Code for Information Interchange". Jedes auf dem Bildschirm darstellbare Zeichen besitzt seinen eigenen ASCII-Code. Wenn die Zahl 84 aus dem Akkumulator ausgelesen und mittels der unter der Adresse 47962 beginnenden Routine auf dem Bildschirm ausgegeben wird, interpretiert die Routine den Inhalt des Registers A als ASCII-Code. Dies führt im vorliegenden Fall zur Darstellung des ASCII-Zeichens T.

Auch unter Kontrolle des BASIC-Interpreters ist dies nicht anders, wenn Sie mittels PRINT 83+1 das Ergebnis dieser Summenbildung ausgeben. In diesem Fall muß der Computer, bevor er die Zeichen 8 und 4 für 84 auf dem Bildschirm darstellt, zunächst entscheiden, was mit den Zeichen 83+1 gemeint ist, dann muß er die Summe berechnen und anschließend das numerische Ergebnis dieser Operation unter Verwendung der ASCII-Codes für die Zahlen 8 und 4 ausgeben. Dies ist in der Tat recht kompliziert. Der Computer hat daher eine eingebaute Routine zur Verfügung, die diese Aufgabe automatisch für den BASIC-Interpreter erledigt.

Die Architektur des Mikroprozessors Z80

Neben dem Akkumulator besitzt der Z80 noch andere Register, wie das vereinfachte Schema in Abb. 1.1 zeigt.

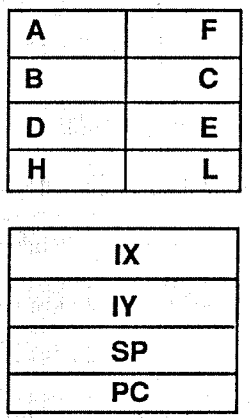


Abb. 1.1: Vereinfachtes Schema der Registerstruktur des Z80

Den Akkumulator, auch A-Register genannt, haben Sie schon kennengelernt. Dieses Register nimmt eine Sonderstellung ein, weil es weitaus mehr Befehle im Z80-Befehlssatz gibt, die mit dem Akkumulator zusammenarbeiten, als mit anderen Registern.

Die Register A bis L (es handelt sich einfach um Speicherzellen hoher Arbeitsgeschwindigkeit auf dem Z80-Chip) gehören zu den 8-bit-Registern, d.h. sie können nur Zahlen im Bereich von 0 bis 255 speichern. Das scheint auf den ersten Blick hin nicht gerade brauchbar zu sein. Es gibt jedoch Wege, die Register so zu kombinieren, daß weitaus größere Zahlenwerte verarbeitet werden können. Wir werden später noch darauf zurückkommen.

Die beiden Register H und L zusammen werden üblicherweise als primäre Datenzeiger bezeichnet. Die Register B und C sowie D und E in Kombination werden dagegen sekundäre Datenzeiger genannt. Der Hauptgrund für diese formale Unterscheidung ist der, daß jene Befehle, die von der Registerkombination H und L Gebrauch machen, in den weitaus meisten Fällen kürzer ausfallen und nach der Übersetzung in Maschinensprache deshalb schneller ausgeführt werden.

Die unmittelbare Adressierung von Registern

Die unmittelbare Adressierung des Akkumulators wurde bereits zuvor im Zusammenhang mit dem Befehl

```
LD A,83
```

erörtert. Mit seiner Hilfe wurde eine Zahl in den Akkumulator geladen. Genau die gleiche Art von Befehlen kann benutzt werden, um Zahlen im Wertebereich zwischen 0 und 255 in jedes andere Register zu übertragen. So wird beispielsweise mit dem Befehl

```
LD B,83
```

die Zahl 83 in das B-Register geladen.

Die allgemeine Form dieses Befehlstyps lautet

LD r,n Lade das Register r unmittelbar mit dem Wert n.

Der Parameter r steht für A, B, C, D, E, H oder L.

Register-Register-Adressierung

Es ist nicht nur möglich, jedes Register mit einer Zahl zu laden. Es kann auch der Inhalt jedes beliebigen 8-bit-Registers in ein beliebiges anderes Register kopiert werden. Das nachfolgend angegebene Programm 1.2 beispielsweise lädt die Zahl 43 in das Register L, kopiert dessen Inhalt dann in den Akkumulator und stellt das entsprechende ASCII-Zeichen (+) auf dem Bildschirm dar.

Programm 1.2

```
10  ENT
20  LD  L,43        Lade L mit 43
30  LD  A,L         Lade A mit dem Inhalt von L
40  CALL 47962      Ausgabe auf dem Bildschirm
50  RET             Rückkehr ins aufrufende Programm
```

Die Zeile 30 enthält den neuen Befehl

```
LD A,L
```

Mit seiner Hilfe wird der Inhalt von L in den Akkumulator A übertragen.

Ähnliche Beispiele für die gerade vorgestellte Adressierungsart lauten:

```
LD B,E
```

oder

```
LD C,A
```

Allgemein lautet der Befehl

LD r1,r2 Lade den Inhalt des Registers r2 in das Register r1.

Man bezeichnet diese Register-Register-Adressierung als implizite Adressierung.

So weit, so gut! Geben Sie nun das Programm 1.2 ein. Falls Sie bei der Eingabe einen Fehler machen, bevor Sie die ENTER-Taste betätigt haben, benutzen Sie bitte einfach die mit DEL (DELete) bezeichnete Taste zur Korrektur. Wenn Sie den Fehler erst nach der endgültigen Eingabe und Bestätigung entdecken, können Sie durch Drücken der mit dem Zeichen @ versehenen Taste den Einfügemodus verlassen und mit dem Kommando R die falsche Zeile vor der Übersetzung gegen die korrekt geschriebene austauschen.

1. Laden und starten Sie das Assemblerprogramm, falls Sie dies nicht bereits erledigt haben. Im letzteren Fall springen Sie durch Betätigen einer beliebigen Taste in das Eingangsmenü.

2. Wählen Sie zur Programmeingabe das Kommando I aus, und drücken Sie die CAPS LOCK-Taste, damit Sie Ihre Eingaben im Großschreibungsmodus durchführen.

3. Sagen Sie nun dem Assembler, wo das Maschinenprogramm im Speicher liegen soll. Im vorliegenden Fall geben Sie einfach die Zeichenfolge ENT ein und bestätigen Ihre Eingabe durch die ENTER-Taste.

4. Geben Sie nun LD L,43 ein, und bestätigen Sie auch diese Eingabe durch die ENTER-Taste. (Denken Sie bitte nunmehr immer daran, daß Sie nach jeder Programmzeile die ENTER-Taste drücken.)

5. Geben Sie dann LD A,L ein.

6. Nun folgt die Zeile CALL 47962.

7. Und nun noch RET.

8. Verlassen Sie den Textmodus durch Betätigen der mit @ bezeichneten Taste und ENTER.

9. Nun lassen Sie sich das Programmlisting auf dem Bildschirm ausgeben, indem Sie das Kommando F eingeben. Schauen Sie es sich das Programm nochmals gut an, und vergleichen Sie es mit der Vorlage.

10. Falls es fehlerfrei ist, geben Sie das Kommando A ein, und starten Sie so den Übersetzungsvorgang.

11. Nach dessen Beendigung starten Sie das Maschinenprogramm durch das Kommando C.

Wenn Sie sich an das zuvor erläuterte Schema halten, können Sie bereits an dieser Stelle eigene Programme schreiben und ablaufen lassen. Überzeugen Sie sich über Ihren aktuellen Kenntnisstand dadurch, daß Sie die nachfolgend gestellte Übungsaufgabe bearbeiten. Vergessen Sie auf keinen Fall, am Ende eines Programms den Befehl RET für den Rücksprung zu vereinbaren, weil andernfalls der Computer nach der Programmausführung auf der Suche nach weiteren ausführbaren Befehlen eine Wanderung durch seinen Speicher unternimmt. In den weitaus meisten Fällen wird er irgend etwas finden, was ihn dazu veranlaßt abzustürzen. Mit anderen Worten: Es wird keine andere Möglichkeit geben, als das System aus- und wieder einzuschalten. In so einem Fall müssen Sie dann leider von vorne beginnen und das Assemblerprogramm neu laden.

Übungsaufgabe 1.1

Laden Sie den Akkumulator unmittelbar mit der Zahl 65, und stellen Sie den Inhalt auf dem Bildschirm dar. Die Zahl 65 ist der ASCII-Code für den Großbuchstaben A. Eine mögliche Lösung zu dieser Aufgabe finden Sie im Anhang G.

Mehr über den Befehl CALL 47962

Bei mehrfacher Anwendung dieses Unterprogrammaufrufs können fortlaufende Ausgaben auf dem Bildschirm erzielt werden. Schauen Sie sich dazu das folgende Programm an:

Programm 1.3

```
10  ENT
20  LD A,72        Lade den Akkumulator mit dem ASCII-Code für H
30  CALL 47962     Gib das Zeichen H auf dem Bildschirm aus
40  LD A,65        Lade den Akkumulator mit dem ASCII-Code für A
50  CALL 47962     Gib das Zeichen A auf dem Bildschirm aus
60  LD A,76        Lade den Akkumulator mit dem ASCII-Code für L
70  CALL 47962     Gib das Zeichen L aus
80  CALL 47962     Gib das Zeichen L aus
90  LD A,79        Lade den Akkumulator mit dem ASCII-Code für
                   das Zeichen O
100 CALL 47962     Gib das Zeichen O aus
110 RET            Kehre zu BASIC zurück
```

Dieses Programm gibt für den Fall, daß Sie es richtig eingegeben und übersetzt haben, den Text "HALLO" auf dem Bildschirm Ihres Schneider CPC 464 aus.

Übungsaufgabe 1.2

Schreiben Sie Ihren Namen in die obere linke Ecke des Bildschirms.

Eine Lösung für den Namen "FRED" finden Sie im Anhang G.

Das Ende dieses ersten Kapitels ist erreicht. So schlimm war es doch bis jetzt nicht, oder? Wenn Sie alles sorgfältig gelesen und die Übungsbeispiele durchgeführt haben, sollten Ihnen die nachfolgend angegebenen Befehle und Begriffe vertraut sein:

```
LD  A,n
LD  r1,r2
RET
ENT
@
CALL 47962
```

ADD A,n
Unmittelbare Adressierung
Register-Register-Adressierung

Was bedeutet wohl der folgende Befehl (r ist irgendein Register)?

ADD A,r

Kapitel 2

Sprünge, Unterprogramme und Labels

Nur sehr wenige Programme laufen in der Praxis ohne irgendwelche Sprünge oder Verzweigungen ab. In diesem Kapitel wollen wir uns deshalb mit Sprungbefehlen und deren programmtechnischer Anwendung vertraut machen. Danach schauen wir uns die Statusbedingungen an, die die Sprünge kontrollieren. Im Anschluß daran kümmern wir uns in diesem Kapitel um bedingte und unbedingte Aufrufe von Unterprogrammen. In diesem Zusammenhang werden symbolische Marken eingeführt, die in der Computerfachsprache "Labels" genannt werden. Sie erweisen sich als äußerst nützlich und tragen erheblich zum Programmierkomfort bei.

Unbedingte Sprünge

Unbedingte Sprungbefehle weisen das Programm an, ohne jede Vorbedingung an eine vereinbarte Stelle zu springen. Der Befehlssatz des Z80 enthält fünf Sprungbefehle dieser Art. Zu diesem Zeitpunkt reicht es, wenn wir uns nur zwei von ihnen näher ansehen. Die anderen drei werden später vorgestellt.

Der erste dieser beiden Befehle lautet:

JP nn Springe (JumP) zu einer vereinbarten Adresse.

Der Befehl JP 200 bedeutet beispielsweise, daß ein Sprung zur Speicherstelle mit der Adresse 200 ausgeführt werden soll.

Im Zusammenhang mit einem kleinen Programm sieht das dann wie in Abb. 2.1 gezeigt aus.

Ein Sprung der zuvor gezeigten Art bewirkt natürlich nicht viel. Man kann aber auf die gezeigte Art und Weise zusätzliche Befehle in ein bereits bestehendes Programm einfügen, ohne die bisherige Befehlsfolge ändern zu müssen. In dem Programmablaufschema aus Abb. 2.1 wurden so die Befehle ADD A,1 sowie CALL 47962 und JP 30005 in das Programm eingefügt.

Geben Sie dieses Programm nun in den Computer ein. Bitte erinnern Sie sich daran, daß Sie jede Programmzeile mit ENTER abschließen müssen und daß Sie für die Programmeingabe das Kommando I verwenden müssen. Es steht für Insert, also Einfügen. Die Startzeile sollten Sie wie bisher auch zu 10 wählen.

Beachten Sie, daß die erste Zeile dieses Programms nicht mit dem Befehl ENT beginnt. Weil das Programm ganz definierte Speicheradressen anspringt, muß der exakte Beginn des Programms bekannt sein. Der Pseudobefehl ORG 30000 weist den Assembler an, das Maschinenprogramm ab der Adresse 30000 im Speicher abzulegen. Schlagen Sie im Anhang F nach, wenn Sie weitere Informationen hierzu wünschen.

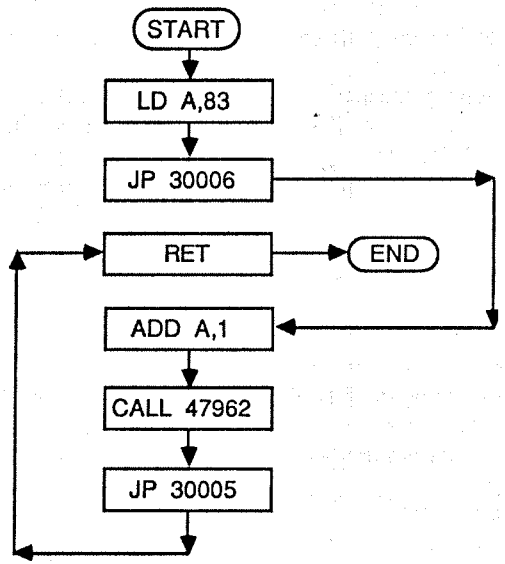


Abb. 2.1: Beispiel für die Verwendung des Befehls JP innerhalb eines Programms

Das Programm lautet:

Programm 2.1

```
ORG 30000
LD A,83
JP 30006
RET
ADD A,1
CALL 47962
JP 30005
```

Übersetzen Sie das Programm durch Aufruf des Kommandos A, und wählen Sie die Option 2.

Auf dem Bildschirm sollte nun etwa das in Abb. 2.2 Gezeigte zu sehen sein.

Adresse	Zeilen-nummer	Objektcode	Quellcode
7530	10		ORG 30000
7530	20	3E 53	LD A,83
7532	30	C3 36 75	JP 30006
7535	40	C9	RET
7536	50	C6 01	ADD A,1
7538	60	CD 5A BB	CALL 47962
753B	70	C3 35 75	JP 30005

Abb. 2.2: Assemblieren des Programms 2.1

Die erste Zahl in der Adressenspalte scheint nicht mit der Adresse 30000 übereinzustimmen, bei der das Programm starten soll. Das liegt daran, daß die Adressen in der Liste in hexadezimaler Schreibweise ausgegeben werden. Kümmern Sie sich im Moment nicht darum. Merken Sie sich nur, daß die dezimale Zahl 30000 in hexadezimaler Schreibweise 7530, die Zahl 30006 der Hexadezimalzahl 7536 und die Dezimalzahl 30005 der hexadezimalen 7535 entspricht.

Bei einem Sprung muß dem Programm mitgeteilt werden, zu welcher Stelle es springen soll. Das heißt, wir müssen eine Sprungadresse angeben. Im vorliegenden Fall ist dies die Adresse 30005. Die Berechnung dieser Adresse ist unkompliziert, wie nachfolgend erläutert wird.

Die Zahlen und Buchstaben in der zweiten Spalte von rechts stellen den Objektcode dar. Er ist das Ergebnis der Übersetzung, Maschinencode genannt.

Jedes Paar alphanumerischer Zeichen (das sind Ziffern und/oder Buchstaben) in dieser Spalte stellt ein Element aus der Assemblersprache dar. So bedeutet 3E beispielsweise LD A, und 53 entspricht der Zahl 83 (kümmern Sie sich nicht darum, warum das so ist. Wir haben es mit der hexadezimalen Schreibweise zu tun, auf die in einem anderen Kapitel eingegangen wird).

Für Zahlen, die den Wert 255 überschreiten, werden vier alphanumerische Zeichen benötigt. So entspricht C3 dem Befehl JP und die Zahl 3075 der dezimalen Adresse 30000. In welcher Weise hilft uns diese Information nun bei der Berechnung der Sprungadressen weiter? Nun, jedes Ziffernpaar benötigt genau eine Speicherstelle, d.h. ein Byte. Da wir wissen, daß der erste Befehl unter der Adresse 30000 abgelegt wird (wegen des Pseudobefehls ORG 30000), brauchen wir einfach nur die Anzahl der Ziffernpaare abzuzählen.

Wenn wir beispielsweise zu dem Befehl ADD A,1 springen wollen, dann müssen wir bis zu jener Adresse springen, unter der der Maschinencode C6 abgespeichert ist. Beginnen wir also bei der Adresse 30000, unter der das Codewort 3E abgelegt ist, dann gelangen wir über 53, C3, 36, 75 und C9 nach C6. Diese Stelle ist 6 Bytes von der Adresse 30000 entfernt. Die Adresse lautet also 30006. Natürlich war zu Beginn Ihres Zählvorgangs die 3675 im JP-Befehl gar nicht vorhanden. Da Sie aber wissen, daß eine Adressenangabe zwei Bytes benötigt, ist das Problem beim Programmentwurf auf einem Blatt Papier leicht lösbar.

Im Anhang finden Sie Tabellen, die Sie darüber informieren, wie viele Bytes ein Maschinenbefehl erfordert. Sie erweisen sich bei der Berechnung von Sprungadressen als eine große Hilfe.

Nachfolgend eine genaue Analyse des Programms:

ORG — Dieser Befehl benötigt keinen Speicherplatz. Er wird ausschließlich vom Assembler benutzt und als Pseudobefehl bezeichnet.

LD A,83 — Diese Anweisung benötigt zwei Bytes im Speicher. Das erste Byte nimmt den Befehlscode auf, das zweite dient zur Abspeicherung des Datenwertes.

JP 30006 — Dieser Befehl erfordert drei aufeinanderfolgende Bytes. Eines dient zur Abspeicherung des Befehlscodes C3, die anderen beiden werden zur Aufnahme der Sprungadresse benötigt.

RET — Hierfür wird nur ein Byte benötigt, um den Befehlscode C9 abzuspeichern.

ADD A,1 — Zwei Bytes sind erforderlich. Das erste enthält den Befehlscode C6, das zweite den Datenwert.

CALL 47962 — Dieser Befehl macht drei Bytes erforderlich. CD ist der Befehlscode, und 5A sowie BB entsprechen der aufgerufenen Adresse.

JP 30005 — Dieser Befehl entspricht dem bereits erläuterten Befehl JP 30006. Nur die Ansprungadresse lautet anders.

Den Speicherbedarf für unser Programm können wir einfach durch Abzählen der Bytes ermitteln: Das Programm 2.1 ist offensichtlich 14 Bytes lang.

Mit dem JP-Befehl können wir jede Adresse im Bereich zwischen 0 und 65535 anspringen. Liegt die Ansprungadresse in einem Bereich von -126 bis +129 Bytes von der aktuellen Adresse entfernt, dann können wir von einem relativen Sprung Gebrauch machen. Der Vorteil eines derartigen Sprungbefehls liegt

darin, daß er im Gegensatz zum bisher verwendeten nur zwei Bytes benötigt. Er wird daher schneller abgearbeitet.

JR e — Führe einen relativen Sprung (Jump Relative) zu der durch e vereinbarten Adresse aus.

Nehmen wir einmal an, das Programm befindet sich gerade bei der Adresse 30000, und es muß einen Sprung zur Adresse 30045 durchführen. Sie können dann einfach schreiben:

```
JR 30045
```

Der Parameter e ist ein Adreßoffset. Im vorliegenden Fall müßte er 39 lauten. Sie brauchen ihn allerdings nicht selbst zu berechnen. Dies tut der Assembler für Sie automatisch, wenn Sie einfach die Zieladresse angeben. Merken Sie sich bitte, daß Sie den Befehl JR nur dann verwenden können, wenn Sie eine Adresse anspringen, die im Abstandsbereich von -126 bis +129 Bytes von jener aktuellen Adresse liegt, unter der der JR-Befehl abgespeichert ist.

Der Programmzähler

Bevor wir in unserem Kurs den nächsten größeren Schritt machen, müssen wir uns ein wenig mit der Frage beschäftigen, wie eine Sprunginformation intern verarbeitet wird. Unser letztes Programm beginnt, wie Sie bei einem kontrollierenden Blick auf die Abb. 2.2 nochmals feststellen können, bei der Adresse 30000. Während des Programmlaufs erhält es die Anweisungen, zu den Adressen 30005 bzw. 30006 zu springen. Wie macht der Computer das eigentlich?

Der Mikroprozessor Z80 enthält auf seinem Chip ein Register mit 16 bit Wortlänge, das Programmzähler oder auch kurz PC (Program Counter) genannt wird. Dieses Register enthält immer die Adresse des aktuell auszuführenden Befehls. Beim Start eines Maschinenprogramms wird der Programmzähler mit dessen Anfangsadresse geladen. Nach Ausführung des ersten Befehls wird der Inhalt des Zählers so verändert, daß er auf den nächsten Befehl verweist. Ein Sprungbefehl lädt demnach den PC mit der Zieladresse, d.h. er verweist auf den neuen Befehl. Als Folge davon setzt das Programm seinen Lauf bei der im Sprungbefehl vereinbarten neuen Adresse fort. Der Befehlssatz des Z80 enthält auch noch andere Instruktionen, die den Inhalt des Programmzählers verändern. Sie werden an anderer Stelle dieses Kurses vorgestellt.

Die Abb. 2.3 erläutert die Ausführung des Programms etwas eingehender.

Bisher haben wir uns nur mit sogenannten unbedingten Sprüngen beschäftigt. Sie sind zwar sehr nützlich, besser wäre es jedoch, wenn ein Sprung in Abhän-

gigkeit von einer irgendwie gearteten Bedingung ausgeführt werden könnte. Hierfür steht die Gruppe der bedingten Sprungbefehle zur Verfügung.

Programm		PC vor der Befehlsausführung	PC nach der Befehlsausführung
ORG	30000	?	30000
LD	A,83	30000	30002
JP	30006	30002	30006
ADD	A,1	30006	30008
CALL	47962	30008	30010
JP	30005	30010	30005
RET		30005	Rücksprung

Abb. 2.3: Die Werte des Programmzählers bei der Abarbeitung von Programm 2.1

Bedingte Sprünge

Jedes Programm, das in seinem Verlauf Bedingungen abfragt, benötigt für Programmverzweigungen bedingte Sprungbefehle. In BASIC würde diesen die IF...THEN-Anweisung entsprechen, d.h. beispielsweise

```
10 IF X=Y THEN GOTO 500
```

Diese Programmzeile führt dazu, daß der Computer die Variablen X und Y miteinander vergleicht. Für den Fall, daß sie denselben Wert haben, springt das Programm zur Zeile 500.

Der Z80 führt entsprechende Anweisungen unter Kontrolle eines speziellen Registers aus, das als Statusregister bezeichnet wird. Wie der Akkumulator und die Register B, C, D, E, H und L handelt es sich dabei um ein Register mit 8 bit Wortlänge. Es wird jedoch völlig anders genutzt. Während die zuvor erwähnten anderen Register nämlich dazu verwendet werden, Bytes zu speichern oder zu manipulieren, wird beim Statusregister jedes Bit einzeln interpretiert. Die Bits des Statusregisters werden als Statusbits oder auch als Flaggen bezeichnet. Der Z80 kann jedes der Statusbits einzeln setzen (1) oder auch zurücksetzen (0). Er kann außerdem jedes einzelne Statusbit auf seinen logischen Zustand hin abfragen.

Eines der Statusbits ist beispielsweise das Z-Bit (auch Nullbit genannt). Jedesmal, wenn eine arithmetische Operation als Ergebnis eine 0 liefert, wird dieses Bit gesetzt. Ist das Ergebnis verschieden von 0, wird das Z-Bit zurückgesetzt.

Eine Vielzahl von Befehlen beeinflußt dieses Statusbit. Hierzu gehört auch der Befehl DEC:

DEC d Erniedrige (DECrement) den Inhalt des in d vereinbarten Registers um 1.

Dieser Befehl erniedrigt den Inhalt des im Parameter d angegebenen Registers B, C, D, E, H, L oder A bei jeder Ausführung um den Wert 1. Ist der Inhalt des Registers nach der Befehlsausführung 0, wird das Z-Bit gesetzt, im anderen Fall zurückgesetzt.

Der Befehlsteil DEC wird üblicherweise als Operator bezeichnet, der Befehlsteil d dagegen als Operand. Der Operator wirkt also auf den nachfolgenden Operanden. Einige Operatoren, wie beispielsweise LD, machen zwei Operanden erforderlich. Im Falle von LD A,10 sind dies A und 10.

Wir wollen nun mit der Einführung in die Programmierung fortfahren. Das folgende Programm 2.2 gibt zehn aufeinanderfolgende Zeichen A auf dem Bildschirm aus.

Programm 2.2

```
ORG 30000
LD C,10
LD A,65
CALL 47962
DEC C
JR NZ,30004
RET
```

Geben Sie dieses Programm ein, und übersetzen Sie es mit dem Kommando A und der Option 2. Zum Programmstart verlassen Sie den Assembler mit dem Kommando X und geben dann CALL 30000 ein. Unmittelbar danach werden die zehn Großbuchstaben A auf dem Bildschirm ausgegeben. Zur Rückkehr in den Assembler betätigen Sie bitte die Taste mit dem Dezimalpunkt im numerischen Teil der Tastatur.

Für Sie enthält nur die Zeile mit der JR NZ-Anweisung etwas Neues. Dieser Befehl fragt den aktuellen Zustand des Z-Bits ab und leitet einen relativen Sprung zur Adresse 30004 ein, wenn das Ergebnis der zuletzt ausgeführten Operation nicht 0 ist. Das Programm bewirkt also nichts weiter, als daß der Inhalt des Akkumulators auf dem Bildschirm ausgegeben, das Register C um 1 erniedrigt und das Z-Bit anschließend auf seinen Zustand hin abgefragt wird. Wenn es nicht gesetzt ist, erfolgt ein Sprung zur Adresse 30004. Andernfalls geht das Programm zum nächstfolgenden Befehl weiter, d.h. es führt den Befehl RET aus und springt ins BASIC zurück.

Die allgemeine Form des JR-Befehl lautet:

JR NZ,e Springe relativ zur Adresse e, wenn das Ergebnis der letzten Operation verschieden von 0 ist.

Übungsaufgabe 2.1

Schreiben Sie das Programm 2.2 so um, daß anstelle des C-Registers das B-Register benutzt wird.

Übungsaufgabe 2.2

Warum wurde im Programm von der Operation JR NZ,e und nicht von JP NZ,e Gebrauch gemacht?

Antworten zu beiden Aufgaben finden Sie wiederum im Anhang G.

Es ist zu erwarten, daß es zum DEC-Befehl auch eine komplementäre Anweisung gibt, um einen Operanden im Wert zu erhöhen. Sie lautet:

INC d Erhöhe (INCrement) den Inhalt des angegebenen Operanden um 1.

Bisher haben wir gelernt, wie ein Programm unbedingt oder aber in Abhängigkeit davon, ob ein Ergebnis 0 war oder nicht, zu einer vorgegebenen Adresse springen kann. In beiden Fällen mußten wir die Zieladresse genau kennen. In kurzen Programmen ist das durchaus praktikabel. Bei längeren Programmen wird es jedoch zunehmend schwieriger und damit zeitraubender, die Sprungadressen zu berechnen. Aus dieser mißlichen Situation helfen uns sogenannte Labels heraus, die manchmal auch als Marken bezeichnet werden.

Labels

Bei der Verwendung von Labels ist es möglich, Befehlszeilen mit Namen anzusprechen. Eine Berechnung der zugehörigen Zieladressen erübrigt sich in diesem Falle. Derartige Labels werden häufig auch korrekter symbolische Labels genannt, weil ein Label eine symbolische Bezeichnung für eine Speicherstelle darstellt. Die Anweisung

```
JP  LOOP:
```

beispielsweise führt dazu, daß der Assembler das Label "LOOP" bei jedem Auftreten im Programm durch jene Adreßangabe ersetzt, die dem Label zuvor zugewiesen wurde.

Um dem Assembler mitzuteilen, daß eine Folge von alphanumerischen Zeichen als Label aufzufassen ist, muß an deren Ende ein Doppelpunkt gesetzt werden. Außerdem ist zu beachten, daß ein Label aus nicht mehr als sechs Zeichen bestehen darf.

Wenn beispielsweise innerhalb eines Programms jene Speicherstelle angesprungen werden soll, die den Befehl DEC C enthält, dann sieht das etwa so aus:

```
        .
        .
        .
        .
    JP  LOOP:
        .
        .
        .
        .
LOOP:   DEC C
        .
        .
        .
```

Es gibt noch mehr im Zusammenhang mit Labels zu beachten. So muß der Doppelpunkt immer unmittelbar hinter dem letzten Zeichen stehen (es darf also kein Leerzeichen eingefügt werden). Zwischen dem Doppelpunkt und dem nachfolgenden Befehlsteil muß dagegen ein Leerzeichen eingefügt werden. So kann der Assembler problemlos feststellen, wo das Label zu Ende ist. Machen Sie sich nichts daraus, wenn Sie nicht immer an diese Regeln denken. Der Assembler wird Sie schon rechtzeitig auf Fehler aufmerksam machen

Also, noch einmal:

1. Ein Label darf nur aus bis zu sechs Zeichen bestehen.
2. Unmittelbar nach dem letzten Zeichen des Labels muß ein Doppelpunkt folgen.
3. Zwischen dem Doppelpunkt und dem nachfolgenden Befehlsteil muß ein Leerzeichen eingefügt werden.
4. Ein Label darf kein Leerzeichen enthalten.

Beispielsweise sind die Labels

```
LOOP:
TEST:
NXTCHR:
```

alle korrekt. Die nachfolgenden dürfen jedoch nicht vereinbart werden:

```
LOOP :
BACKONE:
NEXT L:
```

Um uns mit der Vereinbarung von Labels vertraut zu machen, wollen wir nun unser Programm 2.1 ein wenig umschreiben. Da wir keine absoluten Sprünge verwenden, können wir das Programm wieder mit der bereits bekannten Anweisung ENT beginnen. Der ORG-Befehl ist hier entbehrlich.

Programm 2.3

```
ENT
LD A,83
JP NXT:
END: RET
NXT: ADD A,1
CALL 47962
JP END:
```

Wenn Sie sich nun das Listing dieses Programms ausgeben lassen, werden Sie feststellen, daß die Labels in einer eigenen Spalte ausgegeben werden. Die Lesbarkeit wird dadurch merkbar erhöht.

```
        ENT
        LD    A,83
        JP    NXT:
END:    RET
NXT:    ADD   A,1
        CALL  47962
        JP    END:
```

Damit alle Programmbeispiele in diesem Buch leicht lesbar sind, werden sie von jetzt an immer in dieser Form angegeben. Eingeben sollten Sie diese aber auf die bereits erprobte normale Weise.

Mit den bisher vorgestellten Befehlen können wir unbedingt oder aber in Abhängigkeit von einer Bedingung entweder an jeden Punkt des Speichers oder aber bei Benutzung eines relativen Sprungbefehls innerhalb eines Adreßbereichs von +129 bis -126 springen. Labels können außerdem zur Erleichterung der Berechnung von Sprungadressen verwendet werden.

Wir wollen alle bisher vorgestellten Befehle dazu verwenden, um die Zahlen 1 bis 9 auf dem Bildschirm auszugeben. In Kapitel 1 haben Sie gesehen, daß der ASCII-Code für den Buchstaben T der Zahl 84 entspricht. Nun, die Ziffern 0 bis 9

besitzen natürlich auch jeweils einen ASCII-Code. Schauen Sie einmal im Handbuch im Anhang 3 nach. Dort werden Sie finden, daß der Code für die Zahl 1 der 49 und für die Zahl 9 der 57 entspricht.

Wir brauchen nun nur noch all unsere bisher gewonnenen Kenntnisse, nämlich wie wir den Inhalt des Akkumulators auf dem Bildschirm ausgeben, wie wir ihn mit Werten laden und seinen Inhalt erhöhen, wie wir den Inhalt eines Registers erniedrigen und auf 0 abfragen, zusammenzufassen, um das angeschnittene Problem zu lösen. Wenn Sie bereits eine Idee entwickelt haben, wie Sie die Zahlen 1 bis 9 auf dem Bildschirm abbilden können, sollten Sie an dieser Stelle das Buch schließen und versuchen, die Aufgabe in eigener Regie zu lösen. Wenn es nicht klappt, ärgern Sie sich bitte nicht. Die Lösung folgt auf dem Fuße.

Dazu sehen wir uns zunächst den Lösungsweg anhand eines Flußdiagramms (Abb. 2.4) an.

Aufgabe: Stelle die Ziffern 1 bis 9 auf dem Bildschirm dar!

In ein Programm übersetzt, sieht das so aus:

Programm 2.4

```
        ENT             Programmstart setzen
        LD    C,9       C mit der Zeichenanzahl laden, d.h. mit 9
        LD    A,49      A mit ASCII-Code für 1 laden
NXT:    CALL  47962     Inhalt von A auf dem Bildschirm ausgeben
        INC   A         Nächstes Codezeichen (für 2)
        DEC   C         Zähler erniedrigen
        JR    NZ,NXT:   Wenn C nicht 0, dann springe zu NXT:
        RET             Zurück, wenn C=0
```

Bevor Sie das Programm starten, wollen wir es auf dem Papier Schritt für Schritt ablaufen lassen:

Schritt Nr.	Akkumulator	Register C	Z-Bit
1	49	9	0
2	50	8	0
3	51	7	0
4	52	6	0
5	53	5	0
6	54	4	0
7	55	3	0
8	56	2	0
9	57	1	0
10	50	0	1

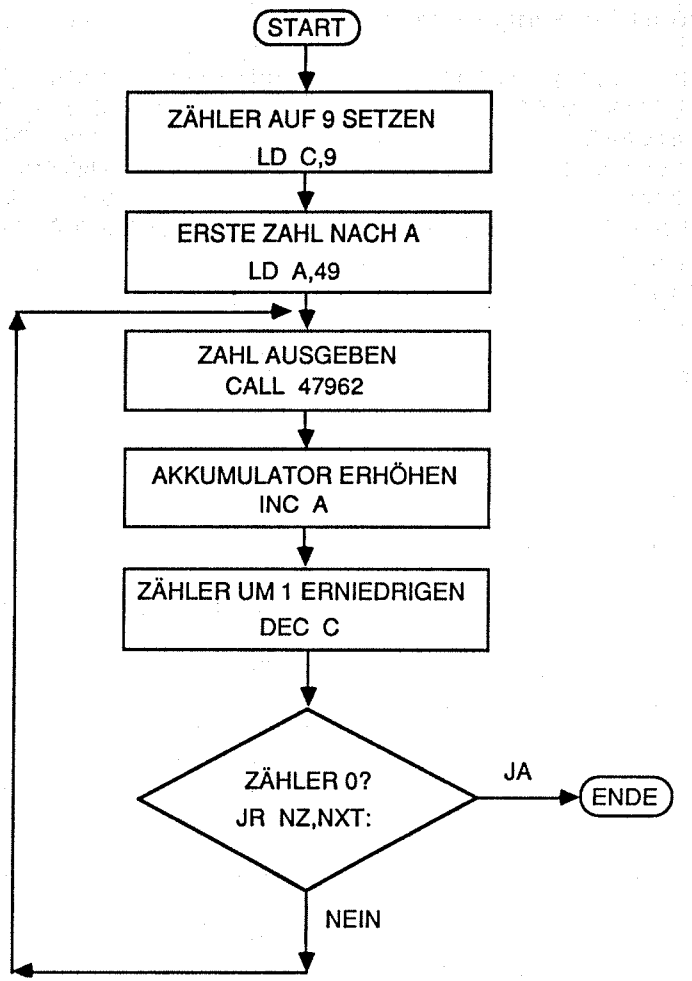


Abb. 2.4. Das Flußdiagramm zu Programm 2.4

Wenn das Z-Bit gesetzt wird (1), ist das Programm beendet. Es kehrt dann zum Assembler zurück (Schritt 10)

Jetzt assemblieren Sie bitte das Programm und lassen es laufen.

Übungsaufgabe 2.3

Schreiben Sie ein Programm, das das Alphabet auf dem Bildschirm ausgibt. (Hinweis: Der ASCII-Code für den Buchstaben A lautet 65.)

Sie finden eine mögliche Lösung im Anhang G.

Bisher haben wir nur eines von sieben Statusbits, das Nullbit, eingesetzt. Jetzt wollen wir uns die übrigen ansehen.

Die Statusbits

Das Übertragsbit (C-Bit)

Dieses Statusbit wird immer dann gesetzt, wenn eine Addition oder eine Subtraktion zu einem Übertrag (oder einem Borger) führt; wenn nicht, wird es gelöscht. Darüber hinaus wird es durch einige einfache oder zyklische Verschiebeoperationen beeinflußt.

Das N-Bit

Dieses Bit wird weniger im Rahmen von Programmen bei arithmetischen Operationen als vielmehr vom Z80 selbst zur Kontrolle interner Abläufe benutzt.

Das Paritäts-/Überlauf-Bit (P/V-Bit)

Dieses Statusbit übernimmt zwei verschiedene Aufgaben. Zum einen zeigt es die Parität eines Ergebnisses an. Diese wird dadurch ermittelt, daß alle Bits mit dem Wert logisch 1 addiert werden. Führt diese Addition zu einer geraden Zahl, dann wird das Parity-Bit gesetzt, im anderen Fall zurückgesetzt (auf 0). Zum anderen wird dieses Statusbit auch dann gesetzt, wenn im Verlauf arithmetischer Operationen ein Überlauf auftritt.

Das Zwischenübertrags-Bit (H-Bit)

Dieses Statusbit wird vom Z80 bei arithmetischen Operationen mit BCD-Zahlen (Binary Coded Decimal) ausgewertet. Wir werden an anderer Stelle darauf zurückkommen.

Das Nullbit (Z-Bit)

Das Nullbit wird immer dann gesetzt, wenn das Ergebnis einer Operation 0 ist. Es spielt im Zusammenhang mit Vergleichsvorgängen eine zentrale Rolle.

Das Vorzeichenbit (S-Bit)

Dieses Statusbit signalisiert das Vorzeichen eines arithmetischen Ergebnisses oder eines zu übertragenden Bytes. Es wird immer dann gesetzt, wenn das höchstwertige Bit (MSB) eines Bytes den Wert 1 aufweist.

Unterprogramme

Wir hatten den bereits eingeführten JP-Befehl mit dem BASIC-Statement IF X=Y THEN 500 verglichen. Auch für den BASIC-Befehl GOSUB existiert ein ähnlicher Befehl auf der Maschinenebene. Dieser Befehl lautet CALL. Er wurde bisher schon dazu benutzt, um den Inhalt des Akkumulators auf dem Bildschirm auszugeben.

CALL nn Rufe das Unterprogramm auf, das bei der Adresse nn beginnt (nn kann durch ein Label angegeben werden).

Wie unter BASIC auch, wird zum Abschluß eines Unterprogramms ein RETURN-Befehl benötigt.

RET Springe aus dem Unterprogramm ins aufrufende Programm zurück.

Wir wollen uns nun einmal ansehen, wie wir die genannten Befehle in einem Programm einsetzen können.

Aufgabe: Gib die ASCII-Zeichen mit den Codes 200 bis 250 auf dem Bildschirm aus. Verwende dazu ein Unterprogramm für die Zeichenausgabe.

Programm 2.5

```
        ENT
        LD      A,200       A = 1. ASCII-Code
        LD      B,50        B = Zählregister
WEITER: CALL    PRINT:      Aufruf der Ausgaberoutine
        DEC     B           Zähler erniedrigen
        INC     A           Nächster ASCII-Code
        JR      NZ,WEITER:  Wenn B verschieden von 0,
                            dann springe zu WEITER:
        RET
PRINT:  CALL    47962
        RET
```

Übersetzen Sie das Programm, und starten Sie es!

Um das Programm durchschaubarer zu gestalten, ist der Ausgaberoutine ein Label mit dem Namen PRINT zugeordnet worden. Bei Unterprogrammaufrufen gibt es, wie bei einfachen Sprüngen auch, unbedingte und bedingte Aufrufe.

CALL cc,nn Springe ins Unterprogramm, das bei der Adresse nn beginnt, wenn die unter cc angegebene Bedingung erfüllt ist.

Die Bedingungen entsprechen hier genau denjenigen, die Sie bereits von den bedingten Sprüngen her kennen. Für den Fall, daß ein CALL-Befehl ausgeführt wird, wird der Inhalt des Programmzählers zwischengespeichert. Der PC wird dann mit der unter nn vereinbarten Adresse geladen. Nach Beendigung des Unterprogramms, d.h. wenn der RET-Befehl erreicht ist, wird die zwischengespeicherte Adresse wieder in den Programmzähler zurückgeladen.

Aufgabe: Geben Sie 100mal den Buchstaben A auf dem Bildschirm aus. Nach jeder zehnten Ausgabe soll ein Leerzeichen eingefügt werden. In Form eines Flußdiagramms sieht die Lösung wie in Abb. 2.5 gezeigt aus.

Nachfolgend das entsprechende Programm:

Programm 2.6

```
        ENT
        LD      C,100       C dient als 1. Zählregister
        LD      B,10        B dient als 2. Zählregister
        LD      A,65        A = "A"
WEITER: CALL    47962       Akkumulator ausgeben
        DEC     B
        CALL    Z,LEERZ:    Wenn B=0, gib ein Leerzeichen aus
        DEC     C
        JR      NZ,WEITER:  Bereits 100 Zeichen A ausgegeben?
                            Wenn nicht, im Programm bei WEITER:
                            fortfahren
        RET                 Zurück zum Assembler
LEERZ:  LD      A,32        Lade den ASCII-Code für das Leerzeichen in den Akkumulator
        CALL    47962       Leerzeichen auf dem Bildschirm ausgeben
        LD      A,65        ASCII-Code für "A" nach A
        LD      B,10        Zähler B wieder mit 10 laden
        RET                 Zurück ins Hauptprogramm
```

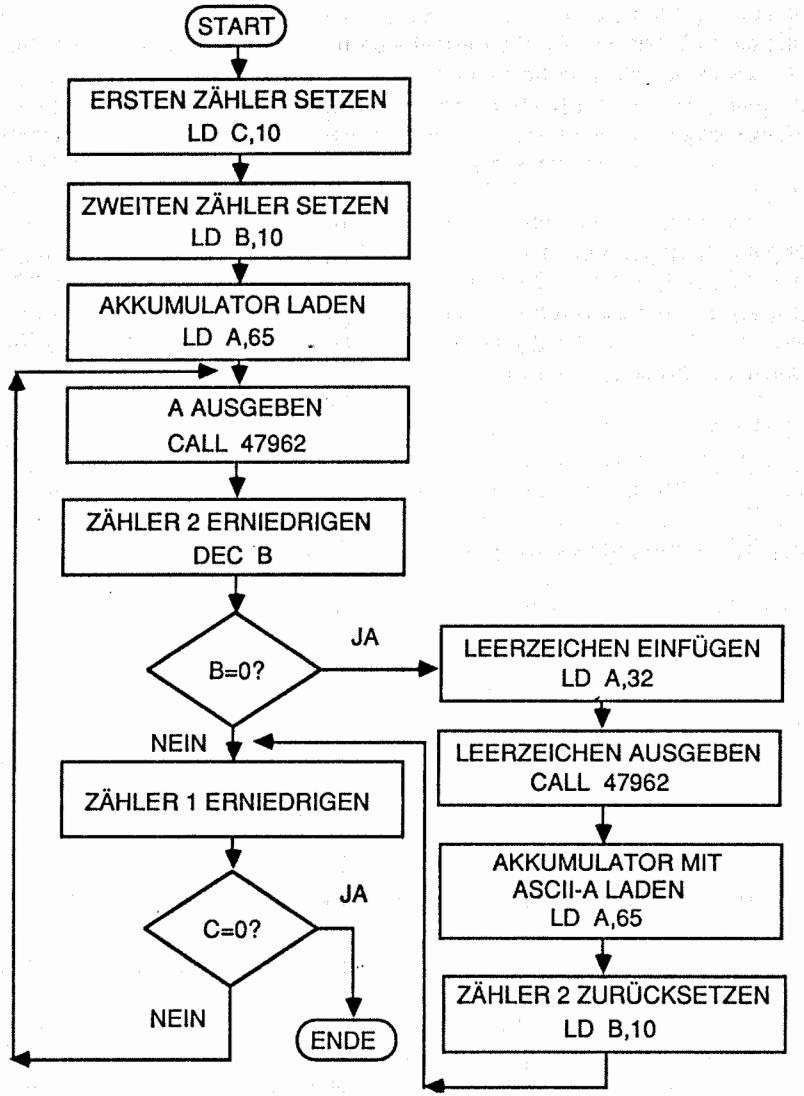


Abb. 2.5: Das Flußdiagramm zu Programm 2.6

Lassen Sie das Programm nun ablaufen. Es werden 100 A's ausgegeben. Nach jedem zehnten Buchstaben folgt ein Leerzeichen. Die zuvor erläuterte Art des bedingten Unterprogrammaufrufs erweist sich im Zusammenhang mit der strukturierten Programmierung als äußerst nützlich.

Zusammenfassung

Der Programmzähler (PC bzw. PC-Register genannt) ist ein 16-bit-Register, das immer die Adresse des aktuell auszuführenden Programmteils enthält.

Das Statusregister ist 8 bit lang. Jedes Bit enthält eine Information über den logischen Zustand des Rechenwerks des Z80.

Im Zusammenhang mit Sprungbefehlen können Labels verwendet werden. Ein Label darf bei dem hier verwendeten Assembler nicht mehr als 6 Zeichen lang sein. Es muß mit einem Doppelpunkt enden und durch ein Leerzeichen vom nachfolgenden Befehlsteil getrennt sein.

Es gibt zwei Arten von Sprüngen, unbedingte und bedingte. Sprünge können sowohl absolut als auch relativ vereinbart werden. Beispiel: JP WEITER: oder JR WEITER:.

Unterprogrammaufrufe mittels des CALL-Befehls können ebenfalls unbedingter oder bedingter Natur sein. Zu jedem CALL-Befehl muß ein entsprechender RET-Befehl existieren.

Die nachfolgende Liste von Befehlen sollten Sie ohne Schwierigkeiten richtig deuten können, auch wenn Sie nicht alle Statusbedingungen an dieser Stelle bereits vollständig verstehen.

```
JP  nn
JP  cc,nn
JR  e
JR  Z,e
JR  NZ,e
```

Hierbei ist

nn eine absolute Adresse oder ein Label;
cc einer der Operanden, der festlegt, welches Statusbits abgefragt werden soll;
e eine absolute Adresse mit der Einschränkung, daß sie im Bereich zwischen -126 bis +129 bezogen auf die aktuelle Adresse liegen muß. Es kann auch ein Label angegeben werden.

Bedeutungen von cc:

NZ	nicht Null
Z	Null
NC	Kein Übertrag
C	Übertrag

Bedeutungen von cc (Fortsetzung):

PO	ungerade Parität
PE	gerade Parität
P	positiv
M	negativ

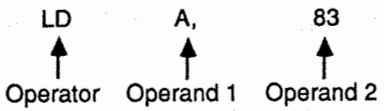
Außerdem sollten Sie in der Lage sein, folgende Befehle zu verstehen:

INC r
DEC r

Hierbei ist r eines der folgenden 8-bit-Register:

B, C, D, E, H, L, A

Operand und Operator sind wie folgt definiert:



Kapitel 3

Registerpaare und Adressierungsarten

Bisher können wir ein Register nur mit einem 8-bit-Wort laden. Der Z80 besitzt jedoch einige Register, die zu Registerpaaren zusammengefaßt werden können. Dann können auch Datenworte von 16 bit Länge abgespeichert und verarbeitet werden. Die nachfolgend angegebenen Register können derartige Registerpaare bilden:

B und C
D und E
H und L

Die Befehle, mit deren Hilfe 8-bit-Datenworte in einzelne Register geladen werden können, haben wir bereits kennengelernt. Der Befehl, der uns erlaubt 16-bit-Daten zu laden, lautet allgemein:

LD dd,nn Lade das Registerpaar dd mit dem16-bit-Datenwort nn.

Im vorliegenden Fall steht dd für eines der Registerpaare BC, DE, HL oder für das Register SP. nn ist ein 16-bit-Datenwort.

Anmerkung: Das mit SP bezeichnete Register ist ein spezielles Register. Es wird Stapelzeiger (SP = Stack Pointer) genannt. Wir kommen zu einem späteren Zeitpunkt noch darauf zurück.

Wenn wir ein Registerpaar in der zuvor erwähnten Art laden, dann führen wir den Ladevorgang mit den vereinbarten Daten unmittelbar aus. Diese Art der Adressierung wird deshalb als unmittelbare Adressierung bezeichnet. Da der Operand hier 2 byte lang ist, spricht man auch von unmittelbar *erweiterter* Adressierung.

Alle bisher vorgestellten Programme haben Daten ausschließlich in Registern abgespeichert. Das ist so lange möglich, wie wir keine großen Datenmengen zu verarbeiten haben. Unter praktischen Gegebenheiten benötigen wir einen größeren Speicherumfang, als die sieben 8-bit- oder drei 16-bit-Register bieten können.

Wir brauchen folglich Befehle, die es uns erlauben, Daten im Arbeitsspeicher abzulegen oder aus diesem auszulesen. Beim Z80 ist es möglich, den Inhalt von Registern im Speicher abzulegen und den Inhalt von Speicherzellen in Register zu überführen.

Diese Form der Adressierung wird direkte Adressierung genannt, weil sie die Inhalte von Registern oder Speicherzellen direkt benutzt, ohne daß zuvor etwas mit den Daten geschieht. Die allgemeine Form eines derartigen Ladebefehls lautet:

LD (nn),dd Lade die durch nn definierte Speicherzelle mit dem Inhalt des Registerpaares dd.

Ein Beispiel für diesen Befehl ist:

```
LD  (200), BC
```

Dieser Befehl überträgt den Inhalt des Registerpaars BC in die Speicherzelle mit der Adresse 200. Beachten Sie bitte, daß die Adresse in Klammern angegeben werden muß!

Der Befehl, mit dessen Hilfe der Inhalt einer Adresse in ein Registerpaar übertragen wird, lautet dagegen:

LD dd,(nn) Lade das Registerpaar dd mit dem Inhalt der durch nn angegebenen Speicherzelle.

Auch hierfür ein Beispiel:

```
LD  BC,(200)
```

Was bewirkt dieser Befehl? Er überführt natürlich den Inhalt der durch die Adresse 200 gekennzeichneten Speicherzelle in das Registerpaar BC.

So weit, so gut! Jetzt wollen wir diese Befehle in Programmen anwenden.

Im bisherigen Verlauf dieses Kurses haben wir Zeichen auf dem Bildschirm durch einen Unterprogrammaufruf mittels eines CALL-Befehls ausgegeben. Das benutzte Unterprogramm ist nicht die einzige verfügbare Routine des Systems. Der Festwertspeicher (ROM) des Schneider CPC enthält ebenfalls einige Grafikroutinen, die uns unter anderem die Möglichkeit eröffnen, Linien auf dem Bildschirm zu zeichnen. Wir wollen uns an dieser Stelle nicht darum kümmern, wie diese Routinen arbeiten, sondern nur darum, wie wir sie uns nutzbar machen können. Wenn wir in BASIC eine gerade Linie vom Koordinatennullpunkt zum Punkt mit den Koordinaten 400,200 zeichnen möchten, benutzen wir dazu die Anweisung:

```
DRAW 400,200
```

Hierbei ist angenommen, daß der Grafikcursor gerade am Punkt mit den Koordinaten X=0,Y=0 steht. Notfalls stellen Sie dies durch Eingabe von


```
PLOT 0,0
```

sicher.

Wenn wir jene Grafikroutine benutzen, die unter der Adresse 48118 aufgerufen werden kann, sind wir auch in Maschinensprache in der Lage, die zuvor formulierte Aufgabe zu lösen. Unter BASIC werden die Zielkoordinaten als Bestandteil des DRAW-Statements eingegeben. Für den Fall, daß wir Maschinensprache-Routinen verwenden, übergeben wir die Daten (hier also die Koordinatenwerte X und Y) über Register. Das Registerpaar DE wird im vorliegenden Fall mit dem X-Wert, das Paar HL mit dem Y-Wert geladen.

Wir wollen alle für die Grafikroutine benötigten Angaben einmal zusammenfassen:

Die Startadresse lautet 48118.
Die Parameter sind X und Y.
X wird im Registerpaar DE gespeichert.
Y wird im Registerpaar HL gespeichert.

Jetzt wollen wir unser Vorhaben ausführen und eine Linie zu Punkt 400,200 zeichnen.

Das Programm dazu sieht so aus:

Programm 3.1

```
ENT
LD      DE,400
LD      (35000), DE
LD      HL, 200
LD      (35002), HL
LD      DE, (35000)
LD      HL, (35002)
CALL    48118
RET
```

Geben Sie das Programm ein, und starten Sie es.

Wir wollen uns nun das Programm einmal näher anschauen. Als erstes richten wir unser Augenmerk darauf, daß der Inhalt des Registerpaares DE unter der Adresse 35000 abgespeichert wird, der von HL dagegen unter der Adresse 35002. Warum wurde der Inhalt von HL nicht unter 30001 abgelegt? Der Grund dafür ist, daß jede Speicherzelle nur einen 8-bit-Datenwert aufnehmen kann. Wir müssen

aber im vorliegenden Fall den Inhalt eines 16-bit-Registers, also 16 Bits, abspeichern. Das Datenwort von 16 bit Wortlänge wird einfach in zwei Hälften zerlegt, die aufeinanderfolgend abgespeichert werden. Um ein 16-bit-Datenwort abspeichern zu können, werden folglich zwei Speicherzellen benötigt. Sehen Sie sich dazu die Abb. 3.1 an.

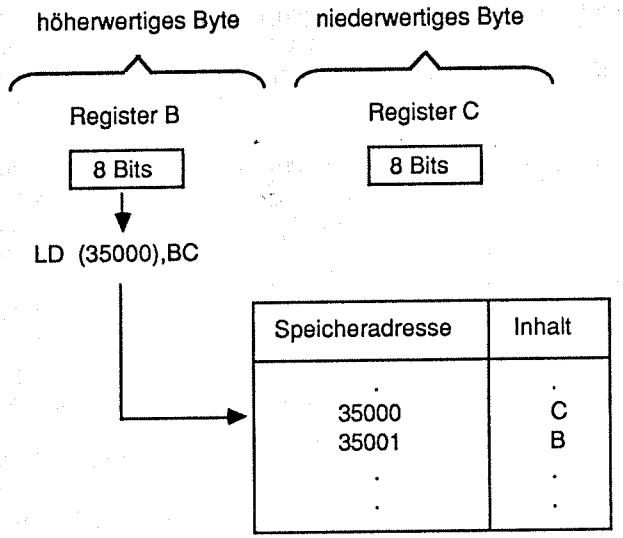


Abb. 3.1: Ablegen des Inhalts eines 16-bit-Registerpaares im Speicher

Das niederwertige Byte wird zuerst abgespeichert, unmittelbar gefolgt vom höherwertigen Byte.

Übungsaufgabe 3.1

Notieren Sie sich die Reihenfolge, in der die Register mit Daten gefüllt werden, wenn die nachfolgend angegebenen Befehle ausgeführt werden:

```
LD  (200),DE
LD  (202),HL
```

Schlagen Sie im Anhang G nach, wenn Sie eine Antwort wünschen.

Bei der Bildung von Registerpaaren nehmen die C-, E- und L-Register die niederwertigen Bytes und B, D sowie H die höherwertigen Bytes auf.

Übungsaufgabe 3.2

Schreiben Sie ein Maschinenprogramm, das die Speicherzelle mit der Adresse 35000 mit dem Wert 100 und die Zelle mit der Adresse 35002 mit dem Wert 400 füllt und anschließend eine gerade Linie zu der so vereinbarten Koordinate zeichnet.

Achten Sie bitte darauf, daß der Grafikcursor zunächst auf 0,0 zurückgesetzt werden muß. Eine Möglichkeit besteht darin, daß Sie zu diesem Zweck über das Kommando X nach BASIC zurückspringen und anschließend den Befehl PLOT 0,0 eingeben. Ein etwas abenteuerlicherer Weg sieht so aus: Das ROM des Schneider CPC enthält ebenfalls eine Routine, um einen Punkt auf dem Bildschirm auszugeben. Wir können diese also in einem Programm verwenden. Auf diese Weise sparen wir uns den Rücksprung zu BASIC.

Der Aufruf ist von der Form:

```
Startadresse: 48106
DE = X-Koordinate
HL = Y-Koordinate
```

(Diese Vereinbarungen entsprechen somit denen der zuvor erläuterten Routine zur Ausgabe einer Linie.)

Um einen Punkt bei 0,0 zu zeichnen, müssen wir also DE und HL mit dem Wert 0 laden und anschließend das Unterprogramm unter der Adresse 48106 aufrufen.

Auch zu dieser Übungsaufgabe finden Sie eine Lösung im Anhang G.

Übungsaufgabe 3.3

Schreiben Sie ein Programm, das die nachfolgend angegebenen Punkte miteinander verbindet.

X	Y
200	300
400	200
0	0

Anmerkung: Starten Sie mit 200,300. Erinnern Sie sich dazu daran, daß DE mit der X- und HL mit der Y- Koordinate geladen werden muß.

Eine mögliche Lösung finden Sie im Anhang G.

Der Z80 unterstützt noch weitere Adressierungsarten. Bis zu diesem Punkt haben wir die unmittelbare Adressierungsart für 8- und 16-bit-Daten sowie die direkte Adressierung für 16-bit-Werte kennengelernt. Jetzt wollen wir uns um die indirekte Adressierung kümmern.

Die indirekte Adressierung

Bei der indirekten Adressierungsart wird der Inhalt eines Registers dazu benutzt, um auf eine Speicherzelle zu verweisen. Das erweist sich als sehr nützlich, denn wir können beispielsweise so ein 16-bit-Register mit der Startadresse eines Datenbereichs laden und diese Daten dadurch nacheinander einlesen, daß wir fortlaufend den Inhalt des 16-bit-Registers um 1 erhöhen.

Die zu diesem Adressierungstyp gehörende Befehlsfamilie lautet:

LD r, (HL) Lade das Register r mit dem Inhalt jener Speicherzelle, auf die der Inhalt von HL verweist. (Das heißt ganz einfach, daß wir HL mit der Adresse der entsprechenden Speicherzelle laden.)

LD (HL), r Lade die Speicherzelle, auf die HL verweist, mit dem Inhalt des Registers r.

LD A, (BC) Lade den Akkumulator mit dem Inhalt der Speicherzelle, auf die BC verweist.

LD A, (DE) Lade den Akkumulator mit dem Inhalt der Speicherzelle, auf die DE verweist.

LD (BC),A Lade die Speicherzelle, auf die BC verweist, mit dem Inhalt des Akkumulators.

LD (DE),A Lade die Speicherzelle, auf die DE verweist, mit dem Inhalt des Akkumulators.

Alle zuvor angegebenen Befehle werden im Zusammenhang mit Programmbeispielen erläutert.

Zu diesem Zweck wollen wir die ASCII-Codes für drei aufeinanderfolgende Zeichen A im Speicher ablegen und diese dann mit Hilfe des HL-Registers als "Zeiger" auf dem Bildschirm ausgeben.

Um die Ladeoperation zu vereinfachen, benutzen wir BC als einen variablen Zeiger. Wenn somit der Akkumulator den Code für A (65) enthält, können wir über

die Befehlsfolge LD (BC),A und INC BC die drei Zeichen in aufeinanderfolgende Speicherzellen laden. Diese werden über BC adressiert, dessen Inhalt dreimal um 1 erhöht wird.

Das zugehörige Programm sieht so aus:

Programm 3.2 (Bitte noch nicht assemblieren)

```
ENT
LD      BC,35000    BC=Startadresse des Datenspeichers
LD      A,65        A=ASCII-Code für das Zeichen A (65)
LD      (BC),A      Das erste A abspeichern
INC     BC          Zeiger um 1 erhöhen
LD      (BC),A      Das zweite A abspeichern
INC     BC          Zeiger um 1 erhöhen
LD      (BC),A      Das dritte A abspeichern
```

Jetzt schreiben Sie den Programmteil, der die Daten beginnend bei der Adresse 35000 aus dem Speicher ausliest und auf dem Bildschirm darstellt.

```
LD      HL,35000    HL enthält die Startadresse
LD      A,(HL)      Der Inhalt des über HL adressierten Speichers wird in
                    den Akkumulator übertragen
CALL    47962       Inhalt von Register A auf dem Bildschirm ausgeben
INC     HL          Adreßzeiger um 1 erhöhen
LD      A,(HL)      A erneut laden
CALL    47962       Zweiten Buchstaben ausgeben
INC     HL          Adreßzeiger um 1 erhöhen
LD      A,(HL)      A erneut laden
CALL    47962       3. Buchstaben ausgeben
RET                 Zurück ins aufrufende Programm
```

Übersetzen Sie nun das gesamte Programm, und starten Sie es. Es erfüllt zwar seinen Zweck, ist aber nicht sehr effizient geschrieben. Wir wollen uns deshalb einmal ansehen, wie wir es verbessern können.

Zunächst wenden wir uns dem ersten Teil zu:

Die Befehlsfolge

```
LD  (BC),A
INC BC
```

wird dreimal hintereinander verwendet, um die Daten in den Speicher einzuschreiben. Warum verwenden wir zu diesem Zweck keine Programmschleife? Wir können ein nicht benötigtes Register - beispielsweise E - mit dem Zählwert 3 laden, dessen Inhalt nach jedem Ladevorgang um 1 erniedrigen und daraufhin abfragen, ob er 0 geworden ist. Ist dies der Fall, beenden wir die Schleife; wenn nicht, springen wir auf den Anfang der Schleife zurück. Auf diese Weise erledigen wir die uns gestellte Aufgabe erheblich eleganter und codesparender.

Der erste Teil des Programms sieht nach der Änderung dann so aus:

Programm 3.3 (bitte noch nicht assemblieren!)

```
        ENT
        LD    BC, 35000
        LD    A, 65
        LD    E, 3          Zählregister laden
WEITER: LD    (BC), A
        INC   BC
        DEC   E             Zählregister erniedrigen
        JR    NZ, WEITER:   wenn nicht 0, dann zurück zu WEITER:
```

Wegen der Verwendung eines Labels (WEITER:) kann die Berechnung der Sprungadresse entfallen.

Bei der Eingabe des Programmtextes ergibt sich im vorliegenden Fall keinerlei Zeitersparnis. Aber stellen Sie sich vor, Sie müßten 200 A's auf die zuerst angewandte Art abspeichern!

Im zweiten Teil des Programms können wir die gleiche Technik benutzen.

Übungsaufgabe 3.4

Fügen Sie den zweiten Teil des Programms hinzu, in dem ebenfalls eine Schleife für die Ausgabe der 3 A's eingesetzt wird, und lassen Sie das Programm dann ablaufen!

Übungsaufgabe 3.5

Schreiben Sie ein Programm, mit dessen Hilfe das Alphabet im Speicher abgelegt und anschließend ausgelesen sowie auf dem Bildschirm ausgegeben wird.

Eine Lösung finden Sie im Anhang G.

An dieser Stelle wollen wir uns nochmals die bisher behandelten Adressierungsarten ins Gedächtnis zurückrufen:

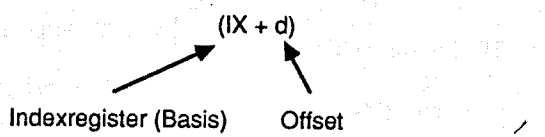
```
Register-Register   Unmittelbar
LD  B,A             LD A,83 oder LD BC,300

Direkt              Indirekt
LD  A,(2000)        LD  B,(HL)
```

Eine vollständige Übersicht über alle Z80-Mnemonics finden Sie im Anhang A. Die meisten der Ladebefehle für 8-bit- oder 16-bit-Daten sollten Sie allerdings bereits jetzt verstehen können.

Die indizierte Adressierung

Im Programm 3.3 haben wir ein Registerpaar benutzt, um auf einen Block sequentiell abgespeicherter Daten zuzugreifen. Diese Methode ist in vielen Fällen sehr brauchbar. Mit dem Z80 gibt es jedoch noch ein alternatives Verfahren. Neben den bisher bereits vorgestellten Registern gibt es beim Z80 nämlich noch zwei Indexregister IX und IY mit 16 bit Wortbreite. Wenn Sie beispielsweise nach dem Wort "Speicher" im Sachregister (Index) dieses Buchs suchen, werden Sie wahrscheinlich zunächst nach dem Beginn der Wortgruppe mit dem Anfangsbuchstaben S suchen und erst von dort aus die Suche nach dem gewünschten Wort "Speicher" fortsetzen. Der Beginn der Gruppe ist so etwas wie eine Ausgangsbasis, von der Sie Ihre Suche starten. Dieser Vorgang ist direkt vergleichbar mit der indizierten Adressierung. Das IX- oder das IY-Register wird zunächst mit einer Basisadresse geladen. Zu dieser Basisadresse wird dann ein Offset addiert, um auf einen bestimmten Teil der Information zuzugreifen.

Die allgemeine Form des indizierten Befehlsteils einer entsprechenden Anweisung sieht so aus:



Der Parameter d kann eine Zahl im Bereich von -127 bis +128 enthalten. Nachfolgend ein Beispiel für das Indexregister IX:

```
LD  A,(IX+3)
```

Was bewirkt diese Anweisung?

Nehmen wir einmal an, daß IX den Wert 200 enthält. Welcher Wert wird dann in den Akkumulator geladen? Schauen Sie sich dazu die folgende Datentabelle an.

Speicher-adresse	Inhalt
200	1
201	2
202	3
203	4
204	5

Der Akkumulator enthält nach der Ausführung des Befehls den Wert 4. Welchen Offset hätten Sie vereinbaren müssen, um den Akkumulator mit dem Wert 3 zu laden? Die Antwort lautet: (IX+2).

Anstelle des IX-Registers hätten wir auch vom Indexregister IY Gebrauch machen können:

```
LD   A,(IY+3)
```

Wir wollen uns nachfolgend einmal alle Ladebefehle ansehen, die sich des Indexregisters IX oder IY bedienen.

Befehle mit indizierter Adressierung

Es gibt eine abgekürzte Schreibweise zur anschaulichen Erläuterung der Wirkung eines Befehls, d.h. eine sogenannte symbolische Operation. Ein Beispiel dafür lautet:

Befehl	Symbolische Operation
LD r, (IX+d)	r ← (IX+d)

Mit diesem Befehl wird das Register r mit dem Inhalt derjenigen Speicherzelle geladen, auf die nach Addition des Inhalts von IX und d verwiesen wird. Der Anhang A enthält eine vollständige Tabelle mit den Z80-Befehlen einschließlich deren symbolischer Operationen.

Die Ladebefehle mit indizierter Adressierung lauten:

Befehl	Symb. Operation
LD r,(IX+d)	r ← (IX+d)
LD r,(IY+d)	r ← (IY+d)
LD (IX+d),r	(IX+d) ← r
LD (IY+d),r	(IY+d) ← r
LD (IX+d),n	(IX+d) ← n
LD (IY+d),n	(IY+d) ← n

Nun wollen wir uns ansehen, wie wir die Indexregister IX und IY selbst mit Werten laden können. Schauen Sie sich dazu die Tabelle der 16-bit-Ladebefehle an, und versuchen Sie jene zu finden, die diese Aufgabe erfüllen.

Hier sind Sie. Das Format sollte Ihnen bereits vertraut sein:

Befehl	Symb. Operation
LD IX,nn	IX ← nn
LD IY,nn	IY ← nn
LD IX,(nn)	IX ← (nn)
LD IY,(nn)	IY ← (nn)
LD (nn),IX	(nn) ← IX
LD (nn),IY	(nn) ← IY

Eine Anmerkung ist an dieser Stelle zu machen: Es ist nicht möglich, den Wert des Offsets d zu berechnen. Er muß immer absolut, d.h. als unmittelbarer Wert angegeben werden.

Damit Sie lernen, wie Indexregister eingesetzt werden, wollen wir das Programm 3.2 etwas umschreiben.

Das Programm lautet:

Programm 3.4

```
ENT
LD      IX,35000        IX = Basisadresse
LD      A,65
LD      (IX+0),A        Erstes A abspeichern
LD      (IX+1),A        Zweites A speichern
LD      (IX+2)          Drittes A speichern
```

Der zweite Teil des Programms, der den Speicherinhalt auf dem Bildschirm ausgibt, sieht dann wie folgt aus:

```
LD      A,(IX+0)
CALL    47962       Ersten Buchstaben ausgeben
LD      A,(IX+1)
CALL    47962       Zweiten Buchstaben ausgeben
LD      A,(IX+2)
CALL    47962       Dritten Buchstaben ausgeben
RET
```

Starten Sie nun bitte das Programm.

Da der Z80 bei indizierten Operationen zur Basisadresse erst den Offset hinzuaddieren muß, benötigt dieser Adressierungstyp mehr Zeit als eine indirekte oder unmittelbare Anweisung. Da der Offset d außerdem in absoluter Form angegeben werden muß, erweist sich die indizierte Art der Adressierung als nicht so günstig innerhalb von Schleifen, weil d nicht durch einen Befehl erhöht werden kann. Der Hauptvorteil der Indexregister IX und IY liegt darin, daß mit ihrer Hilfe auf Daten zugegriffen werden kann, die im Speicher relativ zu einer bekannten Adresse abgelegt sind. D.h. sie werden für den Zugriff auf Datentabellen verwendet, in denen verschiedene Informationen mit bekannten Offsets abgelegt sind.

Übungsaufgabe 3.6

Schreiben Sie ein Programm, das im Zusammenhang mit der Abspeicherung von fünf Buchstaben (beispielsweise für einen Namen) vom Indexregister IX Gebrauch macht. Geben Sie anschließend den dritten und den vierten Buchstaben auf dem Bildschirm aus!

Die nachfolgend angegebene Tabelle soll Ihnen bei der Lösung helfen:

Speicheradresse	Buchstabe	ASCII-Code
35000	S	83
35001	U	85
35002	S	83
35003	A	65
35004	N	78

Eine Lösung finden Sie wie gewöhnlich im Anhang G.

Wir haben wieder das Ende eines Kapitels erreicht. Alles was wir jetzt noch tun sollten, ist, unser bisher in diesem Kapitel erworbenes Wissen noch einmal in Erinnerung zu rufen. Wenn Sie sich bereits sicher genug fühlen, sollten Sie ruhig schon eigene Programme schreiben. Versuchen Sie beispielsweise einmal, ein Zeichen auf dem Bildschirm auszugeben und es anschließend dadurch wieder auszulöschen, daß Sie es mit einem Leerzeichen überschreiben. Das könnte schon die Ausgangsbasis für ein Bildschirmspiel sein.

Zusammenfassung

Die folgenden Begriffe sollten Ihnen inzwischen vertraut sein:

1. Registerpaare
2. Die Adressierungsarten
   a. Register-Register
   b. unmittelbar
   c. direkt
   d. indirekt
   e. indiziert
3. Symbolische Operation
4. Obgleich noch nicht speziell eingeführt, sollten Ihnen die nachfolgenden Kommandos sofort etwas sagen (ss ist ein Registerpaar).

   ```
   INC ss
   DEC ss
   ```

5. Sie sollten die nachfolgend angegebenen drei Befehle verstehen. Sie wurden bereits kurz in Kapitel 2 angerissen.

   ```
   DEC  (HL)
   DEC  (IX+d)
   DEC  (IY+d)
   ```

Kapitel 4

Arithmetische Operationen

Viele praktische Aufgaben machen die Verarbeitung von Zahlenwerten erforderlich. Bisher haben wir nur gelernt, wie wir den Inhalt von Registern oder Speicherzellen erhöhen oder erniedrigen können. Der Z80 kennt noch einige andere arithmetische Befehle, die zwar nicht sehr komfortabel sind, aber dennoch eine gute Ausgangsbasis für leistungsfähige Operationen darstellen.

Das vorliegende Kapitel ist diesen Befehlen gewidmet. Falls Sie noch nicht mit der binären oder der hexadezimalen Darstellung von Zahlen vertraut sind, sollten Sie sich erst den Anhang E ansehen.

Sie werden bald erkennen, warum wir uns der hexadezimalen Zahlendarstellung bedienen. Ein Grund ist beispielsweise der, daß sie die Zahleneingabe sehr vereinfacht.

Wir können die arithmetischen Befehle in zwei Gruppen unterteilen: die 8-bit- und die 16-bit-Befehle. Wir werden uns in diesem Kapitel vornehmlich mit der Gruppe der arithmetischen 8-bit-Befehle beschäftigen. Die16-bit-Befehle verhalten sich jedoch ähnlich.

Die Addition

Die am meisten benutzte arithmetische Operation ist die Addition zweier Zahlen. Hierfür wird der folgende Befehl verwendet:

ADD A,n Addiere den Wert n zum Inhalt des Akkumulators. Der ursprüngliche Inhalt wird durch das Ergebnis der Operation ersetzt.

Zur Erläuterung dieses Befehls wollen wir die Zahlen 65 und 20 addieren. Das Ergebnis lautet 85. Wenn wir dieses auf dem Bildschirm ausgeben, erscheint der Buchstabe U. (Dies ist der Buchstabe, der den ASCII-Code 85 besitzt.)

Geben Sie das folgende Programm ein.

Programm 4.1

```
        ENT
        LD      A,65        Lade A mit 65
```

```
ADD   A,20      Addiere 20 zum Inhalt von A
CALL  47962     Ergebnis ausgeben
RET
```

Was passiert? Es wird ein U auf dem Bildschirm erscheinen.

Der Assembler akzeptiert übrigens auch Zahlenangaben in hexadezimaler Notation. Damit er zwischen den unterschiedlichen Zahlentypen unterscheiden kann, werden hexadezimale Zahlen durch Voranstellen des Zeichens & gekennzeichnet.

Bei hexadezimaler Darstellung der Zahlen 47962, 65 und 20 sieht das Programm 4.1 aus wie nachstehend angegeben.

Programm 4.2

```
ENT
LD    A,&41
ADD   A,&14
CALL  &BB5A
RET
```

Bitte überzeugen Sie sich dadurch davon, daß dieses Programm exakt dem Programm 4.1 entspricht, daß Sie es eingeben und starten.

Übungsaufgabe 4.1

Schreiben Sie ein Programm, das die Zahlen 200 und 48 zusammenzählt und das Ergebnis auf dem Bildschirm ausgibt.

Übungsaufgabe 4.2

Schreiben Sie ein Programm, das die Zahlen &41 und &10 addiert. Lassen Sie das Ergebnis auf dem Bildschirm ausgeben.

Sie finden mögliche Antworten im Anhang G.

Bisher waren die Ergebnisse aller Berechnungen kleiner als 255. Sie lagen also niedriger als die maximal mit 8-bit-Datenworten darstellbare Zahl. Was glauben

Sie, wird passieren, wenn die beiden Zahlen 150 und 171 addiert werden? Versuchen Sie es mit dem nächsten Programm.

Programm 4.3

```
ENT
LD    A,150     Lade den Akkumulator mit der Zahl150
ADD   A,171     Addiere 171 zum Inhalt von A
CALL  47962     Ergebnis ausgeben
RET
```

Die Addition führt zu einem Überlauf des Akkumulators. Nach Erreichen der maximal darstellbaren Zahl 255 wurde der Akkumulator zurückgesetzt, um dann letztlich die Zahl 65 zu erreichen. Daher rührt also das A auf dem Bildschirm.

Das Programm ist nicht in der Lage, den Zustand der Statusbits abzufragen. Wenn es das getan hätte, würde es gesehen haben, daß in dem Moment, als der Akkumulator überlief, das Übertragsbit (C-Bit) den Wert 1 angenommen hat. Dieser Sachverhalt kann dazu ausgenutzt werden, um zwei Zahlen zu addieren, deren Summe 255 überschreitet. Bevor wir ein Programm hierzu schreiben, wollen wir die Aufgabe zunächst auf dem Papier lösen.

Aufgabe: Addiere 1157 zu sich selbst, d.h.: 1157 +1157 = ?

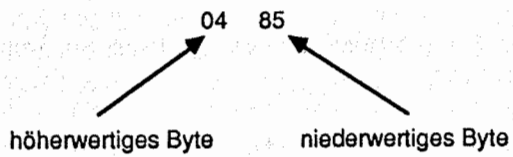
1. Schritt

Als erstes wollen wir die Dezimalzahl 1157 in eine Hexadezimalzahl umwandeln:

```
1157 : 4096 = 0  Rest   1157
1157 :  256 = 4  Rest    133
 133 :   16 = 8  Rest      5
   5 :    1 = 5  Rest      0
```

Die Zahl 1157 lautet also in hexadezimaler Schreibweise &0485.

Diese Hexadezimalzahl entspricht in dualer Schreibweise einer 16-bit-Zahl. Sie muß daher in zwei Bytes aufgeteilt werden, damit wir zu ihrer Verarbeitung 8-bit-Arithmetikbefehle verwenden können. In hexadezimaler Notation ist das ganz einfach (übrigens einer der Gründe dafür, daß wir diese Zahlendarstellung in der Computertechnik verwenden).



2. Schritt

Um die zwei Zahlen &0485 zusammenzuzählen, müssen wir zunächst die beiden niederwertigen und dann die beiden höherwertigen Bytes addieren. Ein bei der Addition des niederwertigen Teils auftretender Übertrag muß dabei berücksichtigt werden.

```
 85
+85
---
 0A  +  Übertrag
```

Das Ergebnis lautet also &0A plus Übertrag.

3. Schritt

Nun werden die höherwertigen Bytes unter Berücksichtigung des Übertrags addiert.

```
 04
+04
---
 08
```

Jetzt wird das Übertragsbit hinzuaddiert.

```
 01
+08
---
 09
```

4. Schritt:

Die beiden Bytes werden zu einem 16-bit-Wort zusammengefaßt:

1157 +1157 = &090A

Überzeugen Sie sich davon, daß &090A der Dezimalzahl 2314 entspricht.

Merken Sie sich bitte, daß bei allen Operationen mit doppelter Genauigkeit, d.h. bei der Verwendung von 16-bit-Werten, zuerst die niederwertigen Bytes und dann erst die höherwertigen Bytes verarbeitet werden, damit ein eventueller Übertrag berücksichtigt werden kann.

Bevor wir nun das entsprechende Additionsprogramm schreiben können, müssen wir uns noch mit einigen neuen Befehlen vertraut machen:

AND A Logische UND-Verknüpfung des Akkumulatorinhalts mit sich selbst.

Im aktuellen Zusammenhang brauchen Sie sich nur einen einzigen Effekt dieses Befehls zu merken: Er setzt das Übertragsbit auf 0. Warum brauchen wir dies? Die Antwort ist einfach: Wenn das Übertragsbit unbeabsichtigt gesetzt ist, wird das Ergebnis der Addition nicht richtig sein können (wenn der Befehl ADC benutzt wird), da der nicht beabsichtigte Übertrag im Ergebnis berücksichtigt wird.

Ein Befehl, der die Inhalte zweier Register unter Berücksichtigung eines Übertrags addiert, lautet:

ADC A,s Addiere den Inhalt des Register s plus Übertrag zum Inhalt des Akkumulators. Das Ergebnis wird im Akkumulator abgelegt.

Wie Sie sehen, verwendet der ADC-Befehl ein Register als einen seiner Operanden. Es gibt darüber hinaus auch einen Befehl ähnlich ADD A,n, der anstelle eines Datenwertes ein Register verwendet.

ADD A,s Addiere den Inhalt des Registers s zum Inhalt des Akkumulators. Das Ergebnis steht im Akkumulator.

Jetzt folgt das Programm.

Programm 4.4 (bitte noch nicht assemblieren)

```
ENT
LD    C,&85      Lade C mit dem 1. niederwertigen Byte
LD    A,&85      Lade A mit dem 2. niederwertigen Byte
AND   A          Übertragsbit zurücksetzen
ADD   A,C        Niederwertige Bytes addieren
LD    (&7000),A  Niederwertigen Teil abspeichern
LD    C,&04      Lade C mit dem 1. höherwertigen Byte
LD    A,&04      Lade A mit dem 2. höherwertigen Byte
ADC   A,C        Höherwertige Bytes mit Übertrag addieren
LD    (&7001),A  Höherwertigen Teil abspeichern
```

Dieses Programm zählt die beiden Zahlen zusammen. Wie aber kann das Ergebnis überprüft werden? Wir benötigen jetzt offensichtlich eine Routine, die uns das Ergebnis in angemessener Form darstellt. Warum können wir den Inhalt der beiden Speicherzellen nicht einfach auf dem Bildschirm ausgeben? Nun, die ASCII-

Codes &09 und &0A führen nicht zu sichtbaren Zeichen, sondern zu Kontrollzeichen. Wir müssen einen Offset hinzuaddieren, um sie in den Bereich des ASCII-Alphabets zwischen 65 und 122 zu bringen. Da der Code für A 65 lautet, könnten wir diese Zahl als vernünftigen Offset wählen. Fügen Sie also dem vorherigen Programm die nachfolgende Befehlsfolge hinzu:

Programm 4.4 (b)

```
LD      C,65
LD      A,(&7001)
ADD     A,C
CALL    &BB5A
LD      A,(&7000)
ADD     A,C
CALL    &BB5A
RET
```

Jetzt sollten Sie das ganze Programm assemblieren und anschließend starten. Auf dem Bildschirm werden ein J und ein K ausgegeben (entsprechend &09 + 65 und &04 + 65).

Wenn Sie das Programm 4.4 aufmerksam studiert haben, wird Ihnen aufgefallen sein, daß wir den Befehl AND A eigentlich gar nicht brauchen, da wir zuerst den Befehl ADD zur Addition benutzen, der das Übertrags-Bit ja nicht mit addiert. Nach dem ADD-Befehl ist das Übertrags-Bit dann für die folgende Addition auf jeden Fall richtig eingerichtet. Wir wollten jedoch einmal den Trick mit dem AND A-Befehl vorführen.

Übungsaufgabe 4.3

Schreiben Sie ein Programm, das die Zahlen 250 und 600 addiert. Verwenden Sie dazu die hexadezimale Schreibweise. Nehmen Sie als Offset für die beiden Bytes die Zahl 65. Geben Sie das Ergebnis auf dem Bildschirm aus.

Eine Lösung finden Sie in Anhang G.

Die Subtraktion

Neben den zuvor erläuterten Additionsbefehlen kennt der Z80 noch die folgenden Subtraktionsbefehle:

SUB s Der Befehl SUB subtrahiert den Operanden s vom Inhalt des Akkumulators. Das Ergebnis wird im Akkumulator abgespeichert.

SBC A,s Mit dem Befehl SBC wird der Operand s unter Berücksichtigung eines eventuell auftretenden Übertrags vom Inhalt des Akkumulators abgezogen. Das Ergebnis steht im Akkumulator.

Den Vorgang der Subtraktion wollen wir im Rahmen der folgenden Übungsaufgabe 4.4 kennenlernen. Versuchen Sie, diese zu lösen!

Übungsaufgabe 4.4

Schreiben Sie ein Programm, mit dessen Hilfe Sie die Zahl 9 von 233 abziehen. Geben Sie das Ergebnis anschließend auf dem Bildschirm aus. Sie brauchen diesmal keinen Offset hinzuzuaddieren. Warum nicht?

Übungsaufgabe 4.5

Schreiben Sie ein Programm, das das Ergebnis der folgenden Aufgabe berechnet:

(97 + 126) - 153 = ?

Sie finden mögliche Antworten im Anhang G.

Wie wir zuvor gesehen haben, führt die Addition zweier Zahlen mit einem Ergebnis, das größer als 255 ist, zu einem Überlauf. Als Folge davon wird das Übertragsbit gesetzt.

Genau das Gegenteil passiert dann, wenn Sie eine große Zahl von einer mit kleinerem Wert abziehen wollen. Schauen Sie sich in diesem Zusammenhang einmal das folgende Beispiel an:

```
 25
-17
----
  ?
```

Wir können im ersten Schritt die 7 nicht unmittelbar von der 5 abziehen. Es ist notwendig von der nächsten Zahl eine Einheit zu borgen. Wir wollen diese Einheit als "Borger" bezeichnen. Jetzt sieht der erste Schritt so aus:

```
  5     +    Borger   =   15
 -7                      -7
 --                      --
  ?                       8
```

Der erste Zwischenschritt führt also zum Ergebnis 8.

Da wir aus der Spalte mit der nächsthöheren Wertigkeit etwas geborgt haben, müssen wir dies im nächsten Schritt berücksichtigen:

```
  2     -    Borger   =    1
 -1                      -1
 --                      --
  ?                       0
```

Im zweiten Schritt erhalten wir das Ergebnis 0. Beide Schritte zusammen ergeben also

0 + 8 = 8

Demnach lautet die Lösung der Aufgabe 25 - 17 = 8.

Die gleiche Methode können wir verwenden, wenn wir eine Subtraktion mit Z80-Befehlen durchführen.

Aufgabe: Subtrahiere 2000 von 2224.

Schritt 1

Zunächst werden die beiden Zahlen in Hexadezimalzahlen umgewandelt.

```
2000 : 4096 = 0   Rest   2000
2000 :  256 = 7   Rest    208
 208 :   16 = D   Rest      0
   0 :    1 = 0   Rest      0

2224 : 4096 = 0   Rest   2224
2224 :  256 = 8   Rest    176
 176 :   16 = B   Rest      0
   0 :    1 = 0   Rest      0
```

Somit gilt

2000 = &07D0

und

2224 = &08B0

Schritt 2

Zunächst werden die beiden niederwertigen Bytes subtrahiert

```
  B0     +    Borger   =   1B0
 -D0                      -D0
 ---                      ---
  ??                       E0
```

0 = 0 - 0
E =1B - D

Schritt 3

Jetzt kommen die höherwertigen Bytes dran:

```
 08                       08
 07                      -07  +  Borger = -08
 --                       --
 ??                       00
```

Stellengerecht zusammengefügt, folgt somit:

2224 - 2000 = &00E0

Bei dieser Art der Berechnung verhält sich der Übertrag wie ein Borger. D.h das entsprechende Statusbit wird dann gesetzt, wenn ein Borger notwendig wird. Nun wollen wir unsere frisch erworbenen Kenntnisse in ein Programm umsetzen.

Programm 4.5

```
ENT
LD    C,&D0      C enthält das 1. niederwertige Byte
LD    A,&B0      A enthält das 2. niederwertige Byte
SUB   C          A enthält nun das Ergebnis des 1. Zwischenschritts
LD    (&7000),A  Niederwertiges Byte abspeichern
LD    C,&07      C enthält das 1. höherwertige Byte
LD    A,&08      A enthält das 2. höherwertige Byte
```

```
SBC   A,C        A = A - Übertrag
LD    (&7001),A  Ergebnis abspeichern
LD    A,(&7000)  Höherwertiges Byte abrufen und
CALL  &BB5A      ausgeben
RET
```

Anmerkung: Im vorliegenden Fall wissen wir, daß das höherwertige Byte des Ergebnisses 0 ist. Wir brauchen es also auch nicht auszugeben. Außerdem brauchen wir keinen Offset vorzusehen, weil der Wert des niederwertigen Bytes innerhalb des Bereichs von 65 bis 255 liegt.

Übungsaufgabe 4.6

Schreiben Sie ein Programm, das eine 16-bit-Subtraktion (4248 - 4008 = ?) durchführt. Verwenden Sie für diesen Zweck die nachfolgend angegebenen Befehle (ss ist ein Registerpaar, im vorliegenden Fall BC oder DE).

```
SBC HL,ss
```

(Hinweis: Es ist etwas einfacher als das Programm 4.5.)

Eine Antwort finden Sie im Anhang G.

Übungsaufgabe 4.7

Speichern Sie zwei Zahlen im Speicher ab, und addieren Sie anschließend zu beiden den Inhalt des Akkumulators. Ersetzen Sie dann die alten Werte durch die neuen.

Die folgenden Tabellen sollen Ihnen bei der Lösung helfen.

Speicher-adresse	Inhalt
35000	10
35001	20

Inhalt des Akkumulators = 65

Nach dem Programmlauf:

Speicher-adresse	Inhalt
35000	75
35001	85

Verwenden Sie den Befehl ADD A,(HL). Überprüfen Sie, ob die Ergebnisse mit den zu erwartenden übereinstimmen, indem Sie die Inhalte der beiden Speicherzellen auf dem Bildschirm zur Kontrolle ausgeben.

Eine Lösung finden Sie im Anhang G.

Übungsaufgabe 4.8

Zeichnen Sie unter Verwendung der entsprechenden ROM- Routine eine gerade Linie zum Punkt mit den Bildschirmkoordinaten 100,50. Addieren Sie anschließend zu beiden Koordinatenwerten die Zahl 75, und verbinden Sie die Punkte miteinander.

Die Routine zur Ausgabe einer Geraden auf dem Bildschirm ist vollständig, aber kurz im Anhang dokumentiert:

Die Startadresse lautet 48118. Als Parameter müssen die Koordinaten X an das Registerpaar DE und Y an das Registerpaar HL übergeben werden.

Eine Lösung können Sie im Anhang G nachschlagen.

Hiermit haben wir die Grundlagen der Addition und der Subtraktion abgeschlossen. Die nachfolgend angegebenen Befehle wurden zwar bisher nicht im einzelnen vorgestellt. Sie sollten aber selbsterläuternd sein

Befehl	Symb. Operation
ADD HL,ss	HL ← HL + ss
ADC HL,ss	HL ← HL + ss + Übertrag
SBC HL,ss	HL ← HL - ss - Borger
ADD IX,pp	IX ← IX +pp
ADD IY,rr	IY ← IY + rr

Hierbei ist

ss	eines der Register BC, DE, HL oder SP
pp	eines der Register BC, DE, IX oder SP
rr	eines der Register BC, DE, IY oder SP

Bei einem Blick auf die symbolische Operation sollten Sie sofort verstehen, wie die einzelnen Befehle sich verhalten. Mit den zuvor genannten Befehlen kann man sogar 32-bit-Zahlen unter Ausnutzung des Übertragsbits addieren. Die Methode entspricht der, die wir bereits bei 16-bit-Operationen unter Anwendung von 8-bit-Befehlen kennengelernt haben.

Befehle zur Manipulation des Übertragsbits.

Vor der Durchführung arithmetischer Operationen muß häufig das Übertragsbit zurückgesetzt werden. Wir haben dies dadurch erreicht, daß wir von dem Befehl AND A Gebrauch gemacht haben. Die folgenden Befehle wirken ebenfalls unmittelbar auf dieses Statusbit:

SCF Setze das Übertragsbit auf 1.

CCF Kehre den logischen Zustand des Übertragsbits um.

Um mit diesen beiden Befehlen das Übertragsbit sicher zurückzusetzen, muß es erst mit dem erstgenannten Befehl gesetzt und dann mit dem zweiten durch Umkehren des logischen Zustands zurückgesetzt werden. Bei der Bildung des Komplementes einer Dualzahl wird jede 1 in eine 0 und jede 0 in eine 1 umgewandelt. Wir haben bisher deshalb den Befehl AND A verwendet, weil er denselben Effekt in der halben Zeit erzielt wie die beiden Instruktionen SCF und CCF zusammen.

Zusammenfassung

Sie sollten nach den bisherigen Ausführungen die Grundlagen der Addition und der Subtraktion von 8- und 16-bit-Zahlen einschließlich der Anwendung des Übertragsbits verstanden haben.

Folgende Begriffe sollten Ihnen vertraut sein:

ADD A,n
ADD A,s
SUB A,s
SBC A,s
Borger
Übertrag
SBC HL,ss
ADD HL,ss
ADC HL,ss
ADD IX,pp
ADD IY,rr

Kapitel 5

BCD-Zahlen und logische Operatoren

Neben den in dualer, hexadezimaler und dezimaler Schreibweise angegebenen Zahlen ist in der Computertechnik noch eine weitere Art der Zahlendarstellung bekannt. Sie wird als BCD-Darstellung bezeichnet. BCD ist eine Abkürzung für Binary Coded Decimal. Während in der dualen Schreibweise eine Zahl im Bereich zwischen 0 und 255 normalerweise in einem Byte abgespeichert wird, wird in der BCD-Notation ein Byte in zwei Datenworte aufgeteilt, die dann als Nibble bezeichnet werden.

So erfordert beispielsweise die Zahl 7564 in der BCD-Darstellung 2 Bytes (4 Nibbles). Das sieht dann so aus:

	7	5
Byte 1	0111	0101
	6	4
Byte 2	0110	0100

Bitte merken Sie sich, daß das Byte 1 nicht unbedingt als erstes im Speicher abgelegt werden muß. Die Speicherfolge hängt von dem im Programm vereinbarten Format ab.

In der BCD-Schreibweise wird somit jede Ziffer einer Dezimalzahl im Bereich von 0 bis 9 einzeln mit 4 bit verschlüsselt. Von den insgesamt 16 verschiedenen Nibbles zwischen 0000 und 1111 werden demnach nur die ersten 10 (bis 1001) benötigt. Angewendet wird diese Art der Zahlenverschlüsselung im Bereich des Finanz- und Geschäftswesens oder auch bei der Übertragung von Werten an spezielle Anzeigeeinheiten (beispielsweise an sogenannte 7-Segment-Anzeigen in Taschenrechnern oder bei Digitaluhren).

Wir wollen uns nun ansehen, wie wir BCD-Zahlen addieren können.

Beispiel:

Dezimal	BCD
8	1000
+2	0010
10	1010

Das sieht zwar recht vernünftig aus, denn 8 + 2 ergibt ja dezimal 10. Dies entspricht einer dualen 1010. Die Vorschrift für die Bildung von BCD-Zahlen besagt jedoch, daß immer nur eine der Ziffern 0 bis 9 in einem Nibble verschlüsselt werden darf. Die oberen 6 Nibbles von 1010 bis 1111 dürfen daher für eine Zahlenverschlüsselung nicht benutzt werden. Wir benötigen also eine Art Korrektur, die es verhindert, daß die nicht erlaubten Codeworte (Pseudotetraden genannt) bei BCD-Rechenoperationen auftreten. Diese Korrektur besteht darin, daß wir bei der Überschreitung des zulässigen Wertebereichs (maximal die Ziffer 9) einfach die Zahl 6 zum Ergebnis hinzuaddieren.

Das sieht dann so aus:

1. Nibble	2. Nibble	
0000	1010	
+0000	+0110	(6 dezimal)
0001	0000	

Jetzt stimmt das Ergebnis. Es lautet 0001 0000, also dezimal 10.

In der Praxis wäre es sehr lästig, wenn wir nach jeder Berechnung mit BCD-Zahlen die Gültigkeit des Ergebnisses überprüfen und es anschließend gegebenenfalls korrigieren müßten. Der Z80 besitzt zu diesem Zweck glücklicherweise einen Befehl, der das alles für uns automatisch erledigt. Er lautet:

DAA Führe eine BCD-Korrektur durch.

Nach diesen einleitenden Bemerkungen zum Thema wollen wir uns jetzt weiter um die Maschinenprogrammierung kümmern.

Das folgende Programm 5.1 addiert die Zahlen 8 und 2 und gibt das Ergebnis als BCD-Zahl aus.

Programm 5.1

```
ENT
LD    A,8
ADD   A,2
DAA
ADD   A,65
CALL  47962
RET
```


Achten Sie hier wieder auf den Offset (65), der das Ergebnis in einen darstellbaren Bereich verlagert.

Bevor dieser Offset addiert wird, lautet das Ergebnis 0001 0000 (BCD-Zahl für 10). Dies entspricht bei dualer Interpretation einer dezimalen 16. Das auf dem Bildschirm ausgegebene Zeichen ist somit der Buchstabe Q (ASCII-Code 81 wegen 16 + 65 = 81).

Übungsaufgabe 5.1

Addieren Sie die Zahlen 7 und 5 im BCD-Modus, und stellen Sie das Ergebnis als Buchstabe auf dem Bildschirm dar.

Übungsaufgabe 5.2

Subtrahieren Sie die Zahl &12 von &35, wandeln Sie das Ergebnis in eine BCD-Zahl um, und stellen Sie es auf dem Bildschirm in Form eines ASCII-Zeichens dar.

Sie finden die Lösungen im Anhang G.

Manchmal ist es notwendig , das niederwertige Nibble eines Bytes zu isolieren. Eine Möglichkeit besteht darin, die höchstwertigen vier Bits auszublenden. Hierfür können Sie den Befehl

AND s Logische UND-Verknüpfung mit dem Operanden s.

verwenden.

Für s können A, B, C, D, E, H, L,(HL), (IX+d) oder (IY+d) eingesetzt werden.

Jeder, der schon einmal etwas mit digitaler Elektronik zu tun hatte, wird das nachfolgend gezeigte Symbol kennen. Es ist das Symbol für ein UND-Gatter.

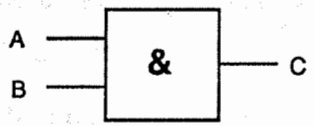


Abb. 5.1: Symbol eines UND-Gatters

So ein UND-Gatter arbeitet folgendermaßen: Der Ausgang C wird dann und nur dann gesetzt (1), wenn beide Eingänge A und B gesetzt, also logisch 1 sind. Die Beschreibung des Verhaltens eines UND-Gatters ist recht einfach. Je komplizierter jedoch das logische Verhalten eines Gatters wird, desto schwieriger ist es, sein Verhalten zu erläutern. Am besten benutzt man eine sogenannte Wahrheitstabelle. Sie ermöglicht es uns, das Verhalten des Ausgangs in Abhängigkeit von den logischen Eingangsgrößen leicht zu durchschauen.

Die Wahrheitstabelle eines UND-Gatters zeigt Abb. 5.2:

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Abb. 5.2: Wahrheitstabelle für eine UND-Verküpfung

Wie Sie sehen, ist C nur dann logisch 1, wenn die Eingänge A und B gesetzt sind. Daher der Name UND.

Übungsaufgabe 5.3

Entscheiden Sie anhand der Tabelle in Abb. 5.2, welchen logischen Zustand C in den folgenden Fällen annimmt

1. A=0 B=1
2. A=1 B=1
3. A=0 B=0

Sie finden die Lösungen in Anhang G.

Wenn der Z80-Prozessor einen UND-Befehl ausführt, führt er die Verknüpfung Bit für Bit durch. Schauen Sie sich einmal das folgende Beispiel an:

Aufgabe: Wie lautet das Ergebnis der UND-Verknüpfung von 10101101 mit 00001111?

Die Lösung lautet:


```
  10101101
UND 00001111
    --------
  00001101
```

Was ist passiert?

Das höherwertige Nibble ist offensichtlich mit Hilfe der Maske 00001111 ausgeblendet worden, d.h. daß die vier höchstwertige Bits unabhängig von ihrem logischen Zustand auf 0 gesetzt wurden. Die niederwertigen vier Bits dagegen bleiben ihrem Wert nach erhalten.

Dieser Trick ist höchst wertvoll, weil er uns die Möglichkeit eröffnet, jeden Teil eines Bytes durch eine entsprechende Maske zu isolieren. Wenn wir beispielsweise das niederwertige Nibble ausblenden wollen, brauchen wir nur die Maske 11110000 zu verwenden:

```
  1010 1101
UND 1111 0000
    ---------
  1010 0000
```

Allgemein gesagt muß nur jede Bitposition, die nicht beeinflußt werden soll, über eine logische 1 verknüpft werden, die übrigen dagegen über eine logische 0. Wenn beispielsweise das Ergebnis der UND-Verknüpfung einer 8-bit-Dualzahl mit der Maske 00000011 eine 2 liefert, dann wissen wir, daß die Zahl durch 2 dividierbar ist - gleichgültig, welchen Wert sie aktuell hat.

Übungsaufgabe 5.4

Welche Maske muß benutzt werden, um aus dem Datenwort 10101011 das Ergebnis 0000011 zu erzeugen?

Übungsaufgabe 5.5

Wie sieht das Ergebnis einer UND-Verknüpfung der Zahlen 253 und 75 aus?

Die Antworten finden Sie im Anhang G.

Jetzt wollen wir den UND-Befehl in einem Programm einsetzen. Programm 5.2 verknüpft 255 mit 224 über eine UND-Operation und gibt das Ergebnis auf dem Bildschirm aus.

Programm 5.2

```
LD    C,255
LD    A,224
AND   C
CALL  47962
RET
```

Als Resultat sehen Sie auf dem Bildschirm ein kleines "Smiley", da das Ergebnis zum ASCII-Code 224 führt.

Die UND-Verknüpfung ist nicht die einzige logische Operation, die der Z80 unterstützt. Wir wollen uns nun noch das ODER-Gatter ansehen. Dessen Standardsymbol nach der hierzulande üblichen DIN-Norm zeigt Abb 5.3:

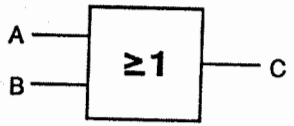


Abb. 5.3: Das Symbol für ein ODER-Gatter

Der Ausgangs C ist immer dann logisch 1, wenn A oder B oder beide Eingänge gleich 1 sind. Dies wird durch die nachfolgend gezeigte Wahrheitstabelle zum Ausdruck gebracht.

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

Abb. 5.4: Wahrheitstabelle für die ODER-Verknüpfung

Der entsprechende ODER-Befehl des Z80 lautet allgemein:

OR s Verknüpfe den Inhalt des Akkumulators mit dem des Operanden mit ODER.

Hier ist s wieder A, B, C, D, E, H, L, (HL), (IX+d) oder (IY+d).

Übungsaufgabe 5.6

Wie lautet das Ergebnis der folgenden logischen Operationen?

1. 1001 ODER 1101
2. 250 ODER 25
3. (209 ODER 20) UND 27

Antworten finden Sie wieder im Anhang G.

Es mag seltsam erscheinen, daß ein ODER-Gatter ein Ausgangssignal 1 erzeugt, wenn beide Eingänge logisch 1 sind. In der Tat kann dies in speziell gelagerten Fällen zu Problemen führen. Das sogenannte Exklusiv-ODER-Gatter hilft in diesem Fall weiter. Für die damit verbundene Verknüpfungsform ist der Name XOR gebräuchlich. Das Symbol für ein entsprechendes Gatter zeigt Abb. 5.5:

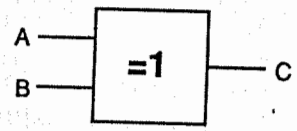


Abb. 5.5: Das Exklusiv-ODER-Gatter

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Abb. 5.6: Die Wahrheitstabelle der Exklusiv-ODER-Verknüpfung

Der entsprechende Z80-Befehl lautet:

XOR s Führe eine Exklusiv-ODER-Verknüpfung zwischen dem Inhalt des Akkumulators und dem des Operanden s durch.

Auch hier ist s wieder A, B, C, D, E, H, L, (HL), (IX+d) oder (IY+d).

Übungsaufgabe 5.7

Wie sehen die Ergebnisse der nachfolgend angegebenen logischen Verknüpfungen aus? Probieren Sie sie erst auf dem Papier aus. Schreiben Sie dann ein Programm, mit dessen Hilfe Sie Ihre Lösungen überprüfen können.

1. 1011 XOR 1110100
2. 77 XOR 200
3. (25 ODER 255) UND 200

Antworten finden Sie im Anhang G.

Vorzeichenbehaftete Zahlen

Bisher waren alle Zahlenwerte, mit denen wir gearbeitet haben, positiv. In vielen Fällen benötigen wir jedoch auch negative Zahlen.

Wie können wir negative Dualzahlen darstellen? Die im Nachhinein vorgestellte Methode der Zahlendarstellung sorgt dafür, daß der Z80 negative Zahlen in genau derselben Art und Weise verarbeiten kann wie positive. Sie ist mit dem Begriff "Zweierkomplement" verknüpft. Bevor wir jedoch die Zweierkomplementdarstellung von Zahlen einführen, sollten wir zunächst die Grundlagen der Einerkomplementbildung kennen- und verstehen lernen.

Das Einerkomplement

In der Einerkomplement-Notation von Zahlen werden positive ganze Zahlen grundsätzlich genauso dargestellt wie bei der üblichen dualen Notation. Das Einerkomplement zur Darstellung negativer Zahlen wird dadurch gebildet, daß die logischen Werte der einzelnen Bits einfach vertauscht werden. Aus einer Null wird eine Eins und umgekehrt.

Die Ausweitung des Wertebereichs auf negative Zahlen bedingt allerdings, daß wir bei gleichem absolutem Werteumfang ein Bit mehr spendieren oder bei gleicher Wortlänge den Wertebereich verringern müssen.

Beispiel: Die Zahl +9 lautet in der normalen dualen Schreibweise 01001. Deren Einerkomplement lautet einfach 10110. Das heißt, daß die Zahl -9 der 10110 entspricht. Bitte beachten Sie, daß wegen der Erweiterung des Wertebereichs (+9 bis -9) ein Bit mehr benötigt wird. Negative Zahlen sind somit eindeutig dadurch gekennzeichnet, daß das höchstwertige Bit logisch 1 ist. Bei dieser Art der Zahlendarstellung tritt das Problem auf, daß für die Zahl 0 zwei verschiedene Verschlüsselungen auftreten (+0 = 00000, -0 = 11111). Außerdem ergeben sich bei normalen arithmetischen Operationen einige Probleme. Dies sind die Hauptgründe dafür, daß dem Einerkomplement das Zweierkomplement zur Darstellung negativer Zahlen vorgezogen wird. Es läßt sich auf einfache Weise aus dem Einerkomplement bilden.

Übungsaufgabe 5.8

Bilden Sie von den angegebenen Zahlen das Einerkomplement.

1. 1011
2. 1011101
3. 14

Antworten finden Sie im Anhang G.

Das Zweierkomplement

Wie beim Einerkomplement werden auch beim Zweierkomplement die positiven ganzen Zahlen in der üblichen dualen Notation angegeben. Negative Zahlen werden dadurch gebildet, daß nach der Ermittlung des Einerkomplements noch die Zahl 1 addiert wird. Auch hier werden negative Zahlen durch den Wert 1 des höchstwertigen Bits gekennzeichnet. Denken Sie daran, daß ein Bit mehr benötigt wird als bei rein positiver Darstellung des absoluten Zahlenbereichs.

Beispiel: Für die Zahl +5 reichen in der üblichen dualen Notation drei Bits: +5 = 101, denn mit drei Bits können die Ziffern 0 bis 7 dargestellt werden. Werden auch negative Zahlen zugelassen, lautet die duale Verschlüsselung +5 = 0101.

Wir wollen uns die Bildung des Zweierkomplements anhand eines Beispiels einmal ansehen und anschließend mit Zweierkomplementzahlen die Aufgabe lösen, die Zahlen 7 und -5 zu addieren.

5	=	0101
Einerkomplement		1010
1 hinzuaddieren	+	0001
		1011 = -5

Die Zahl -5 lautet in Zweierkomplementform also 1011.

Jetzt folgt die Lösung der Rechenaufgabe:

7	0111
+ (-5)	1011
2	0010 + Übertrag

Wenn wir das Übertragsbit einfach vernachlässigen, ist das Ergebnis unserer Berechnung offensichtlich richtig. Der Umstand, daß wir unter Vernachlässigung des Übertragsbits bei der Verwendung von Zahlen in der Zweierkomplementform zu richtigen Ergebnissen gelangen, trägt wesentlich zur Vereinfachung arithmetischer Operationen mit negativen Zahlen bei.

Übungsaufgabe 5.9

Berechnen Sie die Ergebnisse der nachfolgenden Aufgaben. Stellen Sie negative Zahlen in der Zweierkomplementform dar.

1. -3 + 10 = ?
2. -1 + 7 = ?
3. -10 + 8 = ?

Sie finden die Lösungen im Anhang G.

Mit vier Bits können nach dem zuvor Gesagten die folgenden positiven und negativen Zahlenwerte in Zweierkomplementform dargestellt werden:

dezimal	dual
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Die sechzehn möglichen Kombinationen 0000 bis 1111 geben also nicht mehr einfach die positiven Zahlenwerte 0 bis 15, sondern die ganzen Zahlen im Bereich von -8 bis +7 wider. Genauso verhält es sich bei 8 Bits, mit denen in diesem Fall die Zahlen von -128 bis +127 widergegeben werden können. Probleme treten dann auf, wenn die Addition von Zahlen zu einem Ergebnis führt, das den Wert +127 überschreitet.

Beispiel:

100	01100100
+79	01001111
179	10101011

In der Zweierkomplementform entspricht 10101011 der Zahl -85. Das ist ganz offensichtlich ein falsches Ergebnis. Es ist ein Überlauf aufgetreten. Er wird durch das P/V-Bit (Überlauf-Bit) angezeigt. Es ist Aufgabe des Programmierers, auf diesen Fall in geeigneter Form zu reagieren. Im einfachsten Fall wird das Ergebnis einfach als ungültig gekennzeichnet. Ein Überlauf kann auftreten, wenn

1. zwei große positive oder negative Zahlen addiert werden;
2. eine große positive Zahl von einer großen negativen Zahl oder
3. eine große negative Zahl von einer großen positiven Zahl abgezogen wird.

Der Z80 kennt Befehle, die den Akkumulatorinhalt entweder in die Einer- oder in die Zweierkomplementform überführen:

NEG Bildet das Zweierkomplement des Akkumulatorinhalts.

CPL Bildet das Einerkomplement des Akkumulatorinhalts.

Das folgende Programm 5.3 berechnet das Ergebnis der Summe aus den Zahlen -112 und +104:

Programm 5.3

```
ENT
LD    A,112       A=112
NEG               A=-112
ADD   A,104       A=-112 + 104
CALL  &BB5A
RET
```

Der Programmlauf führt zur Ausgabe eines Grafikzeichens auf dem Bildschirm (ASCII-Code 248 beim Schneider CPC). In dualer Form lautet die Zahl 248

11111000. In der Zweierkomplementform entspricht dies der -8. Sie sehen, daß der Z80 offensichtlich nicht zwischen 248 und -8 unterscheiden kann. Die Zweierkomplementdarstellung negativer Zahlen dient dem Programmierer, um negative Zahlen darzustellen. Sie wird in dieser Form vom Z80 nicht verwendet. Achten Sie bitte darauf, damit Sie bei Ihren Berechnungen auch zu den richtigen Ergebnissen gelangen.

Übungsaufgabe 5.10

Verwenden Sie bitte anstelle des NEG-Befehls die Befehle CPL und INC, um das Ergebnis der nachfolgenden Summe zu berechnen:

-20 + 98 = ?

Eine mögliche Lösung finden Sie im Anhang G.

Zusammenfassung

Die folgenden Begriffe sollten Ihnen nunmehr vertraut sein:

1. BCD-Arithmetik

2. Die logischen Operatoren

   a) UND
   b) ODER
   c) Exklusiv-ODER (XOR)
   d) logische Masken

3. Vorzeichenbehaftete Zahlen

   a) Einerkomplement
   b) Zweierkomplement
   c) Überlaufbit

Auch die folgenden Befehle sollten Sie kennen:

DAA	AND s
CPL	OR s
NEG	XOR s

Kapitel 6

Multiplikation, Division und die Verschiebebefehle

Praktisch alle Computerprogramme bauen auf der Verarbeitung von Zahlenwerten auf. Einige mögen dabei durchaus mit einfachen Additionen oder Subtraktionen auskommen. Die weitaus meisten jedoch arbeiten mit Multiplikations- und Divisionsoperationen. Wir wollen uns in diesem Kapitel ansehen, wir wir diese Art von arithmetischen Operationen auf der Maschinenebene einsetzen können.

Die binäre Multiplikation

Bevor wir uns an die binäre Multiplikation begeben, wollen wir uns die Multiplikation von Dezimalzahlen nochmals vor Augen führen. In dem Produkt 13 *14 ist 13 der Multiplikand und 14 der Multiplikator. Die Multiplikation wird wie folgt ausgeführt:

13	Multiplikand
14	Multiplikator
52	
130	
182	Ergebnis

Zunächst wird der Multiplikand mit der niederwertigsten Ziffer des Multiplikators multipliziert und das Zwischenergebnis abgespeichert, d.h.: 13*4=52. Anschließend multiplizieren wir den Multiplikanden mit der nächsten Ziffer des Multiplikators. Das daraus folgende zweite Teilprodukt wird um eine Stelle nach links geschoben. Danach wird stellengerecht addiert. Das Ergebnis lautet korrekt 182.

Wir können dieselbe Methode bei der binären Multiplikation anwenden.

Die Aufgabe 5*7 läßt sich dann so lösen:

```
5 = 0101
7 = 0111

  0111 = (7)
* 0101 = (5)
```

1. Teilprodukt	0111	0111
2. Teilprodukt	00000	0111
3. Teilprodukt	011100	100011
4. Teilprodukt	0000000	100011

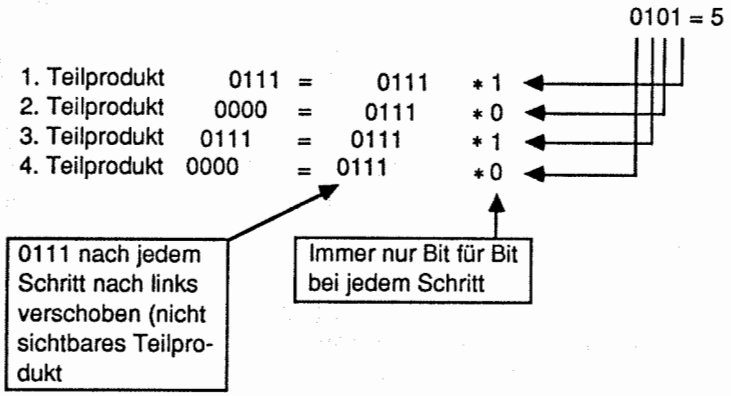
Das Ergebnis lautet 100011.

100011 = (1*32)+(0*16)+(0*8)+(0*4)+(1*12) = 35

Bei Multiplikation mit 1 entspricht jedes Teilprodukt dem Multiplikanden, im anderen Fall ist das Teilprodukt null (0000). Jedes neue Teilprodukt wird um eine Bitposition nach links verschoben. Die binäre Multiplikation läßt sich somit auf eine Folge von Schiebeoperationen und Additionen zurückführen.

Die 8-bit-Multiplikation

Für die Durchführung einer Multiplikation mit dem Z80 werden wir den Akkumulator zur fortlaufenden Abspeicherung der Zwischensumme, das Register C zur Aufnahme des Multiplikanden und das Register E zur Abspeicherung des Multiplikators verwenden. Schauen wir uns nun einmal an, wie wir die Teilprodukte bilden.

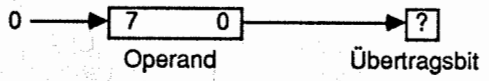


In Form eines Flußdiagramms können wir die binäre Multiplikation in Abb. 6.1 gezeigt ausdrücken.

Zur Entwicklung des entsprechenden Programms benötigen wir nur noch die Befehle, die Bitverschiebungen vornehmen. Der einfachste dieser Befehle lautet:

SRL s Schiebe die Bits des Operanden s nach rechts.

In schematischer Form kann dieser Vorgang so dargestellt werden:



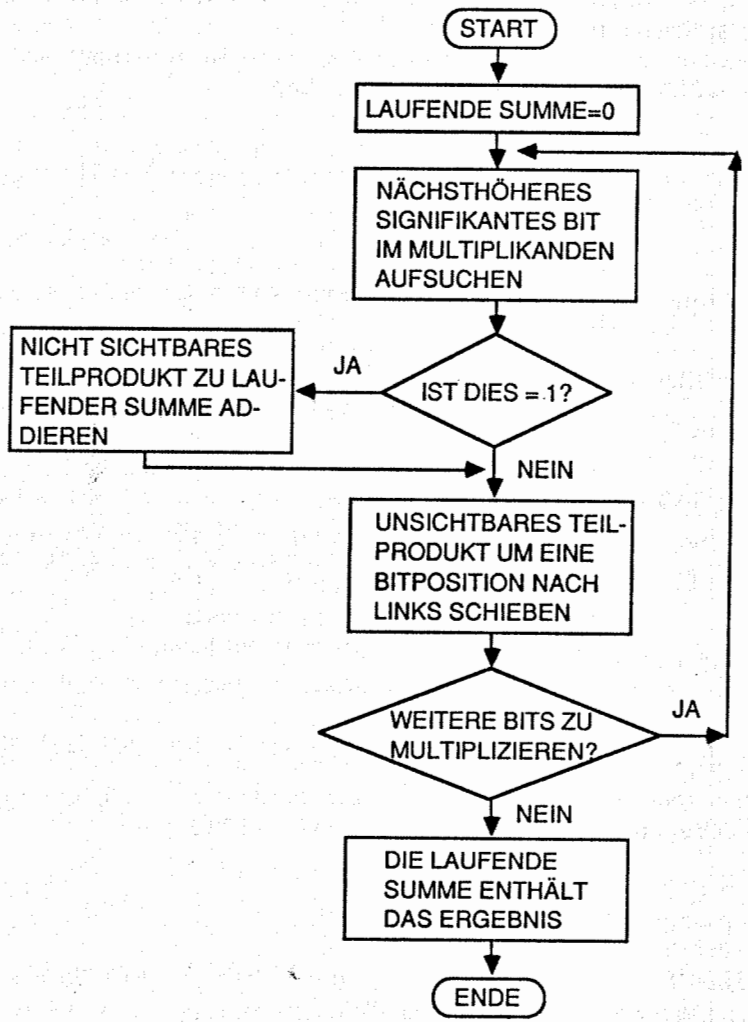


Abb. 6.1: Das Flußdiagramm zur binären Multiplikation

Die Wirkung des Befehls SRL auf das Byte 10110111 sieht wie folgt aus:

	Byte	Übertragsbit
Vorher	10110111	?
Nachher	01011011	1

Bit 7 (das höchstwertige Bit = MSB) wurde durch eine 0 ersetzt, die Bits 1 bis 6 wurden um eine Stelle nach rechts verschoben. Der Inhalt von Bit 0 wurde ins Übertragsbit übertragen.

Das 8-bit-Multiplikationsprogramm lautet wie folgt:

Programm 6.1

```
        ENT
        LD    A,0          Akkumulator nullsetzen
        LD    C,7          C = Multiplikand
        LD    E,5          E = Multiplikator
        LD    B,4          B = Zählregister
ADD:    SRL   C            C nach rechts schieben
        JR    NC,NOADD:    Wenn das Übertragsbit gleich 0, dann
                           springe zu NOADD:
        ADD   A,E          Bilde Zwischensumme in A
NOADD:  SLA   E            Zwischenprodukt nach links schieben
        DEC   B            Zähler erniedrigen
        JR    NZ,ADD:      Zurück zu ADD:, wenn Zähler noch
                           nicht 0
        ADD   A,65         Offset hinzuaddieren
        CALL  &BB5A        Zeichen ausgeben
        RET
```

Bei dem verwendeten Offset wird das Zeichen mit dem ASCII-Code 100 ausgegeben. Dies ist ein kleines d. (Die Antwort lautete ja 7*5 = 35, 35 + 65 = 100.) Ohne Offset wäre ein # ausgegeben worden (# = ASCII 35).

Übungsaufgabe 6.1

Schreiben Sie ein Programm zur Bildung des Produkts 10*9. Stellen Sie das Ergebnis in gewohnter Weise auf dem Bildschirm dar.

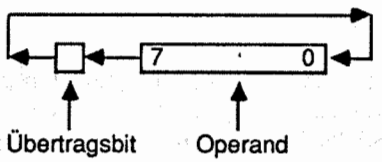
Eine mögliche Lösung finden Sie im Anhang G.

Das Programm 6.1 kann ausschließlich Zahlen miteinander multiplizieren, deren Produkt den Wert 255 nicht überschreitet. Die Hauptursache hierfür ist darin zu suchen, daß das "unsichtbare" Zwischenprodukt unter Umständen völlig aus dem Register geschoben wird. Auf ein ähnliches Problem waren wir bereits im Zusammenhang mit der Addition und der Subtraktion von 16-bit-Zahlen gestoßen. Wir hatten es damals dadurch gelöst, daß wir zum höherwertigen Teil des verwendeten Registerpaares den Übertrag aus der vorangegangenen Addition der beiden niederwertigen Bytes addiert haben. Wir benötigen zur Lösung des anstehenden Problems einen Befehl, der nach der Schiebeoperation beim niederwertigen Byte eine Schiebeoperation einschließlich eines eventuell auftretenden Übertrags beim höherwertigen Byte durchführt. Hierfür steht uns der Befehl

RL s Führe eine zyklische Schiebeoperation mit dem Operanden unter Einbeziehung des Übertrags durch.

zur Verfügung.

In Form eines Schemas sieht läßt sich die Wirkung dieses Befehls so erläutern:



Das Übertragsbit wird bei dieser Art der Verschiebung nach Bit 0 geladen, Bit 0 bis Bit 7 werden um eine Position nach links geschoben, und das Bit 7 wird in das Übertragsbit abgebildet.

Um eine Schiebeoperation auf ein 16-bit-Register auszuüben, werden zwei sich ergänzende Befehle benötigt. Das niederwertige Byte wird normalerweise mit dem SLA-Befehl manipuliert. Danach wird der Inhalt des höherwertigen Bytes mittels des RL-Befehls verschoben. Wenn wir beispielsweise eine Schiebeoperation auf das Registerpaar DE ausüben wollen, erreichen wir dies durch die Befehlsfolge

```
SLA E
RL D
```

Natürlich wollen wir auch diese neuen Befehle in einem Programm einsetzen.

Aufgabe: Geben Sie das Ergebnis des Produktes 7*10 auf dem Bildschirm aus.

Programm 6.2

```
        ENT
        LD      C,7         C = Multiplikand
        LD      E,10        E = Multiplikator
        LD      D,0         Register D löschen
        LD      B,8         B = Anzahl der Bits
        LD      HL,0        Registerpaar HL löschen. Es wird zur
                            Aufnahme des Ergebnisses verwendet.
NXTB:   SRL     C           Multiplikand nach rechts schieben
        JR      NC,NOADD:   Springe zu NOADD, wenn Übertrag 0
        ADD     HL,DE       Zwischensumme hinzuaddieren
NOADD:  SLA     E           Niederwertigen Teil nach links schieben
        RL      D           Inhalt von D nach links schieben unter
                            Einbeziehung des Übertragsbits
        DEC     B           Zähler erniedrigen
        JR      NZ,NXTB:    Rücksprung, wenn noch nicht fertig
        LD      A,L         A mit Ergebnis laden
        CALL    &BB5A       Ergebnis ausgeben (F)
        RET
```

Bitte beachten Sie, daß wir auf Grund der Auswahl des Multiplikanden und des Multiplikators trotz der Verwendung des 16-bit-Registers zur Darstellung des Ergebnisses nur das niederwertige Register L für den Ausdruck auf dem Bildschirm benötigen.

Übungsaufgabe 6.2

Berechnen Sie mit dem Programm 6.2 das Ergebnis von 146*124. Achtung: Das Ergebnis ist 16 bit lang, d.h. es müssen sowohl H als auch L auf dem Bildschirm ausgegeben werden.

Eine Lösung finden Sie wieder im Anhang G.

Das Programm 6.2 macht von dem Register B als Zählregister Gebrauch. Jedesmal, wenn der Multiplikator nach links geschoben wurde, wurde B um 1 erniedrigt. Diese Operation macht die beiden Befehle DEC und JR NZ erforderlich. Sie können sie durch eine einzige ersetzen. Dadurch wird das Programm kürzer und eleganter.

DJNZ e Erniedrige B und springe zur Adresse e, wenn B verschieden von 0 ist.

Übungsaufgabe 6.3

Ersetzen Sie die Befehle DEC B und JR NZ,NXTB: in Programm 6.2 oder in Ihrer Antwort auf die Übungsaufgabe 6.2 durch den Befehl DJNZ NXTB:.

Sie finden die Antwort im Anhang G.

Die binäre Division

Schauen Sie sich einmal die nachfolgende Divisionsaufgabe an:

785 : 10 = ?

Zunächst werden wir versuchen, 7 durch 10 zu teilen. Da das nicht geht, nehmen wir im nächsten Schritt die 8 hinzu und versuchen 78 durch 10 zu dividieren. Das Ergebnis lautet 7 mit einem Rest von 8.

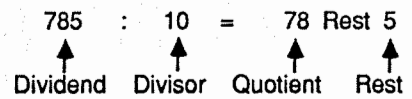
785 : 10 = 7 Rest 8

Im nächsten Schritt nehmen wir die 5 hinzu und versuchen 85 durch 10 zu teilen. Dieses Teilergebnis lautet 8 mit einem Rest von 5.

85 : 10 = 8 Rest 5

Diese Form der Division wird ganzzahlige Division genannt, da gebrochene Zahlen nicht zugelassen werden.

Wie bei der Multiplikation haben auch bei der Division die beteiligten Zahlen Namen. Den Begriff Rest kennen Sie ja schon. Die anderen lauten wie folgt:



Wir wollen uns jetzt einmal die Division einer 16- bit-Zahl durch eine 8-bit-Zahl ansehen.

Aufgabe: Geben Sie das Ergebnis der folgenden Divisionsaufgabe auf dem Bildschirm in lesbarer Form aus:

2765 : 75 = ?

Die binäre Division wird nach folgender Regel durchgeführt: Wenn ohne Auftreten eines Übertrags (Borgers) der 8-bit-Divisor vom höherwertigen Byte

des 16-bit-Dividenden abgezogen werden kann, wird in dem 8-bit-Ergebnisregister das entsprechende Bit gesetzt. Dieser Vorgang wird achtmal wiederholt. Das Ergebnis ist dann ein Quotient und ein Rest von jeweils 8 bit Länge.

Programm 6.3

```
        ENT
        LD      HL,2765   HL = Dividend
        LD      C,75      C = Divisor
        LD      B,8       B = Zählregister
NXT:    ADD     HL,HL     Schiebe den Dividenden nach links
        LD      A,H       A = höherwertiges Byte des Dividenden
        SUB     C         Divisor abziehen
        JR      C,NXTB:   Springe zu NXTB:, wenn ein Übertrag
                          auftritt
        LD      H,A       Höherwertiges Byte des Dividenden retten.
        INC     L         Wert des Ergebnisbytes erhöhen.
NXTB:   DEC     B         Zähler erniedrigen
        JR      NZ,NXT:   Zurück zu NXT:, wenn nicht 0
        LD      A,L       A enthält den Quotienten
        CALL    &BB5A     Quotient ausgeben
        LD      A,H       A = Rest
        CALL    &BB5A     Rest ausgeben
        RET
```

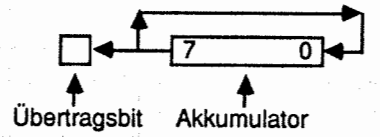
Starten Sie das Programm. Es wird $A auf dem Bildschirm ausgeben. Das Zeichen $ entspricht dem Quotienten und A dem Rest. Überprüfen Sie, daß diese Aussage korrekt ist. Eine Bemerkung ist an dieser Stelle noch notwendig: Weil der Z80 keinen Befehl zur Linksverschiebung eines 16-bit-Registers kennt, wurde im Programm von dem Befehl ADD HL,HL Gebrauch gemacht. Diese Anweisung hat genau denselben Effekt wie eine Verschiebung um eine Position nach links bzw. eine Multiplikation mit 2. Wenn Sie dies mit den Verhältnissen bei Dezimalzahlen vergleichen, werden Sie feststellen, daß dies dann der Multiplikation mit 10 entspricht (beispielsweise 19 führt zu 190). Was, glauben Sie, wird die Wirkung einer Rechtsverschiebung bei einem binären Datenwort sein? (Antwort: Dies entspricht einer Division durch die Zahl 2.)

Bisher haben wir nur einfache Schiebebefehle des Z80 im Zusammenhang mit allgemeinen Routinen zur Multiplikation und zur Division verwendet. Für eine Reihe von Operationen ist es manchmal günstiger, zyklische Verschiebeoperationen zu benutzen. Zwar werden auch durch diese die Registerinhalte verändert, es gehen aber dabei keine Informationen verloren.

Nachfolgend ein Befehl, der den Inhalt des Akkumulators zyklisch nach links verschiebt:

RLCA Verschiebe den Inhalt des Akkumulators nach links und kopiere dabei das Bit 7 in das Übertragsbit.

In Form eines grafischen Schemas sieht das so aus:



Beachten Sie bitte, daß bei diesem Befehlstyp das Bit 7 nicht verloren geht. Es wird vielmehr nach Bit 0 verschoben. Zur zyklischen Rechtsverschiebung steht ein ähnlicher Befehl zur Verfügung. Er lautet:

RRCA Verschiebe den Inhalt des Akkumulators nach rechts und kopiere dabei das Bit 0 in das Übertragsbit.

Wir wollen diese beiden Befehle nachfolgend einmal anwenden.

Problem: Multipliziere 10 mit 16, gib das Ergebnis auf dem Bildschirm aus, dividiere das Ergebnis durch 2 und stelle auch dieses Ergebnis auf dem Bildschirm dar.

Programm 6.5

```
ENT
LD      A,10      A=10
RLCA              A=20
RLCA              A=40
RLCA              A=80
RLCA              A=160
CALL    &BB5A     Gib A aus (^)
RRCA              A=80
CALL    &BB5A     Gib A aus (P)
RET
```

Übungsaufgabe 6.4

Schreiben Sie ein Programm zur Berechnung und Ausgabe der folgenden Aufgaben. Verwenden Sie hierzu Additions- und zyklische Verschiebebefehle.

1. 5*32 = (160)
2. 254 : 2 = (127)

Mögliche Antworten finden Sie im Anhang G.

Der Z80 unterstützt noch weitere einfache und zyklische Verschiebebefehle. Sie finden diese im Anhang A.

Abschließend wollen wir uns noch Befehle ansehen, mit deren Hilfe einzelne Bits manipuliert und abgefragt werden können.

Bitmanipulations- und Bittest-Befehle

Schauen Sie sich einmal den nachfolgend gezeigten Befehl an:

BIT b,r Ermittle den logischen Wert des Bits an der Position b des Operanden s.

Als Operand r kommen A, B, C, D, E, H, L, (IX+d), (IY+d) und (HL) in Frage.

Dieser Befehl wird immer dann benutzt, wenn ein ganz spezielles Bit innerhalb eines Bytes auf seinen Wert hin abgefragt werden soll. Vornehmlich ist dies bei Ein-/Ausgabeoperationen der Fall.

Um die Wirkung dieses Befehls zu demonstrieren, wollen wir ein Register auf 0 setzen und dessen Inhalt fortschreitend so lange um 1 erhöhen, bis das Bit 7 gesetzt wird. Anschließend lassen wir uns den Registerinhalt auf dem Bildschirm ausgeben.

Programm 6.6

```
        ENT
        LD      A,0         A = 0
NXT:    INC     A           A = A + 1
        BIT     7,A         Bit 7 abfragen
        JR      Z,NXT:      Wenn Bit 7 = 0, dann gehe zu NXT:
        ADD     A,1         Erzeuge sichtbares Zeichen (mit ASCII-
                            Code 129)
        CALL    &BB5A       Registerinhalt ausgeben
        RET
```


Dieses Programm lädt den Akkumulator zunächst mit dem Wert 0. Anschließend wird A so lange um jeweils 1 erhöht, bis Bit 7 gesetzt ist. Lassen Sie das Programm laufen, und schauen Sie sich dessen Wirkung an. Da A nach Beendigung der Schleife den Wert 128 enthält, müssen wir, um ein sichtbares Zeichen zu erhalten, noch eine 1 addieren. Es erscheint dann ein Grafikzeichen (ein kleines Rechteck).

Es gibt noch zwei weitere Befehle, mit deren Hilfe ein Bit gesetzt bzw. zurückgesetzt werden kann.

SET b,r Setze das Bit b im Operanden r.

RES b,r Setze das Bit b im Operanden r zurück.

Zur Demonstration der Wirkung dieser Befehle schauen Sie sich einmal den ASCII-Code für das Zeichen C an. Er lautet dual 01000011. Einem Zurücksetzen des Bits 0 entspricht einer Subtraktion von 1. Aus dem C wird ein B.

Programm 6.6

```
        ENT
        LD      A,67        A = ASCII-Code von C
        CALL    &BB5A       Gib das Zeichen C aus
        RES     0,A         Setze Bit 0 zurück
        CALL    &BB5A       Gib das Zeichen B aus
        SET     0,A         Setze Bit 0
        CALL    &BB5A       Gib das Zeichen C aus
        RET
```

Übungsaufgabe 6.5

Schreiben Sie ein Programm, das den Akkumulator mit der Zahl 255 lädt, dessen Inhalt auf dem Bildschirm ausgibt, das Bit 4 zurücksetzt und den neuen Inhalt wiederum ausgibt. Setzen Sie anschließend Bit 4 wieder, und setzen Sie Bit 3 zurück. Geben Sie auch dieses Ergebnis auf dem Bildschirm aus.

Sie finden eine mögliche Lösung zu dieser Aufgabe im Anhang G.

Damit sind wir am Ende dieses Kapitels angelangt. Sie sollten nunmehr bereits in der Lage sein, recht komplexe Programme zu entwickeln.

Zusammenfassung

Die folgenden Begriffe und Befehlsgruppen sollten Ihnen jetzt ebenfalls vertraut sein:

1. Binäre Multiplikation und Division
2. Einfache und zyklische Verschiebungen
3. Bitmanipulations- und Bittest-Befehle

Kapitel 7

Der Stapel

Der Stapel ist nichts anderes als ein Speicherblock unmittelbar unterhalb der Adresse &C000. Er wird im Zusammenhang mit schnellen Datenübertragungen innerhalb des Computers verwendet. Gefüllt wird der Datenblock zu absteigenden Adreßwerten. Auf die nächste freie Position wird dabei mittels eines Registers verwiesen, das als Stapelzeiger oder kurz als SP (stack pointer) bezeichnet wird. Als Analogie wird meistens ein Tellerstapel genannt, von dem immer nur der oberste Teller entfernt wird. Das ist also derjenige, der zuletzt auf den Stapel gelegt wurde. Der Stapel des Computers wird abwärts, bei &DFFF beginnend, gefüllt. Diese Methode, einen Stapel zu füllen und wieder zu leeren, wird als LIFO-Prinzip bezeichnet (Last In First Out).

Eine der Aufgaben eines Stapels besteht darin, die Adressen bei Unterprogrammaufrufen zwischenzuspeichern. Dieses tut der Z80 mit Hilfe des Stapels automatisch. Wenn der Z80 auf einen Befehl wie beispielsweise CALL &BBBA trifft, muß er zunächst einmal feststellen, welche Adresse der nächste auszuführende Befehl im Hauptprogramm hat, damit er nach Abarbeitung des Unterprogramms dorthin zurückspringen kann. Erst dann setzt er den Programmzähler (PC) auf die Sprungadresse &BBBA.

Gehen wir einmal davon aus, daß der Z80 den folgenden Befehl auszuführen hat:

```
CALL &BBBA
```

Dann macht der Z80 folgendes:

1. PC erhöhen
2. Das Befehlsbyte abrufen (CALL)
3. Den PC erhöhen
4. 1. Byte des Operanden abrufen (BA)
5. PC erhöhen
6. 2. Byte des Operanden abrufen
7. PC erhöhen
8. Höherwertiges Byte des PC auf den Stapel retten
9. Stapelzeiger um 1 erniedrigen (SP=SP-1)
10. Niederwertiges Byte des PC auf den Stapel retten
11. Stapelzeiger um 1 erniedrigen
12. Adresse &BBBA in den PC laden.
13. Unterprogramm ausführen bis zum RET-Befehl
14. PC erhöhen
15. Befehlsbyte (RET) abrufen.

16. Niederwertiges Byte des PC vom Stapel holen
17. Stapelzeiger um 1 erhöhen
18. Höherwertiges Byte des PC vom Stapel holen
19. Stapelzeiger um 1 erhöhen
20. Im Hauptprogramm nächstes Befehlsbyte abrufen

Natürlich hätte in unserem Beispiel das Unterprogramm noch weitere Unterprogramme aufrufen können. Jedesmal, wenn der Z80 dabei auf einen CALL-Befehl gestoßen wäre, hätte er dieselbe logische Befehlsfolge abgearbeitet. Nach der Abarbeitung der Unterprogramme wären dann die Rücksprungadressen Schritt für Schritt wieder vom Stapel abgerufen worden, so daß der Z80 letztlich wieder an seinem Ausgangspunkt im Hauptprogramm angelangt wäre. Der Speicherumfang des Stapels ist größer als 256 Bytes, so daß Platz genug für derartige Operationen vorhanden ist.

Glücklicherweise brauchen wir uns um die Verwaltung des Stapels im Einzelfall gar nicht zu kümmern, das macht der Z80 als Folge der entsprechenden Befehle alles automatisch. Wenn Sie von den im Festwertspeicher vorhandenen Maschinensprache-Routinen Gebrauch machen, werden die Inhalte der Z80-Register nicht automatisch auf den Stapel gerettet. Dies müssen Sie per Programm schon selbst tun. Hierfür stehen zwei Befehle zur Verfügung.

Push und Pop

PUSH qq Rette das Registerpaar qq auf den Stapel.

Im vorliegenden Fall ist qq eines der Registerpaare AF, BC, DE oder HL. Auch für die Register IX und IY gibt es entsprechende Befehle:

PUSH IX Rette das Register IX auf den Stapel.

PUSH IY Rette das Register IY auf den Stapel.

Diese Befehle kopieren den Inhalt des jeweils angesprochenen Registers auf den Stapel. Um den ursprünglichen Registerinhalt wiederherzustellen, werden POP-Befehle verwendet:

POP qq Lade das Registerpaar qq mit dem obersten Stapelelement.

Auch hier ist qq wieder eines der Registerpaare AF, BC, DE oder HL.

Für die Register IX und IY gilt entsprechend:

POP IX Lade IX mit dem obersten Wert auf dem Stapel.

POP IY Lade IY mit dem obersten Wert auf dem Stapel.

Um den korrekten Inhalt des Stapelzeigers SP brauchen Sie sich nicht zu kümmern. Dies wird vom Z80 automatisch erledigt.

Bei der Benutzung des Stapels müssen Sie immer daran denken, daß dieser sich wie eine LIFO-Struktur verhält. Register werden also im Normalfall immer in exakt der umgekehrten Reihenfolge wieder vom Stapel abgerufen, in der sie zuvor eingeschrieben wurden. Jedem PUSH-Befehl in einem Programm muß somit ein in der Reihenfolge korrespondierender POP-Befehl zugeordnet sein. Andernfalls kommt der Computer durcheinander. Er könnte in einem derartigen Fall beispielsweise den Stapelinhalt als eine Rücksprungadresse auffassen, obgleich es sich in einem aktuellen Fall um einen versehentlich nicht korrekt abgerufenen BC-Registerinhalt handelt. Im schlimmsten Fall kommt es dadurch zu einem Systemabsturz.

Programm 7.1 ist ein gutes Beispiel für den korrekten Gebrauch von PUSH- und POP-Befehlen im Zusammenhang mit der Zwischenspeicherung der Register A und HL. Neben der bereits bekannten ROM-Routine zur Zeichenausgabe mit der Adresse &BB5A wird noch eine weitere verwendet, die unter Adresse &BB75 angesprungen wird. Mit ihrer Hilfe wird der Textcursor auf eine vereinbarte Bildschirmposition gesetzt (siehe Anhang D). Das Register H muß beim Aufruf die Spalten- und das Register L die Zeileninformation enthalten. Wichtig ist, daß die Routine den Inhalt der Register zerstört. Sie sollten also vorher auf den Stapel gerettet werden!

Programm 7.1

```
ENT
LD    HL,&0604    Lade H mit der Spalte 4 und L mit der Zeile 6
LD    A,&4D       Lade A mit dem ASCII-Code für M
PUSH  HL          Rette HL auf den Stapel
PUSH  AF          Rette A (und das Statusregister F) auf den Stapel
CALL  &BB75       Setze den Cursor
POP   AF          Inhalt von A und F wiederherstellen
CALL  &BB5A       M an der aktuellen Cursorposition ausgeben
LD    BC,&0101    Cursoroffset nach BC laden
POP   HL          HL vom Stapel holen
ADD   HL,BC       Offset zu HL addieren
PUSH  AF          Inhalt von AF auf Stapel retten
CALL  &BB75       Neue Cursorposition setzen
POP   AF          Inhalt von A wiederherstellen
CALL  &BB5A       M an der neuen Position ausgeben
RET
```

Nach dem Start des Programms werden zwei M's schräg untereinander auf dem Bildschirm ausgegeben. Zu dem Programm sind noch einige Erläuterungen notwendig. Zunächst sollte bemerkt werden, daß die Information in den Registern erst durch die Cursor-Routine zerstört wird. Die Rettungsaktion (PUSH) kopiert deren Inhalt nur, zerstört ihn also nicht. Erst die ROM-Routinen ändern die Registerinhalte in nicht vorhersehbarer Weise. Außerdem ist anzumerken, daß auch dann, wenn nur das A-Register zu retten ist, immer A und F auf den Stapel gerettet werden. Der PUSH-Befehl arbeitet nämlich nur mit Registerpaaren und nicht mit einzelnen Registern. Entsprechendes gilt natürlich für den POP-Befehl.

Übungsaufgabe 7.1

Schreiben Sie ein kurzes Programm, das

a) den Grafikcursor auf die Position (0,0) setzt,
b) die Koordinaten 100 nach DE und 200 nach HL lädt,
c) die Registerpaare DE und HL auf den Stapel rettet,
d) eine gerade Linie zum Punkt (100,200) zeichnet,
e) den Grafikcursor auf den Ausgangspunkt (0,0) zurücksetzt,
f) den ursprünglichen Inhalt von DE vom Stapel nach HL und den von HL nach DE lädt und
g) eine Linie zum neuen Punkt (200,100) zeichnet.

Hinweis: Nach Anhang D lauten die Einsprungadressen für das Setzen des Grafikcursors &BBC0 und zur Ausgabe der Linie &BBF6. Im Anhang G finden Sie eine mögliche Antwort zur Aufgabe.

Wenn Sie die Übungsaufgabe zuvor gelöst haben, werden Sie auf eine interessante Methode zum Austausch der Registerinhalte DE und HL gestoßen sein:

```
PUSH DE
PUSH HL
.
.
.
POP  DE
POP  HL
```

Die Register werden hier bewußt in der "falschen" Reihenfolge wiederhergestellt.

Es gibt noch einen anderen Weg, den Inhalt der genannten Register auszutauschen:

EX DE,HL Tausche den Inhalt von DE mit dem von HL.

Dieser Befehl arbeitet nur auf die angegebene Weise, d.h. EX HL,DE oder EX BC,HL oder ähnliche Kombinationen sind nicht erlaubt.

Der Stapelzeiger (SP)

Zur Verwaltung von PUSH- und POP-Operationen verwendet der Z80 ein spezielles Register, das als Stapelzeiger oder kurz SP bezeichnet wird. Im Normalfall verweist der Inhalt dieses Registers immer auf die zuletzt geladene Speicherstelle des Stapels.

Es gibt einige Befehle, die dem Programmierer die Möglichkeit eröffnen, auf den Inhalt des Stapelzeigers zuzugreifen und diesen zu ändern. Mit Hilfe dieser Befehle kann beispielsweise ein Benutzerstapel eingerichtet werden. Wenn das Register SP vom Anwender auf einen neuen Adreßwert gesetzt wird, wird der Z80 diesen als den Beginn eines neuen Stapels verwenden und den alten gänzlich vergessen. Wenn Sie von dieser Möglichkeit innerhalb eines Unterprogramms Gebrauch machen, müssen Sie auf jeden Fall jede Rücksprungadresse dadurch retten, daß Sie diese in den neuen Stapel einschreiben.

Die drei Befehle zur Manipulation des Stapelzeigers lauten:

LD SP,HL Lade den Stapelzeiger mit dem Inhalt von HL.

LD SP,IX Lade den Stapelzeiger mit dem Inhalt von IX.

LD SP,IY Lade den Stapelzeiger mit dem Inhalt von IY.

Den aktuellen Wert des Stapelzeigers können Sie mittels des Befehls

LD (nn),dd Lade die Speicherstelle nn mit dem Inhalt des Registerpaares dd.

abrufen. Neben BC, DE und HL kann dd nämlich auch SP sein.

Damit Sie sich davon überzeugen können, daß ein von Ihnen eingerichteter Benutzerstapel auch entsprechend arbeitet, sollten Sie das folgende Programm ausprobieren:

Programm 7.2

```
ENT
LD    A,43      Lade den ASCII-Code für + nach A
PUSH  AF        Rette A auf den Stapel
```

```
CALL  &BB5A       Gib das Zeichen + aus
LD    (&714A),SP  Aktuellen Wert von SP zwischenspeichern
LD    HL,&7148    Neuen Stapel
LD    SP,HL       bei &7148 einrichten
LD    A,61        ASCII-Code für = nach A laden.
PUSH  AF          A auf neuen Stapel retten
CALL  &BB5A       Zeichen = ausgeben
LD    HL,(&714A)  Alte Stapelposition wieder abrufen
LD    SP,HL       Zurück zum alten Stapel
POP   AF          Zeichen + abrufen
CALL  &BB5A       Zeichen ausgeben
RET
```

Das Programm 7.2 schreibt die Zeichenfolge +=+ auf den Bildschirm. Wäre der Stapelzeiger zwischendurch nicht auf eine neue Adresse gesetzt worden, hätte die Ausgabe +== gelautet.

Zum Schluß sollten wir noch die restlichen Befehle für das Arbeiten mit dem Stapel kennenlernen. Sie erlauben dem fortgeschrittenen (und mit Umsicht arbeitenden) Programmierer Stapelzugriffe ohne Benutzung von PUSH- und POP-Befehlen.

Zunächst drei weitere Austauschbefehle:

EX (SP),HL Tausche den Inhalt von HL mit dem obersten Stapelelement.

EX (SP),IX Tausche den Inhalt von IX mit dem obersten Stapelelement.

EX (SP),IY Tausche den Inhalt von IY mit dem obersten Stapelelement.

Weitere stapelorientierte Befehle lauten:

INC ss Erhöhe den Inhalt des Registerpaares ss.

DEC ss Erniedrige den Inhalt des Registerpaares ss.

Hierbei steht ss für BC, DE, HL oder SP.

ADD IX,pp Addiere den Inhalt des Registerpaares pp zum Inhalt von IX.

Hier ist pp entweder BC, DE, IX oder SP.

ADD IY,rr Addiere den Inhalt des Registerpaares rr zum Inhalt von IY.

Hier ist rr entweder BC, DE, IY oder SP.

Zusammenfassung

Wenn Sie dieses Kapitel sorgfältig studiert haben, sollten Sie die folgenden Begriffe und Befehle kennen:

LIFO
EX DE,HL
SP
PUSH
POP
EX (SP),HL
LD (nn),dd
Benutzerstapel
Sie sollten wissen, auf welches Element der Stapelzeiger verweist.

Kapitel 8

Blockverschiebungen und Blockvergleiche

Der Befehlssatz des Z80 enthält eine Reihe von Instruktionen, die eine einfache Verarbeitung ganzer Speicherbereiche erlauben, ohne daß im Programm jede einzelne Speicheradresse angegeben werden muß. Befehle dieses Typs können in zwei Kategorien eingeteilt werden. Mit den Blockoperationen werden Speicherblöcke von einem Bereich zu einem anderen übertragen. Blockvergleiche dienen zur Suche nach einem speziellen Datenelement innerhalb eines Speicherbereichs.

Blockverschiebungen

Der erste Befehl zur Blockverschiebung, den wir uns in diesem Kapitel ansehen wollen, lautet:

LDI Lade einen Speicherblock und erhöhe automatisch die Datenzeiger.

Bei der Benutzung dieses Befehls muß HL die Quelladresse und DE die Zieladresse enthalten. Nach der Befehlsausführung erhöht der Z80 automatisch die Inhalte von HL und DE. Außerdem erniedrigt er den Inhalt von BC. Dieser Sachverhalt erleichtert die Verwendung des LDI-Befehls innerhalb von Schleifen. BC kann dann als Schleifenzähler benutzt werden. Wenn BC den Wert 1 erreicht, wird das Überlaufbit auf 0 zurückgesetzt. Bei allen anderen Inhalten von BC setzt der LDI-Befehl dieses Statusbit auf den Wert 1. Das Schleifenende kann somit anhand des Zustands dieses Statusbits festgestellt werden. Wir werden darauf noch ausführlich zurückkommen. Zunächst wollen wir jedoch das Testprogramm 8.1 eingeben. Es zeigt, wie mit Hilfe des LDI-Befehls Speicherbereiche auf dem Bildschirm ausgegeben werden können. Im aktuellen Fall gibt es einen Bereich des Stapels (er liegt zwischen den Adressen &B199 und &BFFF) auf dem Bildschirm aus, dessen Bildwiederholspeicher von &C000 bis &FFFF reicht.

Programm 8.1

```
        ENT
        LD      HL,&B992    Lade HL mit der Startadresse des Datenblocks
        LD      DE,&C000    Lade DE mit der ersten Zieladresse
        LD      BC,&A1      Lade BC mit der Anzahl der zu übertragenden Datenwerte +1
LOOP:   LDI                 Kopiere das erste Byte
```

```
        JP      PO,ENDE:  Ende, wenn BC =1
        JP      LOOP:     Nächstes Byte
ENDE:   RET
```

Das Programm führt zu zwei horizontalen vielfarbigen Linien quer über den Bildschirm.

Die Parität

Wie bereits an anderer Stelle erwähnt, wird das Überlauf-/Paritybit des Statusregisters F beim LDI-Befehl dann und nur dann auf 0 gesetzt, wenn das Register BC den Wert 1 enthält. Für den Kopiervorgang muß demnach dieses Register die Anzahl zu übertragender Bytes plus 1 enthalten. Das Statusbit P/V kann auf ungerade (PO = Parity Odd) oder gerade (PE = Parity Even) Parität abgefragt werden.

Allgemein versteht man unter der Parität eines getesteten Datenwortes die Anzahl derjenigen Bits, die den Wert 1 aufweisen. Gerade Parität heißt nichts anderes, als daß diese Anzahl gerade, ungerade Parität, daß die Anzahl ungerade ist. Das Paritybit wird dann gesetzt, wenn die Anzahl der Einsen ungerade ist. Sollten Sie jemals in die Verlegenheit kommen, daß Sie die Parität eines Datenbytes testen wollen, dann erreichen Sie dies einfach dadurch, daß Sie den Akkumulator mit dem Wert 255 laden und diesen mit dem Datenbyte über eine UND-Operation verknüpfen. Als Folge davon wird das Paritybit im Statusregister entsprechend gesetzt. Es kann dann innerhalb eines Befehls wie beispielsweise dem bereits vorgestellten bedingten Sprung JP auf seinen logischen Zustand abgefragt werden.

Paritätstests werden im allgemeinen im Bereich der Datenkommunikation benutzt. Pro übertragenem Datenwort wird ein Bit als Paritybit zur Verfügung gestellt. Es ergänzt jedes der Datenworte auf gerade oder auch ungerade Parität. Bei Einzelfehlern auf der Übertragungsleitung kann dann empfängerseitig festgestellt werden, welches der Datenworte verfälscht wurde.

Übungsaufgabe 8.1

Schreiben Sie ein Programm ähnlich dem zuvor gezeigten Programm 8.1, das &3FFF aufeinanderfolgende Bytes ab der Adresse &B100 zur Anzeige auf dem Bildschirm in den Bildwiederholspeicher ab &C000 kopiert. Achtung: Seien Sie vorsichtig. Ein Programmfehler kann dazu führen, daß der Computer "abstürzt". Gegebenenfalls müssen Sie dann den Assembler neu laden.

Auch zu dieser Aufgabe finden Sie eine Lösung im Anhang G.

Das Lösungsprogramm zu Aufgabe 8.1 füllt den Bildschirm unmittelbar mit Zufallsmustern. Mit dem nachfolgend angegebenen Programm 8.2 geschieht dies ein wenig langsamer, so daß Sie den Vorgang noch verfolgen können.

Programm 8.2

```
        ENT
        LD      HL,&B100
        LD      DE,&C000
        LD      BC,&4000
LOOP:   LDI
        JP      PO,ENDE:
        LD      A,&FF
VERZOE: DEC     A           Verzögerungsschleife
        JP      NZ,VERZOE:
        JP      LOOP:
ENDE:   RET
```

Nach dem Start dieses Programms wird der Bildschirm zeilenweise mit Mustern gefüllt. Die auftretenden Zwischenräume werden nach und nach aufgefüllt.

Daß die Zeilen nicht sofort aneinander anschließen, ist auf die Art zurückzuführen, wie der Bildschirmcontroller die Informationen aus dem Bildwiederholspeicher verarbeitet. Unmittelbar aufeinanderfolgende Bytes des Bildwiederholspeichers (d.h. in dem Bereich zwischen &C000 und &FFFF) führen nicht in jedem Fall zur Ausgabe von unmittelbar aufeinanderfolgenden Bildschirminformationen. Verantwortlich dafür ist das "Betriebssystem" des CPC 464, nicht das Maschinenprogramm. Falls Sie sich für diesen Themenkreis näher interessieren, können Sie sich beispielsweise in dem Firmware-Handbuch der Firma Schneider genauer informieren.

Programm 8.2 demonstriert außerdem, daß ohne Probleme in der durch den LDI-Befehl eingeleiteten Schleife zusätzliche Programmbefehle vereinbart werden können. Nachteilig ist, wie Sie anhand von Programm 8.1 sehen können, daß Sie zur Aufrechterhaltung der Blockoperation auch dann immer wieder auf den LDI-Befehl zurückspringen müssen, wenn Sie gar keine zusätzlichen Befehle vorsehen wollen. Das kostet Zeit und Speicherplatz. In diesem Fall verwenden Sie besser den Befehl

LDIR Kopiere eine Speicherzelle, erhöhe die Datenzähler und wiederhole die Operation.

Er besitzt die gleiche Wirkung wie der LDI-Befehl mit Ausnahme davon, daß er so lange automatisch wiederholt wird, bis der Inhalt des Registerpaares BC 0 wird. Außerdem wird das P/V-Bit unabhängig vom Inhalt von BC immer auf 0 gesetzt. Dies ist im vorliegenden Fall ohne Belang, da es bei diesem Befehlstyp ja nicht

abgefragt zu werden braucht. Programm 8.3 ist eine umgeschriebene Version des Programms 8.1, in dem der LDI-Befehl durch LDIR ersetzt wurde.

Programm 8.3

```
ENT
LD    HL,&B992
LD    DE,&C000
LD    BC,&A0
LDIR
RET
```

Beachten Sie bitte, daß Sie im vorliegenden Fall das Register BC nicht mit der um 1 höheren Anzahl zu übertragender Bytes zu laden brauchen, da die wiederholte Befehlsausführung erst dann ausgesetzt wird, wenn BC gleich 0 und nicht - wie in Programm 8.1 - gleich 1 ist. BC braucht also nur mit der aktuellen Zahl zu kopierender Bytes geladen zu werden.

Neben den zuvor vorgestellten Blockoperationen kennt der Z80 noch zwei weitere:

LDD Kopiere ein Datenbyte, und erniedrige die Datenzeiger.

LDDR Kopiere ein Datenbyte, erniedrige die Datenzeiger, und wiederhole die Operation.

Sie entsprechen den bereits vorgestellten Befehlen LDI und LDIR mit Ausnahme davon, daß DE und HL nach jeder Befehlsausführung erniedrigt und nicht erhöht werden. Schauen Sie sich dazu das folgende Programm an, mit dessen Hilfe der Bildschirm von unten nach oben mit Informationen gefüllt wird.

Programm 8.4

```
        ENT
        LD    HL,&BFFF
        LD    DE,&FFFF
        LD    BC,&4000
LOOP:   LDD
        JP    PO,ENDE:
        LD    A,&FF
VERZOE: DEC   A
        JP    NZ,VERZOE:
        JP    LOOP:
ENDE:   RET
```


Vergleiche

In vielen Programmen erweist es sich als nützlich, wenn zwei Werte miteinander verglichen werden können. In Abhängigkeit von dem Ergebnis eines derartigen Vergleichs kann dann eine Entscheidung getroffen werden. Das funktioniert so ähnlich wie bei einem BASIC-Befehl des Typs IF....THEN. Der entsprechende Assemblerbefehl lautet:

CP s Vergleiche s mit A. s wird dazu von A abgezogen. Der Inhalt von A bleibt bei dieser Operation unverändert.

In dem Vergleichsbefehl ist s eines der Register A, B, C, D, E, H, L, (IX+d), (IY+d) oder (HL). Nach der internen Subtraktionsoperation steht das Ergebnis nicht explizit zur Verfügung, d.h. der Inhalt von A wird nicht zerstört. Beeinflußt werden nur das C-Bit, das Z-Bit, das V-Bit, das S-Bit und das H-Bit des Statusregisters F. Die Statusbits V, S und H werden an anderen Stellen dieses Buchs behandelt. Sie werden vornehmlich im Zusammenhang mit Vergleichsoperationen bei Zweierkomplementzahlen benutzt.

Da bei der Vergleichsoperation der Inhalt von s von A abgezogen wird, wird das Übertragsbit dann gesetzt, wenn s größer als A ist und im anderen Falle zurückgesetzt. Ist s=A, dann wird das Z-Bit gesetzt, im anderen Fall zurückgesetzt. Das Programm 8.5 zeigt, wie der CP-Befehl eingesetzt wird.

Programm 8.5

```
        ENT
        LD    A,&21     Zahl nach A laden
        LD    B,&40     Zahl nach B laden
        CP    B         Vergleiche B mit A
        JP    C,GRSSR:  C-Bit gesetzt: B>A
        JP    Z,GLCH:   Z-Bit gesetzt: B=A
        LD    A,&73     B<A: ASCII s nach A
        CALL  &BB5A     Ausgabe
        LD    A,&3C     ASCII < nach A
        CALL  &BB5A     Ausgabe
        LD    A,&41     ASCII A nach A
        CALL  &BB5A     Ausgabe
        RET             Rücksprung
GRSSR:  LD    A,&73     ASCII s nach A
        CALL  &BB5A     Ausgabe
        LD    A,&3E     ASCII > nach A
        CALL  &BB5A     Ausgabe
        LD    A,&41     ASCII A nach A
        CALL  &BB5A     Ausgabe
        RET             Rücksprung
```

```
GLCH:   LD      A,&73       ASCII s nach A
        CALL    &BB5A       Ausgabe
        LD      A,&3D       ASCII = nach A
        CALL    &BB5A       Ausgabe
        LD      A,&41       ASCII A nach A
        CALL    &BB5A       Ausgabe
        RET                 Rücksprung
```

Testen Sie das Programm auch mit &40 in A und &21 in B sowie mit &21 in A und B, und überzeugen Sie sich davon, daß es wie erwartet arbeitet.

Natürlich steht dem Z80 nicht nur die zuvor erläuterte Vergleichsoperation zur Verfügung. Er kennt auch noch einige Blockvergleichsoperationen. Zu nennen ist zuerst

CPI Vergleiche A mit einem Speicherbyte und erhöhe anschließend den Datenzeiger.

Genau wie beim Befehl LDI wird der Inhalt des Registerpaars BC erniedrigt, das als "Bytezähler" dient. Das Paritybit wird gesetzt, wenn der Inhalt von BC gleich 1 wird. Das Übertragsbit wird nicht durch diesen Befehl beeinflußt, wohl dagegen das Nullbit (Z-Bit). Das heißt, daß der CPI-Befehl nur für einen Test auf Gleichheit, nicht dagegen für einen Test auf "größer als" oder "kleiner als" benutzt werden kann. Das Programm 8.6 sucht einen Speicherbereich auf das Auftreten des Zeichens (ab und gibt für jedes gefundene Zeichen ein (auf dem Bildschirm aus.

Programm 8.6

```
        ENT
        LD      A,&28       ASCII-Zeichen ( nach A
        LD      HL,&0       Beginn des Speicherbereichs
        LD      BC,&0       64k absuchen (BC = Bytes+1, 0-1 =
                            FFFF)
LOOP:   CPI
        JP      PO,ENDE:    Wenn P-Bit = 1, dann Ende
        CALL    Z,GEFDN:    Wenn Z-Bit =1, dann Zeichen
                            ausgeben
        JP      LOOP:       Weitermachen
ENDE:   CALL    Z,GEFDN:    Alle Speicher getestet?
        RET                 Zurück zum Assembler
GEFDN:  CALL    &BB5A       Zeichen ausgeben
        RET                 Rückkehr
```


Eine Anmerkung ist zu diesem Programm notwendig. Die Routine GEFDN muß aus der Routine ENDE heraus aufgerufen werden, weil der Aufruf von &BB5A den Zustand des P/V-Bits für den Fall zerstören würde, daß das Z-Bit vor dem Parityvergleich (PO) abgefragt und ein Zeichen (gefunden würde. Wenn PO zuerst abgefragt wird, tritt dieses Problem nicht auf.

Ein weiterer Befehl für einen Blockvergleich lautet:

CPIR Vergleiche A mit dem Speicherinhalt, erhöhe den Datenzeiger und wiederhole die Operation.

Dieser Befehl entspricht CPI mit Ausnahme davon, daß er so oft wiederholt wird, bis eine Übereinstimmung zwischen Akkumulator und Speicherinhalt erzielt oder BC 0 wird. Das Programm kann leicht dahingehend abgeändert werden, daß CPI durch CPIR ersetzt wird. Das so modifizierte Programm läuft schneller ab, weil nicht nach jedem Vergleich auch noch ein gesonderter Test erfolgen muß.

Die beiden letzten Blockvergleiche entsprechen formal CPI und CPIR. Der Datenzähler HL wird jedoch nicht erhöht sondern erniedrigt.

CPD Vergleiche A mit dem adressierten Speicherbyte, und erniedrige anschließend den Datenzeiger HL.

CPDR Vergleiche A mit dem adressierten Speicherbyte, erniedrige den Datenzeiger HL, und wiederhole die Operation.

Im Gegensatz zu LDD und LDDR beeinflussen diese beiden Befehle das P/V-Bit des Statusregisters.

Wenn Sie jemals vor die Frage gestellt werden, welche der Statusbits C, S und P/V bei welchen Vergleichsoperationen beeinflußt werden, sollten Sie sich in den im Anhang angegebenen Tabellen näher informieren. Dies wird besonders dann der Fall sein, wenn Sie Vergleichsoperationen mit Datenwerten in Zweierkomplementform durchführen (d.h. mit Zahlen, die größer als 128 sind).

Zusammenfassung

Nach dem Studium dieses Kapitels sollten Sie mit den nachfolgend angegebenen Begriffen etwas anfangen können:

LDI	LDD
PE	LDDR
PO	CP s
Parität	CPI
CPD	Verzögerungsschleife
CPIR	LDIR
CPDR	

Kapitel 9

Spezielle Operationen und Unterbrechungen (Interrupts)

Dieses Kapitel beschäftigt sich mit einigen weniger wichtigen Eigenschaften des Z80, zumindest aus der Sicht der Software gesehen. Viele der in diesem Kapitel vorgestellten und erläuterten Operationen werden in der Praxis nur von wenigen Programmierern genutzt. Dennoch sollten wir sie etwas näher kennenlernen. Die nachfolgend erläuterten Unterbrechungsvorgänge erweisen sich in der Praxis zwar als sehr nützlich. Eine eingehende Behandlung dieses Themenkreises geht jedoch über das Ziel dieses Assemblerkurses hinaus.

Unterbrechungen

Wenn Sie gerade mit einer wichtigen Arbeit beschäftigt sind, werden Ihnen Unterbrechungen in der Regel wenig willkommen sein. Dem Z80 geht es nicht anders. Während der Programmablaufphase ist er einzig und allein damit beschäftigt, die Inhalte der Register zu kontrollieren und die Statusbits entsprechend zu setzen bzw. zurückzusetzen.

Durch spezielle äußere Ereignisse kann die Arbeit des Prozessors allerdings unterbrochen werden. Diesen Vorgang nennt man eine Unterbrechung. Das bedeutet, daß beispielsweise eine externe Komponente wie die Tastatur die Aufmerksamkeit des Prozessors erfordert. Bevor der Z80 sich um die Unterbrechung kümmern kann, muß er zunächst die aktuellen Inhalte der Register sowie die Zustände der Statusbits zwischenspeichern. Hierzu benutzt er üblicherweise den Stapel. Nach Abarbeiten der Unterbrechung müssen alle Register und Statusbits wiederhergestellt werden, damit der Prozessor mit seiner ursprünglichen Arbeit fortfahren kann.

Jedes Programm, das auf äußere Unterbrechungen reagieren oder auch nicht reagieren soll, bedarf einer sogenannten Unterbrechungsverarbeitung (Interrupt Handling). Wenn beispielsweise Daten von einem Computer zu einem anderen zu übertragen sind, muß in der Regel ein Quittungsbetrieb stattfinden (handshaking). Dies entspricht einem Dialog der Art "Ich bin fertig zum Senden, bist du bereit, zu empfangen?" - "Ja." - "Jetzt folgen die Daten .... Ende der Übertragung." - "Danke schön!"

Wenn eine Übertragung dieser Art willkürlich unterbrochen wird, ist es sehr wahrscheinlich, daß der Übertragungsvorgang gestört wird und die Daten wertlos werden. Es ist möglich, für den Zeitraum eines derartigen Übertragungsvorgangs

Unterbrechungen zu unterbinden (es gibt allerdings einige Ausnahmen). Unterbrechungen, die durch die Software unterdrückt werden können, werden maskierbare Unterbrechungen genannt. Unterbrechungsvorgänge, die nicht unterbunden werden können, heißen nichtmaskierbare Unterbrechungen (NMI = non-maskable interrupts).

Der Befehl zur Unterdrückung maskierbarer Unterbrechungen lautet

DI Unterdrücke maskierbare Unterbrechungen (Disable Interrupts).

Nach Abarbeitung des so geschützten Programmteils wird die Wirkung des Schutzes durch

EI Lasse maskierbare Unterbrechungen zu (Enable Interrupts).

wieder aufgehoben. Die Abarbeitung einer Unterbrechung durch den Prozessor geschieht im allgemeinen in Form eines in Maschinensprache geschriebenen Unterprogramms. In Übereinstimmung mit den üblichen Regeln für Unterprogramme muß es mit einem Rücksprungbefehl enden. Dieser lautet für nichtmaskierbare Unterbrechungen

RETN Kehre nach einer nichtmaskierbaren Unterbrechung zum Hauptprogramm zurück.

Für maskierbare Unterbrechungen lautet der Rücksprungbefehl

RETI Kehre nach einer maskierbaren Unterbrechung zum Hauptprogramm zurück.

Wenn Sie ein wenig über das zuvor Gesagte nachdenken, werden Sie darauf kommen, daß auch Unterbrechungen unterbrochen werden können. In so einem Fall sind Unterbrechungsprioritäten zu beachten. So kann im allgemeinen eine maskierbare Unterbrechung nicht eine nicht maskierbare Unterbrechung unterbrechen - sie muß in der Regel warten.

Die höchste Priorität besitzt die Busanfrage (BUSRQ = BUS ReQuest). Bei jeder anderen maskierbaren oder auch nichtmaskierbaren Unterbrechung wird der Prozessor zumindest den aktuellen Befehl zu Ende auszuführen. Nicht so beim BUSRQ. Bei dieser Art der Unterbrechungsanforderung reagiert der Z80 bereits beim nächstfolgenden Taktzyklus, egal ob er mit der Abarbeitung des gerade anliegenden Befehls fertig ist oder nicht.

Wie der Z80 im einzelnen auf maskierbare Unterbrechungen reagiert, hängt von dem aktuellen Unterbrechungsmodus ab. In jedem Fall unterbindet er zunächst weitere (maskierbare) Unterbrechungen und rettet den Inhalt des Programm-

zählers (PC) auf den Stapel. Vor der Rückkehr aus der Routine, die die Unterbrechung bedient, müssen immer mittels eines EI-Befehls weitere Unterbrechungen freigegeben werden.

Modi für maskierbare Unterbrechungen

Der Standardmodus ist der Modus 0. Er kann innerhalb eines Programms durch den Befehl

IM 0 Setze den Unterbrechungsmodus 0.

vereinbart werden. Nachdem der Z80 eine Unterbrechung im Modus 0 empfangen und akzeptiert hat, erwartet der Prozessor seitens des unterbrechenden Gerätes beim nächstfolgenden Taktzyklus einen Befehl auf dem Datenbus. Dieser Befehl ist in aller Regel der Aufruf eines Maschinenprogramms (CALL), das der Programmierer zur Abarbeitung bzw. Behandlung der Unterbrechung irgendwo im Speicher abgelegt hat. Es kann aber ebenso ein ReSTart-Befehl des Typs

RST n ReSTart bei Adresse n.

sein.

Bei der Adresse n handelt es sich um eine Ein-Byte-Adresse, die folgende Werte annehmen kann: &0, &8, &10, &18, &20, &28, &30 oder &38. Da RST ein Ein-Byte-Befehl ist, wird er gerne dann verwendet, wenn eine besonders schnelle Reaktion auf eine Unterbrechung erfolgen muß. Unter den eben genannten Ansprungadressen ist allerdings gerade nur soviel Platz, um dort den Sprung zur Startadresse eines Unterprogramms unterzubringen.

Sowohl bei einem Aufruf über CALL wie auch über RST wird der PC automatisch auf den Stapel gerettet. Der Z80 blendet gleichzeitig weitere maskierbare Unterbrechungen aus und arbeitet das der Unterbrechung zugehörige Unterprogramm ab. Wenn über den PC hinaus noch die Inhalte weiterer Register zu retten sind, dann muß das Unterprogramm mit Befehlen des nachfolgenden Typs beginnen:

Programm 9.1

```
PUSH AF
PUSH BC
PUSH DE
PUSH HL
PUSH IX
PUSH IY
```

Entsprechend muß es mit den Befehlen des nachfolgenden Typs enden:

Programm 9.2

```
POP IY
POP IX
POP HL
POP DE
POP BC
POP AF
EI
RETI
```

Beachten Sie die Befehle PUSH AF und POP AF, die gleichzeitig den Akkumulator und das Statusregister übertragen.

Der nächste Unterbrechungsmodus ist der Modus 1.

IM 1 Setze den Unterbrechungsmodus 1.

Dieser Modus entspricht weitgehend dem Modus 0, abgesehen davon, daß der Z80 in dieser Betriebsart direkt einen Restartbefehl des Typs RST &38 als Folge einer maskierbaren Unterbrechung ausführt. Darüber hinaus ignoriert der Z80 den Dateninhalt des Datenbusses im únmittelbar nach der Unterbrechungsanforderung folgenden Taktzyklus.

Der letzte Modus ist der Unterbrechungsmodus 2.

IM 2 Setze den Unterbrechungsmodus 2.

In diesem Modus springt der Z80 nach Beendigung des gerade in Arbeit befindlichen Befehls und nachdem er den PC gerettet und weitere Unterbrechungen unterbunden hat, zu jeder beliebigen geradzahligen Adresse. Das sind all jene, bei denen das niederwertige Bit gleich 0 ist. Die acht höchstwertigen Bits müssen in einem speziellen Register abgespeichert werden, das als I-Register bezeichnet wird. Es spielt die Rolle eines Adreßzeigers (I = interrupt vector). Die acht niederwertigen Bits stellt die unterbrechende Systemkomponente zur Verfügung. Auf diese Art und Weise kann der Z80 jede beliebige Adresse innerhalb einer Tabelle von 128 verschiedenen Adressen anspringen, deren höchstwertiges Byte im I-Register steht.

Der alternative Registersatz des Z80

Sie wissen bisher, daß der Z80 eine Fülle verschiedener Register besitzt, nämlich die Register A, B, C, D, E, H, L und das Statusregister F. Neben diesen Registern hat der Prozessor noch einen zweiten Satz von Registern, die im allgemeinen mit "gestrichelten" Bezeichnungen angegeben werden, also A´, B´, C´, D´, E´, H´, L´ und das Statusregister F´.

Obgleich diese alternativen Register prinzipiell vom Programmierer benutzt werden können, ist dies speziell auf dem Schneider CPC nicht ratsam. Sie werden nämlich durch einige Routinen des Betriebssystems verwendet. Grundsätzlich sind sie dem Programmierer mittels des Befehls

EXX Tausche die Registerpaare.

zugänglich. Dieser Befehl tauscht nämlich die Inhalte der Registerpaare BC, DE und HL mit BC´, DE´ und HL´ aus. Jede nachfolgende Operation mit diesen Registern wird somit mit den neuen Inhalten ausgeführt.

Wenn Sie also beispielsweise die Inhalte der Register B´ und C´ verwenden, müssen Sie diese vor dem Zurückschalten auf den originalen Registersatz und der Freigabe von Unterbrechungen wiederherstellen. Ein Programm, das von alternativen Registern Gebrauch macht, könnte wie folgt aussehen:

Programm 9.3

```
DI              Unterbrechungen verhindern
EXX             Register austauschen
PUSH  BC        Neuen Inhalt von BC retten
.
.
.
POP   BC        BC wiederherstellen
EXX             Register austauschen
EI              Unterbrechungen freigeben
```

Entsprechende Programme sollten möglichst kurz sein, weil viele automatische Unterbrechungen durch das Betriebssystem (beispielsweise durch die Timer oder die elektronische Tastaturabfrage) die alternativen Register benutzen. Zu lange Programme führen unter Umständen zu Problemen. Mit Sicherheit ist dies so, wenn immer Unterbrechungen ausgesetzt werden. (Weitere Hinweise finden Sie im Firmwarehandbuch.)

Ein weiterer Befehl macht den alternativen Akkumulator und das alternative Statusregister zugänglich. Er lautet:

EX AF,AF´ Tausche AF gegen AF´.

Der Inhalt des alternativen Statusregisters muß auf jeden Fall wiederhergestellt werden, bevor ein "Rücktausch" der Register erfolgt. Auch in diesem Fall müssen Unterbrechungen unterbunden werden, solange von den alternativen Registern

Gebrauch gemacht wird. Bitte beachten Sie, daß für den "Rücktausch" exakt derselbe Befehl, d.h. EX AF,AF´ und nicht EX AF´,AF verwendet wird.

Weitere Austauschbefehle

Es existieren noch zwei weitere Kategorien von Austauschbefehlen, die Sie kennen sollten:

EX DE,HL Tausche den Inhalt von DE gegen den Inhalt von HL aus.

und

EX (SP),HL Tausche die obersten Stapelelemente gegen den Inhalt von HL aus.

EX (SP),IX Tausche die obersten Stapelelemente mit dem Inhalt von IX aus.

EX (SP),IY Tausche die obersten Stapelelemente mit dem Inhalt von IY aus.

Die letzten drei Befehle bewirken, daß das oberste Element des Stapels mit dem Register L bzw. dem niederwertigen Byte von IX oder IY, das nächstfolgende Element mit H bzw. den höherwertigen Bytes von IX oder IY ausgetauscht wird.

Halt

Der Befehl

HALT Halte die Programmausführung an.

unterbricht die Arbeit des Z80, bis eine Unterbrechungsanforderung eintrifft. Während der Haltphase wird ausschließlich eine Wiederauffrischung der Speicherinhalte durchgeführt. Wie Sie wissen, vergißt ein Computer alle Programme und Daten in seinem Schreib-/Lesespeicher, wenn er abgeschaltet wird.

Das liegt daran, daß der Schreib-/Lesespeicher (RAM = Random Access Memory) aus dynamischen Speicherbausteinen aufgebaut ist. Das Besondere an diesem Speichertyp ist, daß er auch dann seine Informationen "vergißt", wenn er nicht in regelmäßigen Zeitabschnitten wieder aufgefrischt wird. Der Z80 benutzt zur Kontrolle dieses Wiederauffrischungsvorgangs ein spezielles Register.

Ein- und Ausgabeoperationen des Z80

Der Z80 besitzt eine Reihe von Ein-/Ausgabebefehlen für den Datenverkehr mit externen Geräten. Sie sind natürlich weitgehend Hardware-orientiert und werden aus diesem Grunde in diesem Einführungskurses nur kurz behandelt.

IN A,(n) Lade den Inhalt des Eingangstores n in den Akkumulator.

Unter einem Tor wird ein Anschluß zu einem externen Gerät verstanden. Die Zuordnung der Nummern zu den einzelnen Toren wird durch die Hardware, d.h. durch die elektronischen Komponenten des Systems festgelegt.

Die Wortbreite des jedem Tor zugeordneten Pufferspeichers beträgt beim Z80 8 bit. Der Befehl IN A,(n) beeinflußt nicht die Statusbits.

Der nachfolgend angegebene INPUT-Befehl überträgt die Daten jenes Tores, dessen Nummer im C-Register vereinbart wird, in irgendeines der Register A, B, C, D, E, H oder L:

IN r,(C) Lade den Inhalt des in C vereinbarten Tores in das Register r.

Dieser Befehl beeinflußt alle Statusbits mit Ausnahme des Übertragsbits.

Der Z80 kennt außerdem INPUT-Befehle, die sich ähnlich verhalten wie die bereits vorgestellten Befehle LDI, LDIR usw. Der erste dieser Befehle lautet

INI Übertrage den Inhalt des in C adressierten Tores in die durch HL spezifizierte Adresse, erniedrige B, und erhöhe HL.

Im Zusammenhang mit diesem Befehl kann B als Schleifenzähler eingerichtet werden, sofern dies erforderlich ist. Bitte beachten Sie, daß im Gegensatz zu beispielsweise LDI beim INI-Befehl nur das B-Register - nicht also das Registerpaar BC - benutzt wird. C muß die Nummer des Tores und HL die Ladeadresse enthalten. INI zerstört den Inhalt des S-, des H- und des P/V-Statusbits und setzt das N-Bit. Das C-Bit wird nicht beeinflußt. Das Z-Bit wird dann gesetzt, wenn B=1 wird.

INIR Lade den Inhalt des in C angegebenen Tores an die durch HL angegebene Adresse, erniedrige B, erhöhe HL, und wiederhole diese Operation, bis B gleich 0 ist.

Dieser Befehl entspricht in seiner Wirkung dem INI-Befehl mit Ausnahme davon, daß er so lange erneut ausgeführt wird, bis der Inhalt von B 0 geworden ist. Die

Statusbits werden mit Ausnahme des Z-Bits wie beim INI-Befehl beeinflußt. Das Z-Bit wird im vorliegenden Fall immer gesetzt.

IND Lade den Inhalt des in C angegebenen Tores in die durch HL angegebene Adresse, und erniedrige B und HL.

INDR Lade den Inhalt des in C angegebenen Tores in die durch HL angegebene Adresse, erniedrige B und HL, und wiederhole diese Operation, bis B=0 ist.

Die letzten beiden Befehle entsprechen den Befehlen INI und INIR mit Ausnahme davon, daß im vorliegenden Fall der Inhalt von HL nach jeder Befehlsausführung erniedrigt wird.

Nun wenden wir uns den Ausgabebefehlen zu.

OUT (n),A Gib den Inhalt von A auf dem Tor n aus.

OUT (C),r Gib den Inhalt des durch r bezeichneten Registers an dem in C vereinbarten Tor aus.

Der letzte dieser beiden Ausgabebefehle unterscheidet sich von dem entsprechenden Eingabebefehl IN r,(C) dadurch, daß er keines der Statusbits beeinflußt.

Weitere Ausgabebefehle lauten:

OUTI Gib den Inhalt der in HL angegebenen Speicherzelle an dem in C angegebenen Tor aus, erniedrige anschließend B, und erhöhe den Adreßzeiger HL.

OTIR Gib den Inhalt der in HL angegebenen Speicherzelle an dem in C angegebenen Tor aus, erniedrige anschließend B, erhöhe den Adreßzeiger HL, und wiederhole den Ausgabebefehl, bis der Inhalt von B gleich 0 ist.

OUTD Gib den Inhalt der in HL angegebenen Speicherzelle an dem in C angegebenen Tor aus, erniedrige anschließend B, und erniedrige den Adreßzeiger HL.

OTDR Gib den Inhalt der in HL angegebenen Speicherzelle an dem in C angegebenen Tor aus, erniedrige anschließend B, erniedrige den Adreßzeiger HL, und wiederhole den Ausgabebefehl, bis der Inhalt von B gleich 0 ist.

Die Statusbits werden durch die letztgenannten vier Ausgabebefehle in derselben Weise beeinflußt wie bei den entsprechenden Eingabebefehlen.

Zum Schluß noch ein recht nützlicher Befehl, den wir bisher noch nicht kennengelernt haben:

NOP Keine Operation.

Dieser Befehl bewirkt, daß der Z80 für die Dauer eines Maschinenzyklus (4 Taktzyklen) nichts tut. Er kann sich dann als ganz nützlich erweisen, wenn Verzögerungsschleifen auf einen genauen Zeitwert eingestellt werden sollen. Er kann auch dazu verwendet werden, um als Platzhalter für jeweils ein Byte zu dienen, wenn Sie Programme mit absoluten Adreßangaben anstelle von Labels benutzen.

Zusammenfassung

In diesem Kapitel wurden die folgenden Befehle behandelt:

DI	EX AF,AF´
EI	EX DE,HL
RETN	EX (SP),HL
RETI	EX (SP),IX
IM0	EX (SP),IY
IM1	HALT
IM2	IN A,(n)
RST n	IN r,(C)
EXX	INI
OUT (n),A	INIR
OUT (C),r	IND
OUTI	INDR
OTIR	NOP
OUTD	
OTDR	

Kapitel 10

Externe Befehle und Grafikerweiterungen

Dieses Kapitel erläutert Ihnen, wie Sie den BASIC-Interpreter mit zusätzlichen Befehlen erweitern können. Beispielsweise wird ein CIRCLE-Befehl zum Zeichnen eines Kreises vorgestellt werden. Alle Routinen sind ausschließlich in Maschinensprache geschrieben und mit dem Interpreter über Zusatzbefehle verknüpft. Wir werden dabei einige besonders fortschrittliche Eigenschaften des Assemblers kennenlernen. Bevor Sie sich jedoch nun an die Arbeit machen , sollten Sie auf jeden Fall zunächst Kapitel 6 gelesen und dessen Inhalt verstanden haben.

Externe Befehle (RSX-Befehle)

Alle unmittelbar verfügbaren Befehle des BASIC-Interpreters, wie beispielsweise LIST, GOTO, READ usw. werden als interne Befehle bezeichnet. Wenn Sie ein BASIC-Programm starten und der Interpreter auf einen internen Befehl stößt, sucht er den Festwertspeicher (ROM) auf dessen Bedeutung hin ab. Bei externen Befehlen bezieht er in seine Suche auch noch den Schreib-/Lesespeicher mit ein.

Ein externer Befehl (für ihn ist auch der Fachausdruck RSX = Resident System eXtension gebräuchlich) besteht aus einer Kette alphanumerischer Zeichen, denen ein vertikaler Balken (SHIFT und @) vorangestellt wird. Die nachfolgend angegebenen Befehle zählen alle zu den externen Kommandos

```
|CIRCLE   |TRIANGLE  |BOX
```

Geben Sie probeweise einmal unter Kontrolle des BASIC-Interpreters den Befehl |BOX ein. Der Computer wird erwartungsgemäß mit der Fehlermeldung "Unknown command" reagieren. Das liegt daran, daß der Interpreter den Befehl weder im ROM noch im RAM als zulässig finden kann. Es ist aber ganz einfach, dem Interpreter mitzuteilen, daß dieser externe Befehl legal ist. Und das geht so:

Schritt 1

Zunächst müssen Sie dem Computer mitteilen, daß externe Befehle hinzugefügt werden sollen. Dies wird dadurch erreicht, daß eine Befehlstabelle mit den neuen Kommandos eingerichtet wird. Diese Befehlstabelle dient als Adreßzeiger zu weiteren Tabellen, in denen Informationen über die neuen Kommandos enthalten sind, d.h. beispielsweise über die Syntax, die Lage im Speicher usw. Diese Information wird dadurch an das Betriebssystem übergeben, daß eine Hilfsroutine im

ROM aufgerufen wird, nachdem die entsprechenden Register sowohl mit den vom Betriebssystem geforderten Daten- wie auch Adreßwerten eines vier Byte großen Pufferspeichers gefüllt wurden.

Die Ansprungadresse dieser speziellen Routine lautet &BCD1. Sie verknüpft ein externes Kommando mit dem Betriebssystem, d.h. sie teilt dem Betriebssystem mit, daß ein bestimmtes Kommando existiert.

Beim Einsprung muß das Registerpaar BC die Startadresse der externen Befehlstabelle und HL die Startadresse eines vier Byte langen Puffers enthalten.

Wir wollen uns diesen Vorgang anhand eines Beispiels ansehen. Die Startadresse der externen Befehlstabelle wird nachfolgend unter dem Label EXCOMT (EXternal COMmand Table) vereinbart.

Die schnellste Methode, vier Bytes für einen Puffer zu reservieren, besteht darin, den Pseudobefehl DEFS des Assemblers zu benutzen. Die Startadresse des Puffers wird dem Label BUFF: zugeordnet.

Also:

```
BUFF:   DEFS &04
```

Das Programm sieht dann so aus (bitte noch nicht eingeben):

Programm 10.1

```
BUFF:   DEFS   &04            Puffer einrichten
        LD     BC,EXCOMT:     BC = Start der Befehlstabelle
        LD     HL,BUFF:       HL = Start des Puffers
        CALL   &BCD1          Verknüpfung mit Betriebssystem
        RET                   Zurück zu BASIC
```

Bitte geben Sie das Programm noch nicht ein. Denn auch, wenn der Computer die Lage der externen Befehlstabelle kennt, würde er bei seiner Suche dort keine neuen Befehle finden. Wir müssen die Befehlstabelle also erst mit neuen Befehlswörtern füllem.

Schritt 2

Weitere Einzelheiten der neuen Befehlstabelle müssen in einer zusätzlichen Tabelle abgespeichert werden. Die Adresse dieser Tabelle wird im Label NENAM (NEw NAMe) vereinbart. Die erste Zeile dieses Programmteils lautet demnach


```
EXCOMT: DEFW NENAM:
```

Die neuen Befehlsbezeichnungen werden dann über JP-Befehle hinzugefügt, beispielsweise

```
JP BOX:
```

Auf diese Weise wird eine Sprungadresse für den neuen Befehl BOX vereinbart. Unter der Annahme, daß nur ein externer Befehl vereinbart werden soll, lautet dieser zweite Programmteil

```
EXCOMT: DEFW  NENAM:
        JP    BOX:
```

Jetzt müssen wir in einem dritten Schritt nur noch die genauen Einzelheiten des neuen Befehls hinzufügen.

Schritt 3

Der neue Befehlsname wird, beginnend bei der Adresse, auf die NENAM verweist, in Form einer Kette von ASCII-Zeichen abgelegt. Das Betriebssystem des Schneider CPC 464 macht es erforderlich, daß das höchstwertige Bit (Bit 7) des letzten ASCII-Zeichens gesetzt wird. Am einfachsten erreichen wir dies dadurch, daß wir zu dessen ASCII-Wert die Zahl &80 addieren. Im vorliegenden Fall müssen wir also zum ASCII-Codewert des Buchstaben X eine &80 addieren. Der entsprechende Programmabschnitt lautet:

```
NENAM:  DEFM  "BO"
        DEFB  "X"+&80
        DEFB  &0
```

Die letzte &0 kennzeichnet das Ende der Tabelle.

Alle drei Programmteile zusammen sehen dann so aus:

Programm 10.2 (Bitte noch nicht assemblieren!)

```
        ORG   40000
BUFF:   DEFS  &04
        LD    BC,EXCOMT:
        LD    HL,BUFF:
        CALL  &BCD1
        RET
EXCOMT: DEFW  NENAM:
        JP    BOX:
```

```
NENAM:  DEFM  "BO"
        DEFB  "X"+&80
        DEFB  &0
```

Beachten Sie, daß ORG auf 40000 gesetzt wurde. Wenn Sie MEMORY auf 39999 setzen, vermeiden Sie, daß der BASIC-Interpreter über 39999 liegende Speicheradressen überschreibt. Der externe Befehl kann so nicht versehentlich zerstört werden.

Wenn Sie |BOX von der BASIC-Ebene aus eingeben, wird der Computer zu der durch das Label BOX festgelegten Adresse springen. Nun geben Sie bitte Programm 10.3 unmittelbar nach Programm 10.2 ein.

Programm 10.3

```
BOX:    LD    A,66
        CALL  PRINT:
        LD    A,79
        CALL  PRINT:
        LD    A,88
        CALL  PRINT:
        RET
PRINT:  EQU   &BB5A
```

Achten Sie auf die Verwendung des Labels PRINT:

Assemblieren Sie nun das ganze Programm. Es sollte zu folgendem Listing führen:

Programm 10.4

```
        ORG   40000
BUFF:   DEFS  &04
        LD    BC,EXCOMT:
        LD    HL,BUFF:
        CALL  &BCD1
        RET
EXCOMT: DEFW  NENAM:
        JP    BOX:
NENAM:  DEFM  "BO"
        DEFB  "X"+&80
        DEFB  &0
BOX:    LD    A,66
        CALL  PRINT:
        LD    A,79
        CALL  PRINT:
        LD    A,88
        CALL  PRINT:
        RET
PRINT:  EQU   &BB5A
```

Starten Sie das Programm mit CALL 40004 von der BASIC-Ebene aus. Obgleich nichts Sichtbares passiert, wurde dennoch der externe Befehl BOX dem Betriebssystem des Schneider CPC hinzugefügt. Kehren Sie zu BASIC zurück, und geben Sie

```
|BOX
```

ein. Auf dem Bildschirm wird BOX ausgegeben. Jedes Assemblerprogramm mit dem Label BOX kann offensichtlich mit diesem externen Befehl aufgerufen werden. Im vorliegenden Fall war dies der Programmteil 10.3, der nichts anderes bewirkt, als den Text BOX auf dem Bildschirm auszugeben. Genauso können wir ein Assemblerprogramm schreiben, das unter Ausnutzung der ROM-Routinen des Betriebssystems ein Rechteck auf dem Bildschirm ausgibt. Um die Größe dieses Rechtecks festzulegen, müssen wir einige Parameter an das entsprechende Assemblerprogramm übergeben. Schauen Sie sich dazu einmal die folgende kleine Skizze in Abb. 10.1 an.

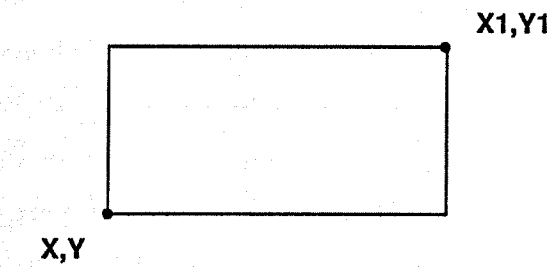


Abb. 10.1: Parameter für ein Rechteck

Ein Rechteck wird durch die diagonal gegenüberliegenden Eck-Koordinaten eindeutig festgelegt. Es wäre ganz nützlich, wenn neben der Größe auch noch die Farbe des Rechtecks definiert werden könnte. In diesem Fall müssen wir fünf Parameter (X, Y, X1, Y1 und C) an das Assemblerprogramm übergeben. Eine mögliche Lösung besteht darin, die entsprechenden Daten über POKE-Befehle direkt in einen Speicherblock einzuschreiben. Das ist zwar recht mühsam, aber es funk-

tioniert natürlich. Glücklicherweise erzeugt die Routine zur Erweiterung mit externen Kommandos solch einen Datenblock automatisch. Beim Aufruf ist die Startadresse im Registerpaar IX enthalten, die Anzahl der Parameter steht dagegen im Akkumulator. Schauen Sie sich einmal das folgende Kommando an, das ein Quadrat mit den Seitenabmessungen 100 *100 in der Farbe 2 bei der Koordinate 200,100 auf dem Bildschirm abbildet.

```
|BOX,200,100,300,200,2
```

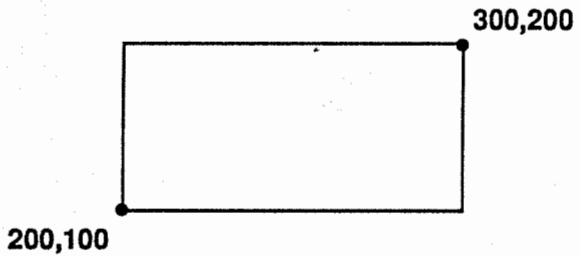


Abb. 10.2: Durch den Befehl |BOX,200,100,300,200,2 erzeugtes Rechteck

Bevor die Parameter für das Rechteck im Speicher abgelegt werden, werden sie in Hexadezimalwerte umgewandelt:

Schritt 1

```
100 = &0064
200 = &00C8
300 = &012C
  2 = &0002
```

Schritt 2

Die Parameter werden anschließend so abgespeichert, daß zuerst das niederwertige und dann das höherwertige Byte belegt wird. Das Register IX verweist auf den zuletzt eingegebenen Parameter. Die Werte werden somit in der nachfolgend gezeigten Art abgespeichert:

Speicher-adresse	Inhalt hexadezimal	dezimal
IX+0	02	2
IX+1	00	
IX+2	00	200

Speicher-adresse	Inhalt hexadezimal	dezimal
IX+3	C8	
IX+4	01	300
IX+5	2C	
IX+6	00	100
IX+7	64	
IX+8	00	200
IX+9	C8	

Mit einem entsprechenden Offset kann demnach auf jeden dieser Parameter zugegriffen werden. Bitte beachten Sie, daß jeder der Parameter zwei Bytes in Anspruch nimmt, d.h. jeder kann einen Wert im Bereich zwischen 0 und 65535 annehmen.

Die angegebenen Parameter werden dazu verwendet, das entsprechende Rechteck auf dem Bildschirm zu zeichnen.

Sehen Sie sich dazu die Abb. 10.3 an:

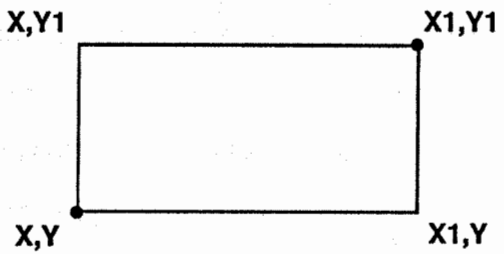


Abb. 10.3: Verwendung der Parameter zur Bestimmung der Koordinaten des Rechtecks

Aus den Eckkoordinaten X,Y und X1,Y1 werden die übrigen Koordinaten berechnet. Anschließend wird das Rechteck ausgegeben. Das nachfolgend gezeigte Programmablaufschema (Abb. 10.4) erläutert den Vorgang etwas anschaulicher.

Die Umsetzung in ein Assemblerprogramm wird in Programm 10.5 gezeigt (beachten Sie bitte, daß die Koordinaten unter Zuhilfenahme des Stapels zwischengespeichert werden)

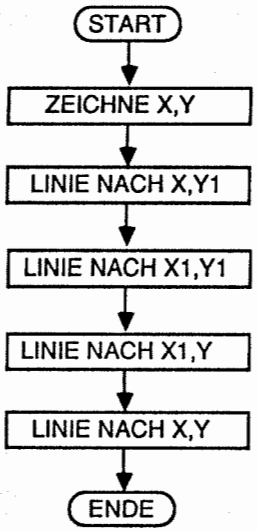


Abb. 10.4: Ablaufschema. Nach dieser Vorgabe müssen die Linien zu den einzelnen Koordinaten gezeichnet werden

Programm 10.5

```
10  DRAW:   EQU   &BBF6
20  PLOT:   EQU   &BBEA
30  INK:    EQU   &BBDE
40  BOX:    LD    A,(IX+0)
50          CALL  INK:
60          LD    E,(IX+8)
70          LD    D,(IX+9)
80          PUSH  DE
90          LD    E,(IX+6)
100         LD    D,(IX+7)
110         PUSH  DE
120         LD    E,(IX+4)
130         LD    D,(IX+5)
140         PUSH  DE
150         LD    E,(IX+2)
160         LD    D,(IX+3)
170         PUSH  DE
180         LD    E,(IX+8)
190         LD    D,(IX+9)
200         LD    L,(IX+6)
210         LD    H,(IX+7)
```



```
220         PUSH  DE
230         CALL  PLOT:
240         POP   DE
250         POP   HL
260         PUSH  HL
270         CALL  DRAW:
280         POP   HL
290         POP   DE
300         PUSH  DE
310         CALL  DRAW:
320         POP   DE
330         POP   HL
340         PUSH  HL
350         CALL  DRAW:
360         POP   HL
370         POP   DE
380         CALL  DRAW:
390         RET
```

Wie Sie sehen, wird die Lesbarkeit des Programms durch den Einsatz von Labels merklich erhöht. Ersetzen Sie nun das Unterprogramm mit dem Label BOX: im Programm 10.4 durch das Programm 10.5 und assemblieren Sie das gesamte Programm. Kehren Sie anschließend zum BASIC-Interpreter zurück, und geben Sie das nachfolgende BASIC-Programm ein. Das zuvor eingegebene externe Kommando wird in Zeile 10 über den Befehl Call 40000 mit dem Betriebssystem verknüpft.

Bitte geben Sie vor der Eingabe des Programms 10.6 den Befehl NEW, um Speicherkonflikte zu vermeiden. Der Assembler muß dann allerdings hinterher wieder neu geladen werden.

Programm 10.6

```
10  MODE 0: CALL 40000
20  X=RND(1)*550
30  Y=RND(1)*350
40  X1=RND(1)*50
50  Y1=RND(1)*15
60  C=RND(1)*15
70  |BOX,X,Y,X+X1,Y+Y1,C
80  GOTO 20
```

Damit sind wir fast am Ende des Abschnitts über die externen Befehle angelangt. Der weitere Teil dieses Kapitels erläutert noch andere Erweiterungen mit Grafik-Kommandos.

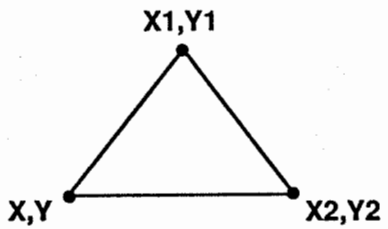
Weitere Grafikbefehle

1. |BOX,X,Y,X1,Y1,C zeichnet ein Rechteck auf dem Bildschirm mit der Farbe C.

2. |BOXF,X,Y,X1,Y1,C zeichnet ein Rechteck, das mit der Farbe C ausgefüllt wird.

3. |TRI,X,Y,X1,Y1,X2,Y2,C zeichnet ein Dreieck in der Farbe C.

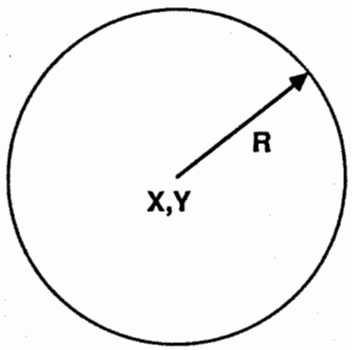
Hierbei gilt:



Bitte beachten Sie, daß die Werte von Y und Y2 übereinstimmen müssen.

4. |CIRCLE,X,Y,R,C

Hierbei gilt:



Die externen Befehle werden im Speicher ab der Adresse 40000 abgelegt. Damit sie nicht durch den BASIC-Interpreter zerstört werden können, muß der Zeiger für den zulässigen RAM-Bereich auf 39999 mittels des BASIC-Befehls

```
MEMORY 39999
```

herabgesetzt werden.

Jedes BASIC-Programm, das von den externen Befehlen Gebrauch macht, sollte daher mit den beiden Zeilen

```
1 MEMORY 39999
2 CALL 40000
```

beginnen.

Der BOXF-Befehl

Der externe BOXF-Befehl zeichnet ein ausgefülltes Rechteck. Es besteht aus einer Aufeinanderfolge horizontaler Linien.

Schauen Sie sich dazu Abb. 10.5 an.

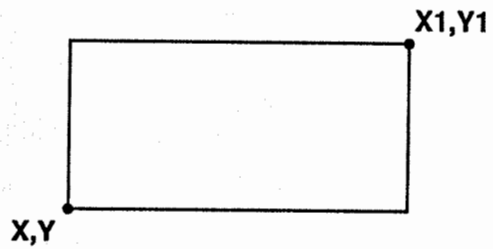
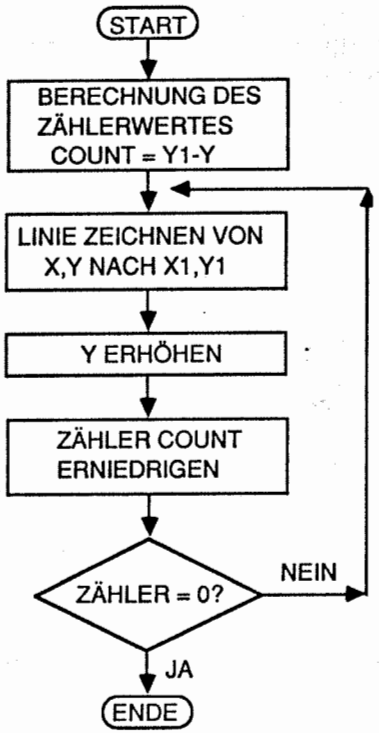


Abb. 10.5: Rechteckkoordinaten für den |BOXF-Befehl

Der Algorithmus sieht in Form eines Programmablaufschemas wie folgt aus:



In Form eines Programms sieht das so aus:

Programm 10.7

```
            ENT
            LD      BC,EXCOMT:
            LD      HL,COUNT:
            CALL    &BCD1
            RET
EXCOMT:     DEFW    NENAM:
            JP      BOXF:
NENAM:      DEFM    "BOX"
            DEFB    "F"+&80
            DEFB    &0
COUNT:      DEFS    4
DRAW:       EQU     &BBF6
PLOT:       EQU     &BBEA
INK:        EQU     &BBDE
BOXF:       LD      A,(IX+0)
            CALL    INK:
            LD      L,(IX+2)
            LD      H,(IX+3)
            LD      E,(IX+6)
            LD      D,(IX+7)
            AND     A
            SBC     HL,DE
            JP      C,END:
            LD      (COUNT:),HL
            LD      E,(IX+8)
            LD      D,(IX+9)
            LD      L,(IX+6)
            LD      H,(IX+7)
            PUSH    HL
            PUSH    DE
            CALL    PLOT:
            LD      E,(IX+4)
            LD      D,(IX+5)
            POP     IX
            POP     IY
NXT:        PUSH    DE
            PUSH    IY
            POP     HL
            CALL    DRAW:
            PUSH    IX
            POP     DE
            POP     IX
            INC     IY
            LD      HL,(COUNT:)
            PUSH    DE
            LD      DE,1
            AND     A
            SBC     HL,DE
            LD      (COUNT:),HL
            POP     DE
            JR      NZ,NXT:
END:        RET
```


Die dem Kurs beigefügte Magnetbandkassette enthält auf der B-Seite eine Datei, die mit GRAFEXT bezeichnet ist. Sie enthält alle zusätzlichen Grafikkommandos, die in diesem Kapitel vorgestellt und erläutert werden. Um diese Datei zu benutzen, laden Sie bitte zunächst den Assembler und unter dessen Kontrolle die ebengenannte Datei. Assemblieren Sie anschließend die Quelldatei. Wenn Sie die zusätzlichen Befehle häufiger innerhalb Ihrer BASIC-Programme verwenden wollen, sollten Sie eine Kopie der Objektdatei auf Band speichern. Kennzeichnen Sie diese einfach durch Anhängen von "-B" an den Dateinamen, also

GRAFEXT-B. Sie können die Datei dann immer unter Kontrolle des BASIC-Interpreters mit dem Befehl LOAD "GRAFEXT-B",40000 laden. Bitte stellen Sie auf jeden Fall sicher, daß der BASIC-Zeiger durch MEMORY 39999 verändert wird. Die Verknüpfung mit dem Betriebssystem erreichen Sie einfach durch den Aufruf CALL 40000. Wenn Sie sich nicht sicher sein sollten, wie die Befehle verwendet werden können, lassen Sie einfach das Demonstrationsprogramm mit dem Namen GRAFIKDEMO ablaufen.

Der TRI-Befehl

Mit dem externen TRI-Befehl läßt sich ein ausgefülltes Dreieck auf dem Bildschirm ausgeben. Hierfür wird hier der Einfachheit halber eine Näherungsmethode verwendet, d.h. es werden nacheinander gerade Linien von der Dreiecksspitze zu Punkten gezeichnet, die auf der unteren Seite des Dreiecks liegen. Diese Ausgabe wird links beginnend so lange fortgesetzt, bis die rechte untere Koordinate erreicht ist. Als Maß für die Schleifenabfrage wird die Differenz zwischen den X-Koordinatenwerten der Basisstrecke gewählt. Im Grunde genommen handelt es sich hierbei um eine wenig effektive Methode. Im Idealfall könnte jeder Punkt auf den Dreiecksseiten direkt ausgegeben werden. Hierzu sind aber etwas kompliziertere Algorithmen notwendig, deren Behandlung über das Ziel dieses Buchs hinausgeht. Falls Sie sich dafür interessieren, sollten Sie sich beispielsweise in dem englischsprachigen Buch "Fundamentals of Interactive Computer Graphics" von J.D. Foley und A. van Dam, erschienen bei Addison-Wesley (ISBN 0-201-14468-9) informieren.

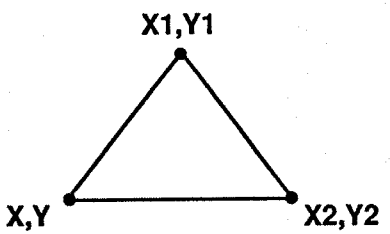


Abb. 10.7: Koordinatenangaben, die für das Dreieck benötigt werden

Das hier verwendete Programm sieht so aus:

Programm 10.8

```
        ORG   40000
DRAW:   EQU   &BBF6
PLOT:   EQU   &BBEA
INK:    EQU   &BBDE
```



```
BUFF:   LD    BC,EXCOMT:
        LD    HL,BUFF:
        CALL  &BCD1
        RET
EXCOMT: DEFW  NENAM:
        JP    TRI:
NENAM:  DEFM  "TR"
        DEFB  "I"+&80
        DEFB  0
COUNT:  DEFS  4
TRI:    LD    A,(IX+0)
        CALL  INK:
        LD    E,(IX+12)
        LD    D,(IX+13)
        LD    L,(IX+4)
        LD    H,(IX+5)
        AND   A
        SBC   HL,DE
        JP    M,END:
        LD    (COUNT:),HL
        LD    E,(IX+8)
        LD    D,(IX+9)
        LD    L,(IX+6)
        LD    H,(IX+7)
        PUSH  DE
        PUSH  HL
        LD    L,(IX+10)
        LD    H,(IX+11)
        PUSH  HL
        POP   IY
        LD    E,(IX+12)
        LD    D,(IX+13)
        PUSH  DE
        POP   IX
NXTT:   POP   HL
        POP   DE
        PUSH  DE
        PUSH  HL
        CALL  PLOT:
        PUSH  IX
        POP   DE
        PUSH  IY
        POP   HL
        CALL  DRAW:
        INC   IX
        LD    HL,(COUNT:)
```

```
        LD      DE,1
        AND     A
        SBC     HL,DE
        LD      (COUNT:),HL
        JR      NZ,NXTT:
        POP     DE
        POP     DE
END:    RET
```

Der CIRCLE-Befehl

Am einfachsten kann ein Kreis auf dem Bildschirm mit den Funktionen SIN und COS erzeugt werden. Schauen Sie sich dazu die Abb. 10.8 an, die den Zusammenhang zwischen dem Radius r eines Kreises, dem Drehwinkels α und den Bildpunktkoordinaten X und Y erläutert.

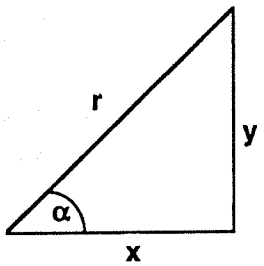


Abb. 10.8: Zusammenhang zwischen Radius, Winkel α und den kartesischen Koordinaten X und Y

Die X- und die Y-Koordinate können durch die Beziehungen

$X = r \cos(\alpha)$ und
$Y = r \sin(\alpha)$

errechnet werden. Wenn wir also den Wert des Winkels α im Bereich $0 < \alpha < \pi/2$ schrittweise erhöhen und an den entsprechenden Koordinatenpunkten X,Y Punkte ausgeben, erhalten wir einen Viertelkreis. Dieser Viertelkreis kann auf einfache Weise durch Spiegelung an den X- und Y-Achsen zu einem Vollkreis ergänzt werden. Abb. 10.9 erläutert dies für den Fall, daß der Mittelpunkt des Kreises mit dem Nullpunkt des Bildkoordinatensystems zusammenfällt.

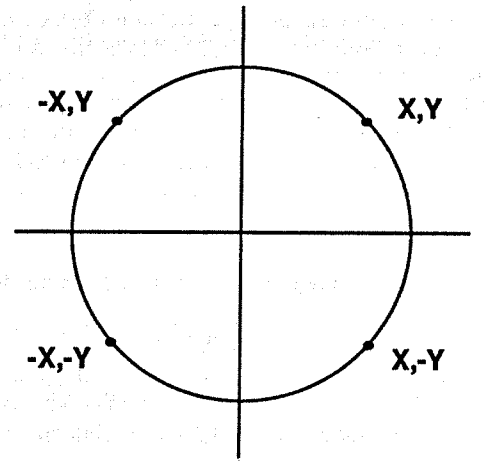
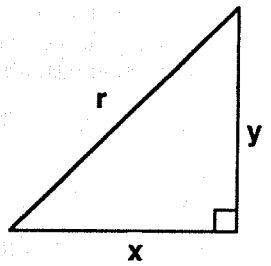


Abb. 10.9: Gewinnung der restlichen Kreislinienkoordinaten durch Spiegelung

Diese Vorgehensweise führt natürlich zum angestrebten Ziel. Vom Gesichtspunkt des Zeitbedarfs her gesehen, ist die Methode jedoch nicht besonders günstig, da die Berechnung der Funktionen COS und SIN beträchtliche Zeit in Anspruch nimmt. Ein schnellerer Algorithmus läßt sich aus dem Satz von Pythagoras ableiten. Dieser Satz besagt, daß in einem rechtwinkligen Dreieck gilt:



$$R^2 = X^2 + Y^2$$

Für die Seite Y folgt daraus:

$$Y = \pm\sqrt{R^2 - X^2}$$

Um einen Quadranten des Kreises zu zeichnen, muß X im Bereich $0 \leq X \leq R$ fortlaufend erhöht werden. Der Koordinatenwert Y folgt dann für einen vorgege-

benen Radius R unmittelbar aus X nach der zuvor angegebenen Formel. Schauen Sie sich den Algorithmus zunächst in Form eines BASIC-Programms an:

Programm 10.9

```
10  MODE 0
20  DEFINT X,Y,R
30  X = 0
40  R = 100
50  WHILE X<R
60  Y = SQR(R*R-X*X)
70  PLOT  X,Y
80  X = X + 1
90  WEND
100 END
```

Dieses Programm erzeugt einen Viertelkreis auf dem Bildschirm, wie in Abb. 10.10 gezeigt.

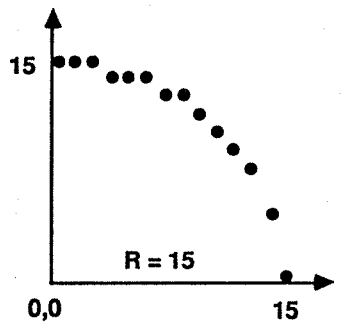


Abb. 10.10: Viertelkreis aus Einzelpunkten, berechnet nach dem Satz des Pythagoras

Die benutzte Methode hat allerdings zwei Nachteile.

1. Die Genauigkeit der Werteberechnung nimmt ab, wenn X gegen R strebt.

2. Auch die Benutzung der Quadratwurzelfunktion SQR benötigt einige Zeit, wenn auch weniger als bei der SIN- oder COS-Funktion.

Es gibt eine Reihe von Methoden, mit deren Hilfe das anstehende Problem gemildert werden kann. Wir werden in diesem Kapitel nur ein Verfahren anwenden. Es basiert auf einem von Bresenham entwickelten Algorithmus, der ursprünglich für mechanische Plotter entworfen wurde. Dieser Algorithmus ist um ein Vielfaches schneller als die beiden zuvor erwähnten Methoden, weil alle arithmetischen Operationen auf Additionen, Subtraktionen und Schiebeoperationen zurückgeführt werden können.

Der Algorithmus von Bresenham

Das Herz des Algorithmus ist eine Routine, die jene Pixel als gültig für die Kreisdarstellung aussucht, die dem wahren Kreis am nächsten liegen. Die Differenz zwischen dem Pixelwert und dem wahren Wert der Kreisfunktion wird als Fehlerterm bezeichnet. Er wird wie folgt abgeleitet.

Nach dem Satz von Pythagoras gilt

$$r^2 = x^2 + y^2$$

Wenn der Bildpunkt bei X,Y gezeichnet wird, dann folgt für den Fehler:

$$E = (x^2 + y^2) - r^2 \qquad \textit{Gleichung 10.1}$$

Dadurch, daß dieser Fehler E schrittweise minimiert wird, wird eine bestmögliche Näherung an den Kreis für das vorhandene Pixelnetz erzielt.

Schauen Sie sich dazu die Abb. 10.11 an, in der verschiedene Fälle für die Lage eines Kreisbogens im Bildpunktgitter gezeigt sind.

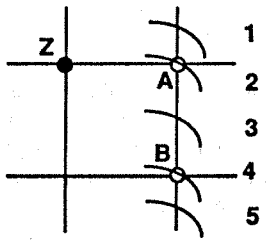


Abb. 10.11: Lage eines Kreisbogens im Bildpunktgitter

Es wird angenommen, daß gerade der schwarz gekennzeichnete Bildpunkt Z gesetzt wurde. Als nächstes kann entweder der Punkt A oder der Punkt B angesteuert werden. Wenn wir den Fehlerterm wie zuvor gesagt durch die Differenz

der quadrierten Abstände zwischen den Punkten A resp. B und dem Kreismittelpunkt ausdrücken, dann lassen sich die nachfolgend angegebenen Gleichungen ableiten:

Für Pixel A gilt:

$$E_A = (x_A^2 + y_A^2) - r^2 \qquad \textit{Gleichung 10.2}$$

Für Pixel B gilt:

$$E_B = (x_B^2 + y_B^2) - r^2 \qquad \textit{Gleichung 10.3}$$

Für den Fall, daß der Absolutwert $|E_A| \geq |E_B|$ ist, wird Pixel B gesetzt. Im umgekehrten Fall wird natürlich Pixel A gesetzt.

Die beiden Gleichungen 10.2 und 10.3 lassen sich zur Berechnung des Gesamtfehlers miteinander kombinieren.

$$E_{Ges} = E_A + E_B$$

In diesem Fall wird Pixel B gesetzt, wenn $E_{Ges} > 0$ ist, andernfalls wird Pixel A gesetzt.

Wenn Sie sich nochmals die fünf Fälle in Abb. 10.11 ansehen, dann gilt:

Fall 1: $E_{Ges} < 0 \rightarrow$ Pixel A wird gesetzt
Fall 2: $E_{Ges} < 0 \rightarrow$ Pixel A wird gesetzt
Fall 3: $E_{Ges} < 0 \rightarrow$ Pixel A wird gesetzt
Fall 4: $E_{Ges} > 0 \rightarrow$ Pixel B wird gesetzt
Fall 5: $E_{Ges} > 0 \rightarrow$ Pixel B wird gesetzt

In der zuvor erläuterten Form funktioniert die Methode einwandfrei. Immer noch müssen wir allerdings Quadrate und Quadratwurzeln von Werten berechnen, um den Fehlerterm zu ermitteln. Es kann jedoch bewiesen werden, daß der Anfangsfehler in erster Näherung auch wie folgt ermittelt werden kann:

$$E_{Ges} = 3 - 2r \qquad \textit{Gleichung 10.4}$$

Der so berechnete Wert hängt stark von der Lage des vorangegangenen Pixels ab, d.h. wenn Pixel A wegen $E_{Ges}<0$ ausgewählt wird, dann ist der neue Wert von E_{Ges} durch die Beziehung

$$E_{Ges+1} = E_{Ges} + 4x + 6 \qquad \textit{Gleichung 10.5}$$

gegeben. Für Pixel B gilt entsprechend

$$E_{Ges+1} = E_{Ges} + 4(x-y)+10 \qquad \textit{Gleichung 10.6}$$

Gleichung 10.5 macht zwei Additionen und eine Schiebeoperation, Gleichung 10.6 dagegen zwei Additionen, eine Subtraktion und zwei Schiebeoperationen erforderlich. Wir brauchen nunmehr nur noch ein Verfahren, wie wir die Pixels an den Koordinatenachsen spiegeln, um einen kompletten Kreis zu erhalten. Wenn wir die X- und Y-Werte des Kreisbogens in einem Quadranten kennen, können wir den Rest des Kreises durch einfache Spiegelung erzeugen. Schauen Sie sich dazu die Abb. 10.12 an.

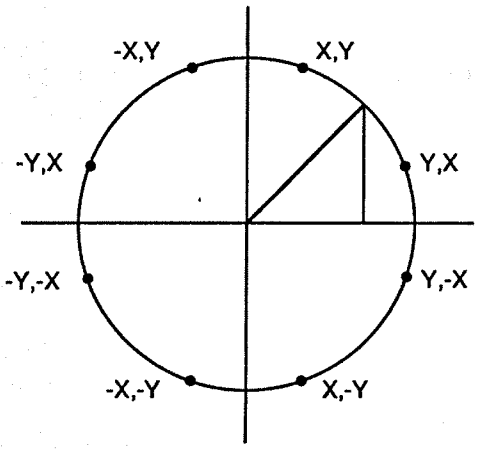


Abb. 10.12: Durch Spiegelung erhaltene Kreisbogenkoordinaten

Der zuvor erläuterte Algorithmus ist Bestandteil des nachfolgend angegebenen BASIC-Programms.

Programm 10.10

```
10  MODE 1
20  RADIUS = 100
30  X = 0
40  Y = RADIUS
50  ORIGIN 320,200
60  DIFF = 3-2*RADIUS
70  WHILE X<Y
```

```
80  GOSUB 150
90  IF D<0 THEN D=D+4*X+6: GOTO 120
100 D=D+4*(X-Y)+10
110 Y=Y-1
120 X=X+1
130 WEND
140 END
150 PLOT X,Y
160 PLOT Y,X
170 PLOT Y,-X
180 PLOT X,-Y
190 PLOT -X,-Y
200 PLOT -Y,-X
210 PLOT -Y,X
220 PLOT -X,Y
230 RETURN
```

Dieses Programm erzeugt für einen Radius von 15 eine Pixelverteilung nach Abb. 10.13.

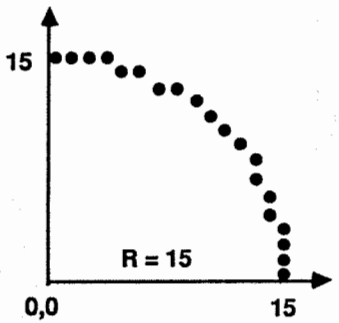


Abb. 10.13: Pixelverteilung nach Programm 10.10

Nachfolgend ist das Programm in Assemblersprache angegeben. Damit Sie es nicht unbedingt selbst eingeben müssen, sind alle externen BASIC-Befehle auf dem Band als Datei GRAFEXT enthalten.

Programm 10.11

```
        ORG     40000
        LD      BC,EXCOMT:
        LD      HL,BUFF:
        CALL    &BCD1
        RET
```



```
EXCOMT: DEFW    NENAM:
        JP      CIRCLE:
NENAM:  DEFM    "CIRCL"
        DEFB    "E"+&80
        DEFB    0
CIRCLE: CALL    ORGET:
        LD      (XO:),DE
        LD      (YO:),HL
        LD      A,(IX+0)
        CALL    INK:
        LD      D,(IX+7)
        LD      E,(IX+6)
        LD      H,(IX+5)
        LD      L,(IX+4)
        PUSH    HL
        LD      H,(IX+3)
        LD      L,(IX+2)
        LD      (RAD:),HL
        POP     HL
        CALL    ORIGIN:
        LD      BC,0000
        LD      (X:),BC
        LD      HL,(RAD:)
        LD      (Y:),HL
        SLA     L
        RL      H
        PUSH    HL
        POP     DE
        LD      HL,3
        XOR     A
        SBC     HL,DE
        LD      (DIFF:),HL
CALC:   LD      HL,(X:)
        LD      DE,(Y:)
        PUSH    HL
        PUSH    DE
        CALL    MIRIM:
        POP     DE
        POP     HL
        XOR     A
        SBC     HL,DE
        JP      P,END2:
        LD      HL,(DIFF:)
        LD      BC,0000
        SBC     HL,BC
        JP      P,LESS:
```

```
        LD      DE,(X:)
        SLA     E
        RL      D
        SLA     E
        RL      D
        LD      HL,6
        ADD     HL,DE
        LD      DE,(DIFF:)
        ADD     HL,DE
        JP      NXT3:
LESS:   LD      HL,(X:)
        LD      DE,(Y:)
        XOR     A
        SBC     HL,DE
        SLA     L
        RL      H
        SLA     L
        RL      H
        LD      DE,10
        ADD     HL,DE
        LD      DE,(DIFF:)
        ADD     HL,DE
        LD      DE,(Y:)
        DEC     DE
        LD      (Y:),DE
NXT3:   LD      (DIFF:),HL
        LD      HL,(X:)
        INC     HL
        LD      (X:),HL
        JP      CALC:
MIRIM:  LD      DE,(X:)
        LD      HL,(Y:)
        CALL    PLOT:
        LD      DE,(Y:)
        LD      HL,(X:)
        CALL    PLOT:
        LD      HL,0000
        LD      BC,(X:)
        XOR     A
        SBC     HL,BC
        PUSH    HL
        PUSH    HL
        LD      DE,(Y:)
        CALL    PLOT:
        POP     DE
        LD      HL,(Y:)
```



```
        CALL    PLOT:
        LD      HL,0000
        LD      BC,(Y:)
        XOR     A
        SBC     HL,BC
        PUSH    HL
        PUSH    HL
        LD      DE,(X:)
        CALL    PLOT:
        POP     DE
        LD      HL,(X:)
        CALL    PLOT:
        POP     HL
        POP     DE
        PUSH    HL
        PUSH    DE
        CALL    PLOT:
        POP     HL
        POP     DE
        CALL    PLOT:
        RET
END2:   LD      DE,(XO:)
        LD      HL,(YO:)
        CALL    ORIGIN:
        RET
BUFF:   DEFS    4
DIFF:   DEFS    2
RAD:    DEFS    2
X:      DEFS    2
Y:      DEFS    2
XO:     DEFS    2
YO:     DEFS    2
ORIGIN: EQU     &BBC9
ORGET:  EQU     &BBCC
PLOT:   EQU     &BBEA
INK:    EQU     &BBDE
DRAW:   EQU     &BBF6
```

Das war's. Sie haben es geschafft! Viel Spaß bei der Entwicklung eigener Programme!

Anhang A

Der Befehlssatz des Z80

Liste der Abkürzungen

Register

r,r'	irgendeines der Register A, B, C, D, E, H, L
dd	irgendeines der Registerpaare BC, DE, HL oder SP
qq	irgendeines der Registerpaare AF, BC, DE, HL
pp	irgendeines der Registerpaare BC, DE, IX oder SP
rr	irgendeines der Registerpaare BC, DE, IY oder SP
e	Offset in Zweierkomplementform
s	r, n, (HL), (IX+d), (IY+d)
d	r, (HL), (IX+d), (IY+d) oder innerhalb eines indizierten Befehls ein Offset in Zweierkomplementform
H	höherwertiges Byte
L	niederwertiges Byte
S	Statusregister mit b = Bit 0 bis 7

Adressierungsarten

RR	Register-Register
unm	unmittelbar
idz	indiziert
dir	direkt
Ind	indirekt
imp	implizit
ext	extern

Statusbits (Flags)

C	Übertragsbit (C-Bit)
Z	Nullbit (Z-Bit)
S	Vorzeichenbit (S-Bit)
P/V	Parity-/Überlaufbit (P/V-Bit)
H	Zwischenübertragsbit (H-Bit)
N	Subtraktionsbit (N-Bit)

Zustände von Statusbits

?	Bit wird in Abhängigkeit von der Operation gesetzt
0	Statusbit wird zurückgesetzt
1	Statusbit wird gesetzt
+	Statusbit wird nicht beeinflußt
-	Zustand unbekannt
V	Bit wird bei Überlauf gesetzt
P	Bit wird in Abhängigkeit von der Parität gesetzt
F	Das P/V-Bit wird in Abhängigkeit vom Unterbrechungs-Flip-Flop gesetzt.

Das P/V-Bit

Wenn in der letzten Rubrik der Befehlstabellen unter der Spalte P/V ein V angegeben ist, bedeutet das, daß im Falle eines Überlaufs das V-Bit gesetzt (1), im anderen Falle zurückgesetzt (0) wird. Wenn die Operation einen Paritätstest ausführt, wird dies durch ein P angedeutet. Das Bit wird dann gesetzt, wenn gerade Parität vorliegt. Bei ungerader Parität wird das Bit zurückgesetzt.

Adressierungsarten (Beispiel)

LD r,r´	Register-Register
LD r,n	unmittelbar
LD r,(IX+d)	indiziert
LD r,(nn)	direkt
LD r,(dd)	indirekt

Im vorliegenden Fall gilt:

r oder r	ist ein 8-bit-Register.
n	ist ein Datenwert mit 8 bit Wortbreite.
d	ist ein Datenwert in Zweierkomplementform.
nn	ist ein Datenwert mit 16 bit Wortbreite.
dd	ist eines der Register(paare) BC, DE, HL oder das 16-bit-Register SP.

8-bit-LD-Gruppe

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
LD r,r´	r ← r´	1	4	RR	+	+	+	+	+	+
LD r,n	r ← n	2	7	unm	+	+	+	+	+	+
LD r,(HL)	r ← (HL)	1	7	ind	+	+	+	+	+	+
LD r,(IX+d)	r ← (IX+d)	3	19	idz	+	+	+	+	+	+
LD r,(IY+d)	r ← (IY+d)	3	19	idz	+	+	+	+	+	+
LD (HL),r	(HL) ← r	1	7	ind	+	+	+	+	+	+
LD (IX+d),r	(IX+d) ← r	3	19	idz	+	+	+	+	+	+
LD (IY+d),r	(IY+d) ← r	3	19	idz	+	+	+	+	+	+
LD (HL),n	(HL) ← n	2	10	ind	+	+	+	+	+	+
LD (IX+d),n	(IX+d) ← n	4	19	idz	+	+	+	+	+	+
LD (IY+d),n	(IY+d) ← n	4	19	idz	+	+	+	+	+	+
LD A,(BC)	A ← (BC)	1	7	ind	+	+	+	+	+	+
LD A,(DE)	A ← (DE)	1	7	ind	+	+	+	+	+	+
LD A,(nn)	A ← (nn)	3	13	dir	+	+	+	+	+	+
LD (BC),A	(BC) ← A	1	7	ind	+	+	+	+	+	+
LD (DE),A	(DE) ← A	1	7	ind	+	+	+	+	+	+
LD (nn),A	(nn) ← A	3	13	dir	+	+	+	+	+	+
LD A,I	A ← I	2	9	RR	+	?	F	?	0	0
LD A,R	A ← R	2	9	RR	+	?	F	?	0	0
LD I,A	I ← A	2	9	RR	+	+	+	+	+	+
LD R,A	R ← A	2	9	RR	+	+	+	+	+	+

16-bit-LD-Gruppe

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
LD dd,nn	dd ← nn	3	10	unm	+	+	+	+	+	+
LD IX,nn	IX ← nn	4	14	unm	+	+	+	+	+	+
LD IY,nn	IY ← nn	4	14	unm	+	+	+	+	+	+
LD HL,(nn)	H ← (nn+1) L ← (nn)	3	16	dir	+	+	+	+	+	+
LD dd,(nn)	dd ← (nn+1) dd ← (nn)	4	20	dir	+	+	+	+	+	+
LD IX,(nn)	IX ← (nn+1) IX ← (nn)	4	20	dir	+	+	+	+	+	+

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
LD IY,(nn)	IY ← (nn+1) IY ← (nn)	4	20	dir	+	+	+	+	+	+
LD (nn),HL	(nn+1) ← H (nn) ← L	3	16	dir	+	+	+	+	+	+
LD (nn),dd	(nn+1) ← dd (nn) ← dd	4	20	dir	+	+	+	+	+	+
LD (nn),IX	(nn+1) ← IX (nn) ← IX	4	20	dir	+	+	+	+	+	+
LD (nn),IY	(nn+1) ← IY (nn) ← IY	4	20	dir	+	+	+	+	+	+
LD SP,HL	SP ← HL	1	6	RR	+	+	+	+	+	+
LD SP,IX	SP ← IX	2	10	RR	+	+	+	+	+	+
LD SP,IY	SP ← IY	2	10	RR	+	+	+	+	+	+

Stapelbefehle

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
PUSH qq	(SP-1) ← qq (SP-2) ← qq	1	11	ind	+	+	+	+	+	+
PUSH IX	(SP-1) ← IX (SP-2) ← IX	2	15	ind	+	+	+	+	+	+
PUSH IY	(SP-1) ← IY (SP-2) ← IY	2	15	ind	+	+	+	+	+	+
POP qq	qq ← (SP) qq ← (SP+1)	1	10	ind	+	+	+	+	+	+
POP IX	IX ← (SP) IX ← (SP+1)	2	14	ind	+	+	+	+	+	+
POP IY	IY ← (SP) IY ← (SP+1)	2	14	ind	+	+	+	+	+	+

Austauschbefehle

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
EX DE,HL	DE ↔ HL	1	4	RR	+	+	+	+	+	+
EX AF,AF´	AF ↔ AF´	1	4	RR	+	+	+	+	+	+
EXX	BC ↔ BC´ DE ↔ DE´ HL ↔ HL´	1	4	RR	+	+	+	+	+	+
EX (SP),HL	H ↔ (SP+1) L ↔ (SP)	1	19	ind	+	+	+	+	+	+
EX (SP),IX	IX ↔ (SP+1) IX ↔ (SP)	2	23	ind	+	+	+	+	+	+
EX (SP),IY	IY ↔ (SP+1) IY ↔ (SP)	2	23	ind	+	+	+	+	+	+

Blocktransfer- und Blockvergleichsbefehle

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
LDI	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	2	16	ind	+	+	?	+	0	0
LDIR	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1 solange, bis BC=0	2	21 (BC<>0) 16 (BC=0)	ind	+	+	0	+	0	0
LDD	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	2	16		+	+	?	+	0	0

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
LDDR	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1 solange, bis BC=0	2	21 (BC<>0) 16 (BC=0)	ind	+	+	0	+	0	0
CPI	A-(HL) HL ← HL-1 BC ← BC-1	2	16	ind	+	?	?	?	?	?
CPIR	A-(HL) HL ← HL+1 BC ← BC-1 solange, bis A=(HL) oder BC=0	2	16 (BC=0 oder A=(HL)) 21 (BC<>0 und A<>(HL))	ind	+	?	?	?	?	?
CPD	A-(HL) HL ← HL-1 BC ← BC-1	2	16	ind	+	?	?	?	?	?
CPDR	A-(HL) HL ← HL-1 BC ← BC-1 solange, bis A=(HL) oder BC=0	2	16 (BC=0 oder A=(HL)) 21 (BC<>0 und A<>(HL))	ind	+	?	?	?	?	?

8-bit-Arithmetik- und Logikbefehle

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
ADD A,r	A ← A+r	1	4	imp	?	?	V	?	0	?
ADD A,n	A ← A+n	2	7	unm	?	?	V	?	0	?
ADD A,(HL)	A ← A+(HL)	1	7	ind	?	?	V	?	0	?
ADD A,(IX+d)	A ← (IX+d)	3	19	idz	?	?	V	?	0	?
ADD A,(IY+d)	A ← (IY+d)	3	19	idz	?	?	V	?	0	?
ADC A,r	A ← A+r+C	1	4	RR	?	?	V	?	0	?
ADC A,n	A ← A+n+C	2	7	unm	?	?	V	?	0	?
ADC A,(HL)	A ← A+(HL)+C	1	7	ind	?	?	V	?	0	?
ADC A,(IX+d)	A ← A+(IX+d)+C	3	19	idz	?	?	V	?	0	?

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
ADC A,(IY+d)	A ← A+(IY+d)+C	3	19	idz	?	?	V	?	0	?
SUB A,r	A ← A-r	1	4	RR	?	?	V	?	1	?
SUB A,n	A ← A-n	2	7	unm	?	?	V	?	1	?
SUB A,(HL)	A ← A-(HL)	1	7	ind	?	?	V	?	1	?
SUB A,(IX+d)	A ← A-(IX+d)	3	19	idz	?	?	V	?	1	?
SUB A,(IY+d)	A ← A-(IY+d)	3	19	idz	?	?	V	?	1	?
SBC A,r	A ← A-r-C	1	4	RR	?	?	V	?	1	?
SBC A,n	A ← A-n-C	2	7	unm	?	?	V	?	1	?
SBC A,(HL)	A ← A-(HL)-C	1	7	ind	?	?	V	?	1	?
SBC A,(IX+d)	A ← A-(IX+d)-C	3	19	idz	?	?	V	?	1	?
SBC A,(IY+d)	A ← A-(IY+d)-C	3	19	idz	?	?	V	?	1	?
AND r	A ← A∧r	1	4	RR	0	?	P	?	0	1
AND n	A ← A∧n	2	7	unm	0	?	P	?	0	1
AND (HL)	A ← A∧(HL)	1	7	ind	0	?	P	?	0	1
AND (IX+d)	A ← A∧(IX+d)	3	19	idz	0	?	P	?	0	1
AND (IY+d)	A ← A∧(IY+d)	3	19	idz	0	?	P	?	0	1
OR r	A ← A∨r	1	4	RR	0	?	P	?	0	1
OR n	A ← A∨n	2	7	unm	0	?	P	?	0	1
OR (HL)	A ← A∨(HL)	1	7	ind	0	?	P	?	0	1
OR (IX+d)	A ← A∨(IX+d)	3	19	idz	0	?	P	?	0	1
OR (IY+d)	A ← A∨(IY+d)	3	19	idz	0	?	P	?	0	1
XOR r	A ← A∨r	1	4	RR	0	?	P	?	0	1
XOR n	A ← A∨n	2	7	unm	0	?	P	?	0	1
XOR (HL)	A ← A∨(HL)	1	7	ind	0	?	P	?	0	1
XOR (IX+d)	A ← A∨(IX+d)	3	19	idz	0	?	P	?	0	1
XOR (IY+d)	A ← A∨(IY+d)	3	19	idz	0	?	P	?	0	1
CP r	A-r	1	4	RR	?	?	V	?	1	?
CP n	A-n	2	7	unm	?	?	V	?	1	?
CP (HL)	A-(HL)	1	7	ind	?	?	V	?	1	?
CP (IX+d)	A-(IX+d)	3	19	idz	?	?	V	?	1	?
CP (IY+d)	A-(IY+d)	3	19	idz	?	?	V	?	1	?
INC r	r ← r+1	1	4	RR	+	?	V	?	0	?
INC (HL)	(HL) ← (HL)+1	1	11	ind	+	?	V	?	0	?
INC (IX+d)	(IX+d) ← (IX+d)+1	3	23	idz	+	?	V	?	0	?
INC (IY+d)	(IY+d) ← (IY+d)+1	3	23	idz	+	?	V	?	0	?
DEC r	r ← r-1	1	4	RR	+	?	V	?	1	?
DEC (HL)	(HL) ← (HL)-1	1	11	ind	+	?	V	?	1	?
DEC (IX+d)	(IX+d) ← (IX+d)-1	3	23	idz	+	?	V	?	1	?
DEC (IY+d)	(IY+d) ← (IY+d)-1	3	23	idz	+	?	V	?	1	?

Allgemeine Arithmetik und CPU-Steuerung

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
DAA	Dezimal-Anpassung	1	4	imp	?	?	P	?	+	?
CPL	A ← Nicht-A	1	4	imp	+	+	+	+	1	1
NEG	A ← 0-A	2	8	imp	?	?	V	?	1	?
CCF	C ← Nicht-C	1	4	imp	?	+	+	+	0	+
SCF	C ← 1	1	4	imp	1	+	+	+	0	0
NOP	keine Opera-	1	4	imp	+	+	+	+	+	+
HALT	CPU anhalten	1	4	imp	+	+	+	+	+	+
DI	IFF ← 0	1	4	imp	+	+	+	+	+	+
EI	IFF ← 1	1	4	imp	+	+	+	+	+	+
IM 0	Interrupt-Modus 0 setzen	2	8	imp	+	+	+	+	+	+
IM 1	Interrupt-Modus 1 setzen	2	8	imp	+	+	+	+	+	+
IM 2	Interrupt-Modus 2 setzen	2	8	imp	+	+	+	+	+	+

16-bit-Arithmetik

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
ADD HL,dd	HL ← HL+dd	1	11	RR	?	+	+	+	0	-
ADC HL,dd	HL ← HL+dd+C	2	15	RR	?	?	V	?	0	-
SBC HL,dd	HL ← HL-dd-C	2	15	RR	?	?	V	?	1	-
ADD IX,pp	IX ← IX+pp	2	15	RR	?	+	+	+	0	-
ADD IY,rr	IY ← IY+rr	2	15	RR	?	+	+	+	0	-
INC dd	dd ← dd+1	1	6	imp	+	+	+	+	+	+
INC IX	IX ← IX+1	2	10	imp	+	+	+	+	+	+
INC IY	IY ← IY+1	2	10	imp	+	+	+	+	+	+
DEC dd	dd ← dd-1	1	6	imp	+	+	+	+	+	+
DEC IX	IX ← IX-1	2	10	imp	+	+	+	+	+	+
DEC IY	IY ← IY-1	2	10	imp	+	+	+	+	+	+

Rotier- und Schiebebefehle

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
RLCA		1	4	imp	?	+	+	+	0	0
RLA		1	4	imp	?	+	+	+	0	0
RRCA		1	4	imp	?	+	+	+	0	0
RRA		1	4	imp	?	+	+	+	0	0
RLC r		1	8	imp	?	?	P	?	0	0
RLC (HL)		2	15	ind	?	?	P	?	0	0
RLC (IX+d)		4	23	idz	?	?	P	?	0	0
RLC (IY+d)		4	23	idz	?	?	P	?	0	0
RL r		2	8	imp	?	?	P	?	0	0
RL (HL)		2	15	ind	?	?	P	?	0	0
RL (IX+d)		4	23	idz	?	?	P	?	0	0
RL (IY+d)		4	23	idz	?	?	P	?	0	0
RRC r		2	8	imp	?	?	P	?	0	0
RRC (HL)		2	15	ind	?	?	P	?	0	0
RRC (IX+d)		4	23	idz	?	?	P	?	0	0
RRC (IY+d)		4	23	idz	?	?	P	?	0	0
RR r		2	8	imp	?	?	P	?	0	0
RR (HL)		2	15	ind	?	?	P	?	0	0
RR (IX+d)		4	23	idz	?	?	P	?	0	0
RR (IY+d)		4	23	idz	?	?	P	?	0	0
SLA r		2	8	imp	?	?	P	?	0	0
SLA (HL)		2	15	ind	?	?	P	?	0	0
SLA (IX+d)		4	23	idz	?	?	P	?	0	0
SLA (IY+d)		4	23	idz	?	?	P	?	0	0
SRA r		2	8	imp	?	?	P	?	0	0
SRA (HL)		2	15	ind	?	?	P	?	0	0
SRA (IX+d)		4	23	idz	?	?	P	?	0	0
SRA (IY+d)		4	23	idz	?	?	P	?	0	0
SRL r		2	8	imp	?	?	P	?	0	0
SRL (HL)		2	15	ind	?	?	P	?	0	0
SRL (IX+d)		4	23	idz	?	?	P	?	0	0

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
SRL (IY+d)		4	23	idz	?	?	P	?	0	0
RLD		2	18	ind	+	?	P	?	0	0
RRD		2	18	ind	+	?	P	?	0	0

Bitmanipulation

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
Bit b,r	Z ← b_r	2	8	imp	+	?	-	-	0	1
Bit b,(HL)	Z ← $b_{(HL)}$	2	12	ind	+	?	-	-	0	1
Bit b,(IX+d)	Z ← $b_{(IX+d)}$	4	20	idz	+	?	-	-	0	1
Bit b,(IY+d)	Z ← $b_{(IY+d)}$	4	20	idz	+	?	-	-	0	1
SET b,r	b_r ← 1	2	8	imp	+	+	+	+	+	+
SET b,(HL)	$b_{(HL)}$ ← 1	2	15	ind	+	+	+	+	+	+
SET b,(IX+d)	$b_{(IX+d)}$ ← 1	4	23	idz	+	+	+	+	+	+
SET b,(IY+d)	$b_{(IY+d)}$ ← 1	4	23	idz	+	+	+	+	+	+
RES b,r	b_r ← 0	2	8	imp	+	+	+	+	+	+
RES b,(HL)	$b_{(HL)}$ ← 0	2	15	ind	+	+	+	+	+	+
RES b,(IX+d)	$b_{(IX+d)}$ ← 0	4	23	idz	+	+	+	+	+	+
RES b,(IY+d)	$b_{(IY+d)}$ ← 0	4	23	idz	+	+	+	+	+	+

Sprungbefehle

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
JP nn	PC ← nn	3	10	unm	+	+	+	+	+	+
JP cc,nn	PC ← nn, falls Bedingung erfüllt	3	10	unm	+	+	+	+	+	+
JR e	PC ← PC+e	2	12	unm	+	+	+	+	+	+
JR cc,e	PC ← PC+e, falls Bedingung erfüllt	2	7/12	unm	+	+	+	+	+	+

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
JP (HL)	PC ← HL	1	4	imp	+	+	+	+	+	+
JP (IX)	PC ← IX	2	8	imp	+	+	+	+	+	+
JP (IY)	PC ← IY	2	8	imp	+	+	+	+	+	+
DJNZ e	B ← B-1 Falls B<>0, dann PC ← PC+e	2	8/13	unm	+	+	+	+	+	+

CALL- und RETURN-Gruppe

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S	N	H
CALL nn	(SP-1) ← PC (SP-2) ← PC PC ← nn	3	17	unm	+	+	+	+	+	+
CALL cc,nn	Falls Bedingung erfüllt, wie CALL nn	3	10/17	unm	+	+	+	+	+	+
RET	PC ← (SP) PC ← (SP+1)	1	10	ind	+	+	+	+	+	+
RET cc	Falls Bedingung erfüllt, wie RET	1	5/11	ind	+	+	+	+	+	+
RETI	PC ← (SP) PC ← (SP+1) SP ← SP+2	2	14	ind	+	+	+	+	+	+
RETN	PC ← (SP) PC ← (SP+1) SP ← SP+2	2	14	ind	+	+	+	+	+	+
RST p	(SP-1) ← PC (SP-2) ← PC PC ← 0 PC ← p	1	11	ind	+	+	+	+	+	+

Ein-/Ausgabe-Befehle

Mnemonic	Symbol Operation	Bytes	Takt-Zyklen	Adress.-Art	Statusbits C	Z	P/V	S.	N	H
IN A,(n)	A ← (n)	2	11	ext	+	+	+	+	+	+
IN r,(c)	r ← (c)	2	12	ext	+	?	P	?	0	?
INI	(HL) ← c B ← B-1 HL ← HL+1	2	16	ext	+	?	-	-	1	-
INIR	(HL) ← (c) B ← B-1 HL ← HL+1 solange, bis B=0	2	16/21	ext	+	1	-	-	1	-
IND	(HL) ← (c) B ← B-1 HL ← HL-1	2	16	ext	+	?	-	-	1	-
INDR	(HL) ← (c) B ← B-1 HL ← HL-1 solange, bis B=0	2	16/21	ext	+	1	-	-	1	-
OUT (n),A	(n) ← A	2	11	ext	+	+	+	+	+	+
OUT (c),r	(c) ← r	2	12	ext	+	+	+	+	+	+
OUTI	(c) ← (HL) B ← B-1 HL ← HL+1	2	16	ext	+	?	-	-	1	-
OTIR	(c) ← (HL) B ← B-1 HL ← HL+1 solange, bis B=0	2	16/21	ext	+	1	-	-	1	-
OUTD	(c) ← (HL) B ← B-1 HL ← HL-1	2	16	ext	+	?	-	-	1	-
OTDR	(c) ← HL) B ← B-1 HL ← HL-1 solange, bis B=0	2	16/21	ext	+	1	-	-	1	-

Anhang B

Wirkung von Befehlen auf die Statusbits

Das Statusregister F

Bit	Bezeichnung	Anmerkung
0	C	Übertrag
1	N	Addition/Subtraktion
2	P/V	Parity/Überlauf
3	-	nicht verwendet
4	H	Zwischenübertrag
5	-	nicht verwendet
6	Z	Nullbit
7	S	Vorzeichenbit

Wirkung auf das Vorzeichenbit

Gruppe	Befehl	Wirkung
8-bit-Ladeoperation	LD A,I	Gesetzt, wenn I-Register negativ, sonst zurückgesetzt.
	LD A,R	Gesetzt, wenn R-Register negativ, sonst zurückgesetzt.
Vergleiche	CPI, CPIR CPD, CPDR	Gesetzt, wenn Ergebnis negativ, sonst zurückgesetzt.
8-bit-Arithmetik	ADD A,S ADC A,S SUB S SBC A,S AND S OR S XOR S CP S INC S DEC S	Gesetzt, wenn Ergebnis negativ, sonst zurückgesetzt.
Allgemeine Arithmetik	DAA	Gesetzt, wenn MSB von A =1, sonst zurückgesetzt.
	NEG	Gesetzt, wenn Ergebnis negativ, sonst zurückgesetzt.
16-bit-Arithmetik	ADC HL,SS SBC HL,SS	Gesetzt, wenn Ergebnis negativ, sonst zurückgesetzt.
Einfache und zyklische Schiebe-Operationen	RLC S RL S RRC S RR S SLA S SRA S SRL S	Gesetzt, wenn Ergebnis negativ, sonst zurückgesetzt.
	RLD RRD	Gesetzt, wenn A nach der Schiebeoperation negativ ist, sonst zurückgesetzt.
Bitmanipulation	BIT B,S	Vorzeichenbit wird zerstört.
Ein- und Ausgabe-operationen	IN R,(C)	Gesetzt, wenn Eingangsdaten negativ, sonst zurückgesetzt.
	IN, INIR IND, INDR OUTI, OUTR OUTD,OUTDR	Inhalt des Vorzeichenbits geht verloren.

Wirkung der Befehle auf das Nullbit

Gruppe	Befehl	Wirkung
8-bit-Ladeoperationen	LD A,I	Gesetzt, wenn I-Register = 0, sonst zurückgesetzt.
	LD A,R	Gesetzt, wenn R-Register = 0, sonst zurückgesetzt.
Vergleichs-operationen	CPI,CPIR CPD, CPDR	Gesetzt, wenn A =(HL), sonst zurückgesetzt.
8-bit-Arithmetik	ADD A,S ADC A,S SUB S SBC A,S OR S XOR S CP S INC S DEC S	Gesetzt, wenn Ergebnis = 0, sonst zurückgesetzt.
Allgemeine Arithmetik	DAA NEG	Gesetzt, wenn Ergebnis = 0, sonst zurückgesetzt.
16-bit-Arithmetik	ADC HL,SS SBC HL,SS	Gesetzt, wenn Ergebnis = 0, sonst zurückgesetzt.
Einfache und zyklische Schiebe-operationen	RLC S RL RRC S RR S SLA S SRA S SRL S RLD RRD	Gesetzt, wenn das Ergebnis = 0, sonst zurückgesetzt.
Bitmanipulation	BIT B,S	Gesetzt, wenn spezifiziertes Bit = 0, sonst zurückgesetzt.
Ein- und Ausgabe-operationen	INR (C)	Gesetzt, wenn Eingangsdaten = 0, sonst zurückgesetzt.
	INI, IND INIR, INDR	Gesetzt, wenn B-1 = 0, sonst zurückgesetzt. Immer gesetzt.
	OUTI, OUTD OTIR, OTDR	Gesetzt, wenn B-1 = 0, sonst zurückgesetzt. Immer gesetzt.

Wirkung der Befehle auf das Zwischenübertragsbit

Gruppe	Befehl	Wirkung
8-bit-Ladebefehle	LD A,I LD A,R	Immer zurückgesetzt.
Blockladebefehle	LDI, LDIR LDD, LDDR	Immer zurückgesetzt.
Vergleiche	CPI, CPIR CPD, CPDR	Gesetzt, wenn kein Borger von Bit 4 auftritt, sonst zurückgesetzt.
8-bit-Arithmetik und -Logik	ADD A,S ADC A,S	Gesetzt, wenn kein Übertrag von Bit 3 auftritt, sonst zurückgesetzt.
	SUB S SBC A,S	Gesetzt, wenn kein Borger von Bit 4 auftritt, sonst zurückgesetzt.
	AND S	Immer gesetzt.
	OR S	Immer gesetzt.
	XOR S	Immer gesetzt.
	CP S	Gesetzt, wenn kein Borger von Bit 4 auftritt, sonst zurückgesetzt.
	INC S	Gesetzt, wenn Übertrag von Bit 3 auftritt, sonst zurückgesetzt.
	DEC S	Gesetzt, wenn kein Borger von Bit 4 auftritt, sonst zurückgesetzt.
Allgemeine Arithmetik	DAA	Inhalt geht verloren.
	CPL	Immer gesetzt.
	NEG	Gesetzt, wenn kein Borger von Bit 4 auftritt, sonst zurückgesetzt.
	CCF	Kein Einfluß.
	SCF	Immer zurückgesetzt.
16-bit-Arithmetik	ADD HL,SS ADC HL,SS	Gesetzt, wenn ein Übertrag aus Bit 11 auftritt, sonst zurückgesetzt.
	SBC HL,SS	Gesetzt, wenn kein Borger von Bit 12 auftritt, sonst zurückgesetzt.
	ADD IX,PP ADD IY,SS	Gesetzt, wenn ein Übertrag von Bit 11 auftritt, sonst zurückgesetzt.

Gruppe	Befehl	Wirkung
Einfache und zyklische Schiebe-operationen	RLC A RLA RRCA RRA RLC S RL S RRC S RR S SLA S SRA S SRL S RLD RRD	Immer zurückgesetzt.
Bitmanipulation	BIT B,S	Immer gesetzt.
Ein- und Ausgabe-operationen	IN R,(C)	Immer zurückgesetzt.
	INI, INIR	Inhalt geht verloren.
	IND, INDR	Inhalt geht verloren.
	OUTI, OTIR	Inhalt geht verloren.
	OUTD, OTDR	Inhalt geht verloren.

Wirkung der Befehle auf das Subtraktionsbit

Gruppe	Befehl	Wirkung
8-bit-Ladebefehle	LD A,I LD A,R	Immer zurückgesetzt.
Blockladebefehle	LDI, LDIR LDD, LDDR	Immer zurückgesetzt.
Vergleiche	CPI, CPIR CPD, CPDR	Immer gesetzt.
8-bit-Arithmetik	ADD A,S	Immer zurückgesetzt.
	ADC A,S	Immer zurückgesetzt.
	SUB S	Immer gesetzt.
	SBC A,S	Immer gesetzt.
	AND S	Immer zurückgesetzt.
	OR S	Immer zurückgesetzt.
	XOR S	Immer zurückgesetzt.
	CP S	Immer gesetzt.
	INC S	Immer zurückgesetzt.
	DEC S	Immer gesetzt.
	DAA	Keine Wirkung.
	CPL	Immer gesetzt.
	NEG	Immer gesetzt.
	CCF	Immer zurückgesetzt.
	SCF	Immer gesetzt.
16-bit-Arithmetik	ADD HL,SS	Immer zurückgesetzt.
	ADC HL,SS	Immer zurückgesetzt.
	SBC HL,SS	Immer gesetzt.
	ADD IX,PP	Immer zurückgesetzt.
	ADD IY,SS	Immer zurückgesetzt.
Einfache und zyklische Schiebe-operationen	RLC A RLA RRCA RRA RLC S RL S RRC S RR S SLA S SRA S SRL S RLD RRD	Immer zurückgesetzt.
Bitmanipulation	BIT B,S	Immer zurückgesetzt.
Ein- und Ausgabe-operationen	IN R,(C)	Immer zurückgesetzt.
	INI, INIR IND, INDR OUTI, OTIR OUTD, OTDR	Immer gesetzt.

Wirkung der Befehle auf das P/V-Bit

Gruppe	Befehl	Wirkung
8-bit-Lade-operationen	LD A,I LD A,R	Kopie des Unterbrechungs-Flip-Flops.
Blockbefehle	LDI, LDD CPI, CPIR CPD, CPDR	Gesetzt, wenn BC <>1, sonst zurückgesetzt.
	LDIR,LDDR	Immer zurückgesetzt.
8-bit-Arithmetik	ADD A,S ADC A,S SUB S SBC A,S	Gesetzt bei Überlauf, sonst zurückgesetzt.
	AND S OR S XOR S	Gesetzt bei gerader Parität, sonst zurückgesetzt.
	CP S	Gesetzt bei Überlauf, sonst zurückgesetzt.
	INC S	Gesetzt, wenn der Operand vor der Erhöhung den Wert &7F aufgewiesen hat, sonst zurückgesetzt.
	DEC S	Gesetzt, wenn der Operand vor der Erhöhung den Wert &80 aufgewiesen hat, sonst zurückgesetzt.
Allgemeine Arithmetik	DAA	Gesetzt, wenn (A) gerade Parität, sonst zurückgesetzt.
	NEG	Gesetzt, wenn (A) vor der Negation den Wert &80 enthielt, sonst zurückgesetzt.
16-bit-Arithmetik	ADC HL,SS SBC HL,SS	Gesetzt bei Überlauf, sonst zurückgesetzt.
Einfache und zyklische Schiebe-operationen	RLC S RL S RRC S RR S SLA S SRA S SRL S RLD S RRD S	Gesetzt bei gerader Parität, sonst zurückgesetzt.
Bitmanipulation	BIT B,S	Inhalt geht verloren.
Ein- und Ausgabe-operationen	IN R, (C)	Gesetzt bei gerader Parität, sonst zurückgesetzt.
	INI, INIR IND, INDR OUTI, OUTIR OUTD, OUTDR	Inhalt geht verloren.

Wirkung der Befehle auf das Übertragsbit

Gruppe	Befehl	Wirkung
8-bit-Lade-operationen	LD A,I LD A,R	Kein Einfluß.
Block-Befehle	CPI, CPIR CPD, CPDR LDI, LDIR LDD, LDDR	Kein Einfluß.
8-bit-Arithmetik	ADD A,S ADC A,S	Gesetzt, wenn Übertrag von Bit 7, sonst zurückgesetzt.
	SUB S	Gesetzt, wenn kein Borger auftritt, sonst zurückgesetzt.
	SBC A,S	
	AND S OR S XOR S	Immer zurückgesetzt.
	CP S	Gesetzt, wenn kein Borger auftritt, sonst zurückgesetzt.
Allgemeine arithmetische Operationen	DAA	Gesetzt bei Übertrag, sonst zurückgesetzt.
	NEG	Gesetzt, wenn A vor der Negation nicht 0 war, sonst zurückgesetzt.
	CCF	Gesetzt, wenn CY vor CCF 0 war, sonst zurückgesetzt.
	SCF	Immer gesetzt.
16-bit-Arithmetik	ADD HL,SS	Gesetzt bei Übertrag von Bit 15, sonst zurückgesetzt.
	SBC HL,SS	Gesetzt, wenn kein Borger auftritt, sonst zurückgesetzt.
	ADD IX,PP ADD IY,RR	Gesetzt bei Übertrag von Bit 15, sonst zurückgesetzt.
Einfache und zyklische Schiebe-operationen	RLC A RLA	Kopie von Bit 7 des Akkumulators.
	RRCA RRA	Kopie von Bit 0 des Akkumulators.
	RLC S RL S	Kopie von Bit 7 des Operanden.
	RRC S RR S	Kopie von Bit 0 des Operanden.
	SLA S	Kopie von Bit 7 des Operanden.
	SRA S	Kopie von Bit 0 des Operanden.
	SRL S	Kopie von Bit 0 des Operanden.

Anhang C

Die Wirkung von Vergleichsoperationen auf die Überlauf-, Vorzeichen- und Übertragsbits

A	S	8-bit	Zweier-komplement	V	S	C
30	20	A > s	A > s	0	0	0
20	C0	A < s	A > s	0	0	1
70	C0	A < s	A > s	1	1	1
C0	70	A > s	A < s	1	0	0
C0	20	A > s	A < s	0	1	0
20	30	A < s	A < s	0	1	1

s ist entweder A, B, C, D, E, H, L, (HL), (IX+d) oder (IY+d)

1 = Statusbit gesetzt
0 = Statusbit zurückgesetzt

Bedingungen:

Statusbit	gesetzt	zurückgesetzt
V	PO (ungerade)	PE (gerade)
S	P (positiv)	M (negativ)
C	C (Übertrag)	NC (kein Übertrag)

Wenn Vorzeichen- und Überlaufbit gleich sind, dann gilt A>s im Zweierkomplement. Wenn sie verschieden sind, dann gilt A<s unter der Annahme, daß das Z-Bit nicht gesetzt ist.

Anhang D

Auswahl von Einsprungadressen des Betriebssystems

&BB00 Tastatur-Manager initialisieren
(KM INITIALISE)

Aufgabe/Wirkung: Löscht den Tastatur-Puffer. Setzt die Wiederholrate der Tasten auf den Standardwert. SHIFT- und CAPS LOCK werden ausgeschaltet. Löscht Tastaturdefinitionen. Unterbrechungen durch BREAK sind nicht möglich

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register AF, BC, DE, HL gehen verloren. Alle anderen Register bleiben unverändert.

&BB06 Tastaturpufferabfrage mit Wartefunktion
(KM RESET)

Aufgabe/Wirkung: Fragt den Tastaturpuffer ab und wartet, bis eine Eingabe erfolgt.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: A enthält den ASCII-Code des Zeichens.

Anmerkung: Das Übertragsbit C wird gesetzt, alle anderen Statusbits werden zerstört.

&BB09 Tastaturpufferabfrage ohne Wartefunktion
(KM READ CHARACTER)

Aufgabe/Wirkung: Liest den Inhalt des Tastaturpuffers.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: A enthält das Zeichen oder ist zerstört.

Anmerkung: Übertragsbit ist nur gesetzt, wenn der Puffer ein Zeichen enthielt. Alle anderen Statusbits gehen verloren.

&BB18 Tastaturabfrage mit Wartefunktion (KM WAIT KEY)

Aufgabe/Wirkung: Tastaturabfrage mit Wartefunktion.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: A enthält das eingegebene Zeichen.

Anmerkung: Das Übertragsbit wird gesetzt, alle anderen Statusbits werden zerstört.

&BB1E Tasten- oder Tasterabfrage (KM TEST KEY)

Aufgabe/Wirkung: Die Routine ermittelt, ob eine vorgegebene Taste oder ein Taster eines Joysticks betätigt wurde.

Einsprungbedingungen: Der ASCII-Code muß in A geladen werden.

Übergabeparameter beim Aussprung: C: 128 = CTRL wurde betätigt.
32 = SHIFT wurde betätigt.
160 = SHIFT & CTRL wurden betätigt.

Anmerkung: Das Übertragsbit wird zurückgesetzt. Das Nullbit wird zurückgesetzt, wenn die angegebene Taste betätigt wurde, andernfalls wird das Nullbit gesetzt. Alle anderen Statusbits sowie A und HL werden zerstört.

&BB21 Tastaturstatus (KM GET STATE)

Aufgabe/Wirkung: Stellt fest, ob die SHIFT LOCK- oder die CAPS LOCK-Taste betätigt ist.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: L enthält eine 0, wenn SHIFT LOCK ausgeschaltet, eine 255, wenn SHIFT LOCK eingeschaltet ist.
H enthält eine 0, wenn CAPS LOCK ausgeschaltet, eine 255, wenn CAPS LOCK eingeschaltet ist.

Anmerkung: A und alle Statusbits werden zerstört.

&BB24 Joystick-Status (KM GET JOYSTICK)

Aufgabe/Wirkung: Fragt den Status der Joysticks ab.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: A und H enthalten den Status des Joysticks 0.
L enthält den Status des Joysticks 1.

Anmerkung: Bedeutung der Statusbits:

b7	b6	b5	b4	b3	b2	b1	b0
0	•	F1	F2	R	L	U	O

• = nicht benutzt
F1 = Feuerknopf 1
F2 = Feuerknopf 2
R = rechts
L = links
U = unten
O = oben

&BB39 Wiederholfunktion (KM SET REPEAT)

Aufgabe/Wirkung: Setzt/unterdrückt die Wiederholfunktion einer Taste.

Einsprungbedingungen: A muß mit dem Tastencode geladen werden. B = 0, wenn Wiederholfunktion ausgeschaltet werden soll. B = 255, wenn Wiederholfunktion aktiv werden soll.

Übergabeparameter beim Aussprung: keine

Anmerkung: A, B, C, H, L und die Statusbits werden zerstört.

&BB3C Test auf Wiederholfunktion (KM GET REPEAT)

Aufgabe/Wirkung: Abfrage des Wiederholstatus einer Taste.

Einsprungbedingungen: A muß mit dem Tastencode geladen werden.

Übergabeparameter beim Aussprung: Das Nullbit ist 0, wenn die Wiederholfunktion aktiv ist, andernfalls ist das Bit gesetzt. Das Übertragsbit wird zurückgesetzt.

Anmerkung: A und HL werden zerstört.

&BB3F Setzen von Verzögerung und Wiederholrate (KM SET DELAY)

Aufgabe/Wirkung: Die Routine setzt die Anfangsverzögerung und die Zeichenwiederholrate einer Taste.

Einsprungbedingungen: H muß mit dem Wert der Ansprechverzögerung geladen werden. L muß mit der Wiederholrate geladen werden. (Beide Werte entsprechen Vielfachen von 1/50 Sekunde, 0 = 255.)

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und die Werte der Statusbits werden zerstört.

&BB42 Abfrage von Ansprechverzögerung und Zeichenwiederholrate (KM GET DELAY)

Aufgabe/Wirkung: Die Routine fragt die aktuellen Werte der Ansprechverzögerung und der Zeichenwiederholrate ab.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: H enthält den Wert der Ansprechverzögerung. L enthält den Wert der Wiederholrate.

Anmerkung: A und F werden zerstört, alle anderen Register bleiben unverändert. Die in H und L stehenden Werte entsprechen ganzzahligen Vielfachen von 20 ms.

&BB4E Text-VDU initialisieren (TXT INITIALISE)

Aufgabe/Wirkung: Initialisieren der Text-VDU. Alle Textparameter (INK, PAPER, WINDOW usw.) werden auf die Standardwerte zurückgesetzt. Anwenderdefinierte Zeichen gehen verloren.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L sowie die Werte der Statusbits gehen verloren.

&BB51 Text-VDU zurücksetzen (TXT RESET)

Aufgabe/Wirkung: Die Routine setzt alle Control-Codes und Textparameter auf die Standardwerte.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte von A, B, C, D, E, H und L sowie die Werte der Statusbits gehen verloren.

&BB54 TEXT-VDU aktivieren
(TXT VDU ENABLE)

Aufgabe/Wirkung: Freigabe der Zeichenausgabe auf dem Bildschirm.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: keine

Anmerkung: Der Inhalt von A geht verloren. Die Werte der Statusbits werden zerstört. Der von TXT OUTPUT benutzte Control-Code-Puffer wird geleert.

&BB57 Text-VDU abschalten
(TXT VDU DISABLE)

Aufgabe/Wirkung: Die Routine schaltet die TEXT-VDU ab.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A und F gehen verloren. Alle anderen Registerinhalte bleiben unverändert. Der Control-Code-Puffer wird geleert.

&BB5A Zeichenübergabe an die VDU
(TXT OUTPUT)

Aufgabe/Wirkung: Die Routine gibt ein Zeichen oder einen Steuercode auf dem aktiven Ausgabekanal aus. Nach dem Systemstart ist dies der Bildschirm.

Einsprungbedingungen: A muß mit dem auszugebenden Zeichen oder Steuercode geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Alle Registerinhalte bleiben unverändert.

&BB5D Zeichenausgabe
(TXT WR CHAR)

Aufgabe/Wirkung: Die Routine gibt ein Zeichen auf dem Bildschirm aus. Control-Codes werden nicht beachtet. Der Cursor wird um eine Position weitergesetzt.

Einsprungbedingungen: A muß das auszugebende Zeichen oder den Control-Code enthalten.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L sowie die Werte der Statusbits gehen verloren.

&BB60 Zeichen lesen
(TXT RD CHAR)

Aufgabe/Wirkung: Es wird ein Zeichen an der aktuellen Position des Cursors eingelesen.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: Wenn an der Cursorposition ein Zeichen vorhanden ist, wird A mit dem Code geladen, das Übertragsbit wird gesetzt. Andernfalls sind Übertragsbit und der Inhalt von A gleich 0.

Anmerkung: Die Werte der übrigen Statusbits gehen verloren, die Inhalte der übrigen Register bleiben erhalten.

&BB63 Textausgabe im Grafikmodus
(TXT SET GRAPHIC)

Aufgabe/Wirkung: Textausgabe im Grafikmodus ein- bzw. ausschalten.

Einsprungbedingungen: A <> 0 zum Einschalten,
A = 0 zum Ausschalten.

Übergabeparameter beim Aussprung: Der Inhalt des Registers A und die Werte der Statusbits gehen verloren.

Anmerkung: Control-Codes werden ausgegeben, aber nicht ausgeführt.

&BB66 Textfenster setzen
(TXT WIN ENABLE)

Aufgabe/Wirkung: Die Routine setzt die Größe des aktuellen Textfensters. Der Cursor wird in die linke obere Ecke gesetzt.

Einsprungbedingungen: H: Spalteninformation für den linken Rand.
D: Spalteninformation für den rechten Rand.
L: Zeileninformation für den oberen Rand.
E: Zeileninformation für den unteren Rand.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und die Werte der Statusbits gehen verloren.

&BB6C Fenster löschen
(TXT CLEAR WINDOW)

Aufgabe/Wirkung: Das aktuelle Fenster wird auf die Hintergrundfarbe und der Cursor in die linke obere Ecke gesetzt.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und die Werte der Statusbits gehen verloren.

&BB6F Textspalte
(TXT SET COLUMN)

Aufgabe/Wirkung: Setzen der horizontalen Cursorkoordinate.

Einsprungbedingungen: A muß mit dem Spaltenwert geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, H und L sowie die Werte der Statusbits gehen verloren.

&BB72 Textzeile
(TXT SET ROW)

Aufgabe/Wirkung: Setzt die vertikale Cursorkoordinate.

Einsprungbedingungen: A muß den Zeilenwert enthalten.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, H und L sowie die Werte der Statusbits gehen verloren.

&BB75 Cursor setzen
(TXT SET CURSOR)

Aufgabe/Wirkung: Positioniert den Cursor.

Einsprungbedingungen: H muß mit dem Spaltenwert und L mit dem Zeilenwert geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Spalten- wie auch Zeilenwert sind auf das Textfenster bezogen. Die Inhalte der Register A, H und L sowie die Werte der Statusbits gehen verloren.

&BB78 Cursorposition lesen
(TXT GET CURSOR)

Aufgabe/Wirkung: Abfrage der aktuellen Cursorposition.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: H enthält die Spalten- und L die Zeileninformation. A enthält einen Wert, der aussagt, wie oft das aktuelle Fenster "gerollt" wurde.

Anmerkung: Die Werte der Statusbits gehen verloren, alle anderen Registerinhalte bleiben unverändert.

&BB87 Test Cursorposition
(TXT VALIDATE)

Aufgabe/Wirkung: Die Routine testet, ob der Cursor sich an einer vorgegebenen Position innerhalb eines Fensters befindet. Ist dies nicht der Fall, wird jene Ausgabekoordinate ermittelt, bei der ein Zeichen bei der Ausgabe erscheinen würde.

Einsprungbedingungen: H muß mit dem Spalten-, L mit dem Zeilenwert der zu testenden Koordinate geladen werden.

Übergabeparameter beim Aussprung: H enthält die Spalten- und L die Zeileninformation.
Fall 1: Die in H und L enthaltenen Werte führen zu keinem Rollvorgang des Textes:
Übertragsbit = 1, B geht verloren.
Fall 2: Die in H und L enthaltenen Werte führen bei der Zeichenausgabe zu einem Aufwärtsrollen:
Übertragsbit = 0, B = &FF.
Fall 3: Die in H und L enthaltenen Werte führen bei der Zeichenausgabe zu einem Abwärtsrollen:
Übertragsbit = 0, B = &00.

&BB90 Vordergrundfarbe setzen
(TXT SET PEN)

Aufgabe/Wirkung: Setzen der Stiftfarbe.

Einsprungbedingungen: A muß mit der Stiftfarbe geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Der Farbwert in A wird mit einer Maske logisch verknüpft, um in jedem Bildschirmmodus eine sichtbare Stiftfarbe zu gewährleisten.

Die Maskenwerte lauten:

MODE 0 = &0F
MODE 1 = &03
MODE 2 = &01

Die Inhalte der Register A, H, L und F gehen verloren.

&BB96 Hintergrundfarbe setzen
(TXT SET PAPER)

Aufgabe/Wirkung: Setzen der Hintergrundfarbe (PAPER).

Einsprungbedingungen: A muß mit dem Farbwert geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Der in A vereinbarte Farbwert wird mit einer Maske logisch verknüpft, um die für den jeweiligen Bildschirmmodus gültigen Farben zu gewährleisten.

Die Maskenwerte lauten:

MODE 0 = &0F
MODE 1 = &03
MODE 2 = &01

Die Inhalte der Register A, H, L und F gehen verloren.

&BB9C Farben invertieren
(TXT INVERSE)

Aufgabe/Wirkung: Austauschen von Vorder- und Hintergrundfarbe.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, H, L und F gehen verloren.

&BBBA Grafik-VDU initialisieren
(GRA INITIALISE)

Aufgabe/Wirkung: Die Grafik-VDU wird auf die Standardwerte gesetzt.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und die Werte der Statusbits gehen verloren.

&BBC0 Absolute Bewegung des Grafikcursors (GRA MOVE ABSOLUTE)

Aufgabe/Wirkung: Der Grafikcursor wird auf eine definierte Pixelposition gesetzt.

Einsprungbedingungen: DE muß mit der X-Koordinate geladen werden. HL muß mit der Y-Koordinate geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und die Werte der Statusbits gehen verloren.

&BBC3 Relative Bewegung des Grafikcursors (GRA MOVE RELATIVE)

Aufgabe/Wirkung: Der Grafikcursor wird auf eine definierte Pixelposition relativ zur vorhergehenden gesetzt.

Einsprungbedingungen: DE muß mit dem X-Koordinatenoffset geladen werden. HL muß mit dem Y-Koordinatenoffset geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und die Werte der Statusbits gehen verloren.

&BBC6 Position des Grafikcursors lesen

Aufgabe/Wirkung: Die Routine ermittelt die aktuellen Koordinaten des Grafikcursors.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: DE enthält die X-Koordinate. HL enthält die Y-Koordinate.

Anmerkung: Die Inhalte von A und F gehen verloren.

&BBC9 Koordinaten-Nullpunkt setzen (GRA SET ORIGIN)

Aufgabe/Wirkung: Die Routine definiert die aktuelle Lage des Koordinaten-Nullpunktes für grafische Ausgaben.

Einsprungbedingungen: DE muß mit der X-Koordinate geladen werden. HL muß mit der Y-Koordinate geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und F gehen verloren.

&BBCC Abfrage Koordinaten-Nullpunkt (GRA GET ORIGIN)

Aufgabe/Wirkung: Die Routine ermittelt die aktuelle Lage des Koordinaten-Nullpunktes.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: DE enthält die X-Koordinate. HL enthält die Y-Koordinate.

Anmerkung: Alle anderen Register bleiben unverändert. Die X- und die Y-Koordinate werden in Standardkoordinaten angegeben. Das Koordinatenpaar 0,0 entspricht der linken unteren Ecke des Bildschirms.

&BBCF Fensterweite (GRA WIN WIDTH)

Aufgabe/Wirkung: Setzen des linken und des rechten Randes des Grafikfensters.

Einsprungbedingungen: DE muß mit der X-Koordinate des linken Randes geladen werden. HL muß mit der X-Koordinate des rechten Randes geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und F gehen verloren.

&BBD2 Fensterhöhe
(GRA WIN HEIGHT)

Aufgabe/Wirkung: Setzen des unteren und des oberen Randes des Grafikfensters.

Einsprungbedingungen: DE muß mit der X-Koordinate des unteren Randes geladen werden. HL muß mit der X-Koordinate des oberen Randes geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und F gehen verloren.

&BBDB Grafikfenster löschen
(GRA CLEAR WINDOW)

Aufgabe/Wirkung: Fenster auf Hintergrundfarbe setzen.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und F gehen verloren.

&BBDE Vordergrundfarbe setzen (Grafik)
(GRA SET PEN)

Aufgabe/Wirkung: Setzen der Vordergrundfarbe für den Grafikstift.

Einsprungbedingungen: A muß mit dem gewünschten Farbwert geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Der in A stehende Wert wird mit einer Maske logisch verknüpft, um eine für den jeweiligen Betriebsmodus gültige Vereinbarung zu gewährleisten.

Die Maskenwerte lauten:

MODE 0 = &0F
MODE 1 = &03
MODE 2 = &01

Die Inhalte von A und F gehen verloren.

&BBE4 Hintergrundfarbe für Grafik
(GRA SET PAPER)

Aufgabe/Wirkung: Die Routine setzt das Grafikfenster auf eine definierte Hintergrundfarbe.

Einsprungbedingungen: A muß mit dem gewünschten Farbwert geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Farbangabe in A wird mit einem Maskenwert logisch verknüpft, um eine für den jeweiligen Betriebsmodus gültige Vereinbarung zu gewährleisten.

Die Maskenwerte lauten:

MODE 0 = &0F
MODE 1 = &03
MODE 2 = &01

Die Inhalte von A und F gehen verloren.

&BBE7 Abfrage der Hintergrundfarbe (Grafik)
(GRA GET PAPER)

Aufgabe/Wirkung: Mit der Routine wird die aktuelle Hintergrundfarbe abgefragt und an das aufrufende Programm übergeben.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: A enthält den Farbwert.

Anmerkung: Die Werte der Statusbits gehen verloren.

&BBEA Absolutes Setzen eines Grafikpunktes

Aufgabe/Wirkung: Es wird ein Punkt an einer angegebenen Koordinate gesetzt.

Einsprungbedingungen: DE muß mit der X-Koordinate des Punktes geladen werden. HL muß mit der Y-Koordinate geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und die Werte der Statusbits gehen verloren.

&BBED Relatives Setzen eines Grafikpunktes (GRA PLOT RELATIVE)

Aufgabe/Wirkung: Es wird ein Grafikpunkt gesetzt, dessen Zielkoordinate durch einen Koordinatenoffset definiert wird.

Einsprungbedingungen: DE muß mit dem X-Koordinatenoffset des Punktes geladen werden. HL muß mit dem Y-Koordinatenoffset geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und die Werte der Statusbits gehen verloren.

&BBF0 Abfrage eines Grafikpunktes (GRA TEST ABSOLUTE)

Aufgabe/Wirkung: Die Routine fragt einen in absoluten Koordinaten vorgegebenen Grafikpunkt auf dessen Vorder- bzw. Hintergrundfarbe ab, sofern er innerhalb des Grafikfensters liegt.

Einsprungbedingungen: DE muß die X-Koordinate enthalten.
HL muß die Y-Koordinate enthalten.

Übergabeparameter beim Aussprung: A enthält den Wert der Vorder- oder der Hintergrundfarbe des angesteuerten Punktes.

Anmerkung: Die Inhalte der Register BC, DE, HL und F gehen verloren.

&BBF3 Abfrage eines Grafikpunktes (GRA TEST RELATIVE)

Aufgabe/Wirkung: Die Routine fragt einen durch einen Koordinatenoffset vorgegebenen Grafikpunkt auf dessen Vorder- bzw. Hintergrundfarbe ab, sofern er innerhalb des Grafikfensters liegt.

Einsprungbedingungen: DE muß den X-Koordinatenoffset enthalten.
HL muß den Y-Koordinatenoffset enthalten.

Übergabeparameter beim Aussprung: A enthält den Wert der Vorder- oder der Hintergrundfarbe des angesteuerten Punktes.

Anmerkung: Die Inhalte der Register BC, DE, HL und F gehen verloren.

&BBF6 Grafiklinie absolut (GRA LINE ABSOLUTE)

Aufgabe/Wirkung: Die Routine zeichnet eine gerade Linie zwischen dem aktuellen Ort des Grafikcursors und einem vorgegebenen Zielpunkt, der durch eine absolute Koordinatenangabe definiert ist.

Einsprungbedingungen: DE muß mit der X-Koordinate des Zielpunktes geladen werden. HL muß mit der Y-Koordinate des Zielpunktes geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und F gehen verloren.

&BBF9 Grafiklinie relativ
(GRA LINE RELATIVE)

Aufgabe/Wirkung: Die Routine zeichnet eine gerade Linie zwischen dem aktuellen Ort des Grafikcursors und einem vorgegebenen Zielpunkt, der durch einen Koordinatenoffset bestimmt wird.

Einsprungbedingungen: DE muß mit dem X-Koordinatenoffset des Zielpunktes geladen werden. HL muß mit dem Y-Koordinatenoffset des Zielpunktes geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H, L und F gehen verloren.

&BBFC Textausgabe im Grafikmodus
(GRA WR CHAR)

Aufgabe/Wirkung: Gibt Text an der aktuellen Position des Grafikcursors aus.

Einsprungbedingungen: Der ASCII-Code des Zeichens muß im Akkumulator vorhanden sein.

Übergabeparameter beim Aussprung: keine

Anmerkung: Das Zeichen wird mit dem linken unteren Matrixpunkt an der Grafikcursorposition ausgegeben. Der Grafikcursor wird automatisch auf eine neue Zeichenposition bewegt.

Die Inhalte der Register A, B, C, D, E, H und L gehen verloren. Die Statusbits werden zerstört.

&BC0E Bildschirmmodus
(SCR SET MODE)

Aufgabe/Wirkung: Bildschirmmodus 0, 1 oder 2 setzen.

Einsprungbedingungen: Im Akkumulator muß der entsprechende Modus vereinbart sein.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H und L gehen verloren. Die Statusbits werden zerstört.

&BC14 Bildschirm löschen
(SCR CLEAR)

Aufgabe/Wirkung: Bildschirmspeicher wird geleert.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H und L gehen verloren. Die Statusbits werden zerstört.

&BC32 Farbwert setzen
(SCR SET INK)

Aufgabe/Wirkung: Setzt die für die Vordergrundfarben benötigten Farbwerte.

Einsprungbedingungen: A muß die Farbnummer enthalten. B muß mit der ersten Farbe, C mit der zweiten Farbe für Farbwechsel geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H und L gehen verloren. Die Statusbits werden zerstört. Wenn B und C voneinander abweichen, erfolgt ein periodischer Farbwechsel.

&BC4D Zeile rollen
(SCR HW ROLL)

Aufgabe/Wirkung: Bewegt den gesamten Bildschirminhalt um ein Zeichen nach oben oder unten.

Einsprungbedingungen: Abwärtsrollen: B muß den Wert 0 enthalten. Aufwärtsrollen: B muß einen Wert verschieden von 0 enthalten. A: Hintergrundfarbe für die neue Zeile.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H und L gehen verloren. Die Statusbits werden zerstört.

&BC59 Grafikmodus
(SCR ACCESS)

Aufgabe/Wirkung: Bildschirmmodus für Grafik festlegen.

Einsprungbedingungen: A muß mit dem Schreibmodus geladen werden. Hierbei gilt:
0: Normalmodus. Farbe wird durch INK bestimmt.
1: XOR-Modus. Farbe wird durch Exklusiv-ODER-Verknüpfung mit der alten Farbe gebildet.
2: UND-Modus. Farbe wird durch UND-Verknüpfung mit der alten Farbe gebildet.
3: ODER-Modus. Farbe wird durch ODER-Verknüpfung mit der alten Farbe gebildet.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H und L gehen verloren. Die Statusbits werden zerstört.

&BCA7 SOUND-Kanal zurücksetzen
(SOUND RESET)

Aufgabe/Wirkung: Setzt den Status des Geräuschgeneratorchips zurück.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H und L gehen verloren. Die Statusbits werden zerstört.

&BCAA Ton auf Kanal schalten
(SOUND QUEUE)

Aufgabe/Wirkung: Lädt einen Ton in den SOUND-Puffer.

Einsprungbedingungen: HL muß mit der Adresse eines Tonereignisses geladen werden.

Übergabeparameter beim Aussprung: keine

Anmerkung: Der Inhalt von HL ist zerstört, wenn ein Tonereignis geladen wurde. Das Übertragsbit ist in diesem Fall gesetzt, andernfalls zurückgesetzt. Die Inhalte der Register A, B, C, D, E und IX gehen verloren. Alle übrigen Statusbits werden zerstört.

Bedeutung der SOUND-Register:

Byte 0: Kanal und Synchronisation
Byte 1: Einhüllende der Amplitude
Byte 2: Hüllkurve der Frequenzmodulation
Byte 3,4: Tonhöhe
Byte 5: Rauschperiode
Byte 6: Startamplitude
Byte 7: Dauer oder Wiederholperiode

Bedeutung der Bits von Byte 0:

Bit 0: Kanal A
Bit 1: Kanal B
Bit 2: Kanal C
Bit 3: warte auf Kanal A
Bit 4: warte auf Kanal B
Bit 5: warte auf Kanal C
Bit 6: Haltefunktion
Bit 7: Warteschlange leeren

&BCAD Status der Warteschlange
SOUND CHECK

Aufgabe/Wirkung: Abfrage der Warteschlange (SOUND-Puffer).

Einsprungbedingungen: A muß den Kanal spezifizieren:

Bit 0 gesetzt: Kanal A
Bit 1 gesetzt: Kanal B
Bit 2 gesetzt: Kanal C

Übergabeparameter beim Aussprung:
A: Kanalstatus
Bit 0 bis 2: freie Plätze im SOUND-Puffer
Bit 3: Kanal wartet auf Rendezvous mit A
Bit 4: Kanal wartet auf Rendezvous mit B
Bit 5: Kanal wartet auf Rendezvous mit C
Bit 6: Kanal wird angehalten
Bit 7: Der getestete Kanal ist aktiv

Anmerkung: Die Inhalte der Register B, C, D, E, H und L gehen verloren. Alle Statusbits werden zerstört.

&BCAD SOUND-Puffer auffüllen
(SOUND ARM EVENT)

Aufgabe/Wirkung: Nachladen einer SOUND-Anweisung, wenn ein Platz im SOUND-Puffer frei wird.

Einsprungbedingungen: A muß mit der Kanalkennung geladen werden. Es gilt:

Bit 0: Kanal A
Bit 1: Kanal B
Bit 2: Kanal C

HL muß mit der Adresse geladen werden, unter der das Ton- oder Geräuschereignis vereinbart wurde.

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Register A, B, C, D, E, H und L gehen verloren. Die Statusbits werden zerstört.

&BCD3 Tonfreigabe
(SOUND RELEASE)

Aufgabe/Wirkung: Die mit einem Haltebit gekennzeichneten Ton- oder Geräuschereignisse einzelner Kanäle werden für die Wiedergabe freigegeben.

Einsprungbedingungen: A muß mit der Kanalkennung geladen werden. Es gilt:

Bit 0: Kanal A
Bit 1: Kanal B
Bit 2: Kanal C

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Registerpaare AF, BC, DE, HL und IX gehen verloren.

&BCB6 Tonereignis sperren
(SOUND HOLD)

Aufgabe/Wirkung: Die Tonausgabe wird unterbrochen.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: Übertragsbit gesetzt, wenn bei der Ausführung eine aktuelle Tonausgabe unterbrochen wurde, ansonsten zurückgesetzt.

Anmerkung: Die Inhalte der Register A, B, C, H und L gehen verloren. Die Werte der Statusbits werden zerstört.

Die Tonausgabe wird unmittelbar durch Ausführung der Unterroutinen SOUND CHANNEL, SOUND RELEASE oder CONTINUE SOUND wieder aufgenommen.

&BCB9 Tonausgabe wieder aufnehmen
(CONTINUE SOUND)

Aufgabe/Wirkung: Alle angehaltenen Töne werden zur Ausgabe freigegeben.

Einsprungbedingungen: keine

Übergabeparameter beim Aussprung: keine

Anmerkung: Die Inhalte der Registerpaare AF, BC, DE, HL und IX gehen verloren.

&BCBC Lautstärke-Einhüllende (SOUND AMPL ENVELOPE)

Aufgabe/Wirkung: Es wird eine von 15 verschiedenen vom Anwender frei definierbaren Lautstärkeeinhüllenden aktiviert.

Einsprungbedingungen: A muß mit der Nummer der Hüllkurve geladen werden. HL muß mit der Adresse geladen werden, bei der die Tabelle der Hüllkurvenparameter beginnt. Die Bytes dieser Tabelle sind folgendermaßen organisiert:

Byte 0: Anzahl der Hüllkurvenabschnitte
Bytes 1,2,3: Parameter des ersten Abschnitts.
Bytes 4,5,6: Parameter des zweiten Abschnitts.
Bytes 7,8,9: Parameter des dritten Abschnitts.
Bytes 10,11,12: Parameter des vierten Abschnitts.
Bytes 13,14,15: Parameter des fünften Abschnitts.

Die Bytes der einzelnen Abschnitte müssen folgendermaßen belegt werden:

Byte 0: Schrittzahl
Byte 1: Schrittweite
Byte 2: Pausendauer

Übergabeparameter beim Aussprung: HL enthält die Tabellenadresse + 16, wenn die vereinbarte Hüllkurve korrekt ist. Im anderen Fall bleibt der Inhalt von HL unverändert. Die Inhalte der Register A, B und C bleiben unverändert, wenn die Hüllkurvenvereinbarung nicht korrekt ist. Andernfalls gehen die Inhalte verloren. Das Übertragsbit ist gesetzt, wenn die Einhüllende korrekt vereinbart wurde, andernfalls zurückgesetzt.

Anmerkung: Die Inhalte von D und E gehen in jedem Fall verloren. Die Werte der oben nicht genannten Statusbits werden zerstört.

&BCBF Frequenzmodulation (SOUND TONE ENVELOPE)

Aufgabe/Wirkung: Es werden die Parameter für eine Frequenzmodulierende vereinbart.

Einsprungbedingungen: A muß mit der Hüllkurvennummer geladen werden. HL muß die Startadresse für eine Parametertabelle enthalten.

Übergabeparameter beim Aussprung: HL enthält die Tabellenadresse + 16, wenn die vereinbarte Hüllkurve korrekt ist. Im anderen Fall bleibt der Inhalt von HL unverändert. Die Inhalte der Register A, B und C bleiben unverändert, wenn die Hüllkurvenvereinbarung nicht korrekt ist. Andernfalls gehen die Inhalte verloren. Das Übertragsbit ist gesetzt, wenn die Einhüllende korrekt vereinbart wurde, andernfalls zurückgesetzt.

Anmerkung: Die Inhalte von D und E gehen in jedem Fall verloren. Die Werte der oben nicht genannten Statusbits werden zerstört.

&BCC2 Adresse einer Lautstärkeeinhüllenden (SOUND A ADDRESS)

Aufgabe/Wirkung: Aufsuchen der Hüllkurvenparameter für eine angegebene Hüllkurvennummer.

Einsprungbedingungen: A muß mit der Hüllkurvennummer (Wert zwischen 1 und 15) geladen werden.

Übergabeparameter beim Aussprung: BC enthält die Anzahl von Bytes für die angegebene Hüllkurve, wenn diese korrekt vereinbart wurde. Andernfalls wird der Inhalt von BC nicht beeinflußt.

HL enthält die Adresse der Tabelle, wenn die Hüllkurve richtig vereinbart wurde. Im anderen Fall wird HL nicht beeinflußt.

Das Übertragsbit wird bei korrekter Einhüllender gesetzt, sonst zurückgesetzt.

Anmerkung: Der Inhalt von A geht verloren. Alle anderen Statusbitwerte werden zerstört.

&BCC5 Adresse einer Frequenzmodulierenden (SOUND T ADDRESS)

Aufgabe/Wirkung: Aufsuchen der Hüllkurvenparameter für eine frequenzmodulierende Einhüllende.

Einsprungbedingungen: A muß mit der Hüllkurvennummer (Wert zwischen 1 und 15) geladen werden.

Übergabeparameter beim Aussprung: BC enthält die Anzahl von Bytes für die angegebene Hüllkurve, wenn diese korrekt vereinbart wurde. Andernfalls wird der Inhalt von BC nicht beeinflußt.

HL enthält die Adresse der Tabelle, wenn die Hüllkurve richtig vereinbart wurde. Im anderen Fall wird HL nicht beeinflußt.

Das Übertragsbit wird bei korrekter Einhüllender gesetzt, sonst zurückgesetzt.

Anmerkung: Der Inhalt von A geht verloren. Alle anderen Statusbitwerte werden zerstört.

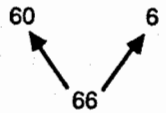
Ausführliche Hinweise über diese und noch eine Fülle anderer ROM-Routinen finden Sie in der Schneider-Publikation: "Das komplette CPC 464 Betriebssystem, Firmware-Handbuch", SOFT 258.

Anhang E

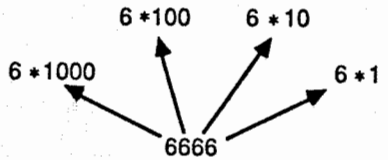
Duale, hexadezimale und BCD-Zahlen

Dezimale Zahlendarstellung

Moderne Rechensysteme überall in der Welt machen vom dezimalen Zahlensystem Gebrauch. Es wurde entwickelt, um über 10 hinaus zählen und mit Zahlen kleiner als 1 rechnen zu können. In diesem Zahlensystem nimmt der Wert einer Ziffer zu, je weiter links sie innerhalb einer Zahlenangabe steht. In der Zahl 66 beispielsweise ist die erste 6 zehnmal so viel wert wie die zweite, d.h.



Bei größeren Zahlen ist das genauso. Jede um eine Stelle weiter links stehende Ziffer ist um ein Vielfaches von 10 mehr wert als die vorherige, also

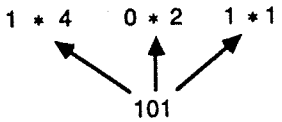


Ein System, in dem die Ziffern je nach ihrer Stellung innerhalb einer Zahl ein unterschiedliches Gewicht besitzen, wird Stellenwert- oder Positionssystem genannt. Beim Dezimalsystem steigen die Gewichte der einzelnen Ziffern mit Vielfachen von 10. Die Zahl 10 wird als Basis des Zahlensystems bezeichnet. Andere Stellenwertsysteme benutzen andere Basiszahlen, gehorchen aber ansonsten dem gleichen Bildungsgesetz. Jede weiter links stehende Ziffer besitzt immer ein um ein Vielfaches der Basiszahl höheres Gewicht.

Duale Zahlendarstellung

Der Computer mit seinen elektronischen Funktionselementen kann nur zwei Zustände unterscheiden, nämlich "Ein" und "Aus". Diese beiden Zustände werden häufig auch als 1 und 0 dargestellt. Computer machen deshalb vom dualen Zahlensystem Gebrauch, dessen Basis die Zahl 2 ist. Jede duale Zahl besteht somit aus den Ziffern 0 und 1 oder - aus elektrischer Sicht - den Zuständen Ein und Aus, die in Form von Spannungszuständen 0 Volt und einigen Volt angegeben

werden können. Um über 1 hinauszählen zu können, muß auch hier ein Stellenwertsystem aufgebaut werden, dessen Basis - wie gesagt - die Zahl 2 ist. Die Dualzahl 101 kann somit wie folgt zerlegt werden:



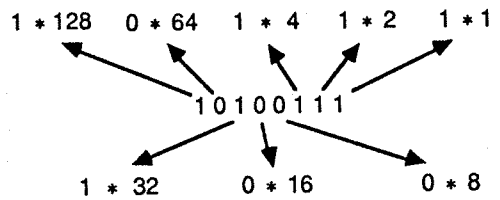
Der Dualzahl 101 entspricht somit die Dezimalzahl 4+0+1=5. Damit es bei der Zahlendarstellung nicht zu Verwechslungen kommt, ist es in Zweifelsfällen üblich, die Basis des Zahlensystems in Form eines Index der Zahl anzuhängen, beispielsweise

101_{10} = Einhundertundeins zur Basis Zehn
101_2 = Eins-Null-Eins zur Basis Zwei (dezimal fünf).

Die derzeit weitverbreiteten Heimcomputer der 8-bit-Klasse arbeiten mit Registern bzw. Speicherelementen mit einer Wortbreite von 8 bit. Sie können daher Dualzahlen bis zum Wert 11111111, d.h. dezimal 255 darstellen.

```
128 +64 +32 +16 + 8 +4 +2 + 1  = 255(10)
 1   1   1   1   1  1  1  1
```

Wir wollen als Beispiel noch eine andere Dualzahl in eine Dezimalzahl umsetzen:



Somit folgt:

```
10100111  = 1*128+0*64+1*32+0*16+0*8+1*4+1*2+1*1
          = 128 + 32 + 4 + 2 + 1
          = 167(10)
```

Zur Überprüfung Ihres Wissensstandes sollten Sie nun die folgende Übungsaufgabe lösen.

Übungsaufgabe AE.1

Berechnen Sie die dezimalen Werte der nachfolgend angegebenen Dualzahlen:

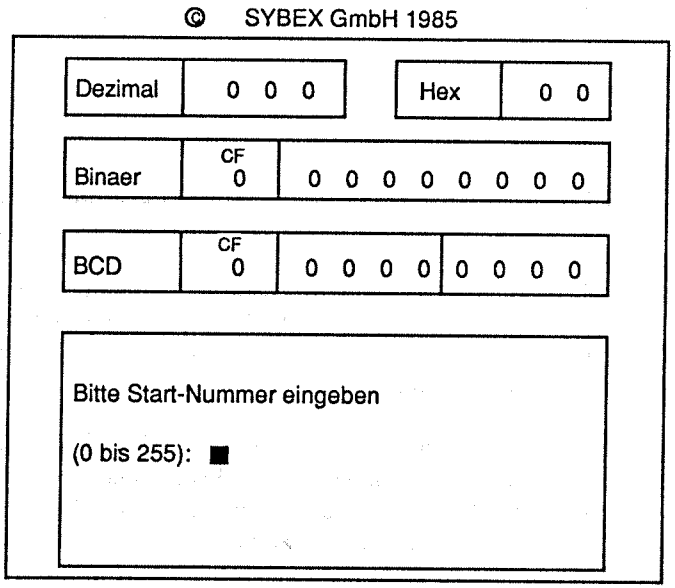
a) 00000011
b) 00000100
c) 10000000
d) 10000011
e) 10110111
f) 01110011

Antworten finden Sie im Anhang G.

Sie können die Umwandlung von Zahlen vom Dezimal- in das Dualsystem mit Hilfe des Programms BIN/HEX auf der Kassette üben. Starten Sie das Programm durch Eingabe des Befehls

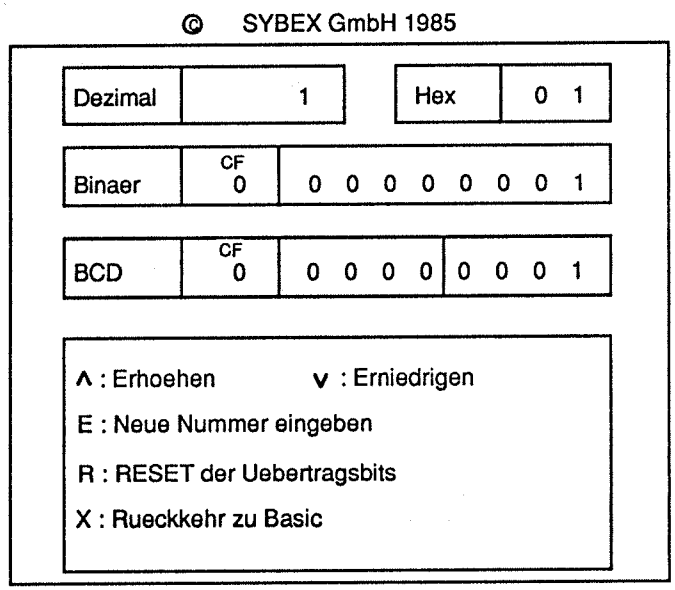
```
RUN "BIN/HEX"
```

Das Programm wird automatisch geladen und gestartet. Sie sehen dann das folgende Bild:



Kümmern Sie sich bei der Anzeige auf dem Bildschirm nicht um die mit Hex bzw. BCD bezeichneten Felder. Wir kommen gleich noch auf sie zurück. Wir interessieren uns an dieser Stelle nur für die mit Dezimal und Binaer gekennzeichneten Felder.

Zunächst erwartet das Programm von Ihnen die Eingabe einer Zahl im Bereich zwischen 0 und 255. Geben Sie eine 1 ein, und betätigen Sie anschließend zur Bestätigung die ENTER-Taste. Auf dem Bildschirm ist anschließend folgendes zu sehen:



In den jeweiligen Feldern wird die 1 in unterschiedlicher Notation angezeigt. Das Kommandofeld macht Sie darauf aufmerksam, daß Sie den aktuellen Wert um 1 erhöhen oder auch erniedrigen können, indem Sie die mit einem Aufwärts- bzw. einem Abwärtspfeil gekennzeichneten Tasten des Cursorblocks benutzen.

Wenn Sie die Taste mit dem aufwärts weisenden Pfeil betätigen, können Sie unmittelbar sehen, wie die Dualzahl jeweils um 1 erhöht wird. Wenn Sie die Taste E drücken, können Sie anschließend eine neue Startzahl eingeben. Geben Sie zur Probe 15 ein. Im Feld für die Dualzahl sollte nun 00001111 stehen. Erhöhen Sie diese Zahl um 1. Die vier niederwertigen Bits werden 0, und das fünfte Bit wird auf 1 gesetzt. Um das zu verstehen, brauchen Sie nur einmal die duale Additionsaufgabe


```
  1111
+    1
```

zu lösen. Die Addition zweier Einsen führt zu 2, also zu 0 mit einem Übertrag 1. Dieser Übertrag führt bei der nächstfolgenden 1 wieder zu einer 0 mit einem Übertrag usw.

Wenn das Register voll ist, also eine 11111111 enthält, führt die Addition einer weiteren 1 zu einem Rücksetzen des gesamten Registerinhaltes auf 0. Die nächstfolgende 1 mit dem Gewicht 256 ist verloren. Beim Z80 steht diese 1 allerdings im Übertragsbit des Statusregisters, so daß die Information noch nicht ganz verloren ist. Dieses Bit wird in der Bildschirmanzeige im CF-Feld dargestellt.

Wenn Sie sich von dem Verhalten des Übertragsbits überzeugen wollen, betätigen Sie die Taste E, geben die Zahl 250 (dezimal) ein und zählen Schritt für Schritt aufwärts.

Wenn Sie über 255 hinauszählen, wird das Übertragsbit gesetzt, und der Inhalt des Zahlenfeldes wird gleich 00000000. Diese Zwischenspeicherung ist aber nur zeitlich begrenzt, bis eine andere Operation das Bit zurücksetzt. Denken Sie bitte daran!

Das Konvertierungsprogramm nimmt auch Zahlen in dualer Notation an. Sie müssen bei der Eingabe derartiger Zahlen allerdings die Zeichenfolge &X voranstellen. Um also die Dualzahl 101010 (42 dezimal) einzugeben, tippen Sie

```
&X101010
```

und bestätigen die Eingabe dann wieder durch die ENTER-Taste. In allen Feldern wird die Dezimalzahl 42 in den unterschiedlichen Notationen angegeben.

Zur Überprüfung, ob Sie mit der dualen Schreibweise vertraut sind, wandeln Sie bitte auf dem Papier die nachfolgend angegebenen Dualzahlen in Dezimalzahlen um, und überprüfen Sie anschließend Ihre Rechnung mit Hilfe des Programms.

Übungsaufgabe AE.2

a) 11111
b) 101001
c) 10111101
d) 10001000

Antworten finden Sie im Anhang G.

Belassen Sie das BIN/HEX-Programm im Computer. Sie werden es sogleich wieder benötigen.

Die Nullen und Einsen sind zwar für den Computer zweckmäßig, für den Programmierer dagegen sind sie weniger angenehm zu verarbeiten und zu handhaben. Die dezimale Notation als Alternative hilft da allerdings auch nicht weiter, denn abgesehen von 1_{10} und 1_2 gibt es keine nennenswerte Übereinstimmung. Man könnte auf die Idee kommen, die ganze 8-bit-Zahl als eine Ziffer in einem Stellenwertsystem mit der Basis 256 aufzufassen. Was glauben Sie, spricht wohl dagegen?

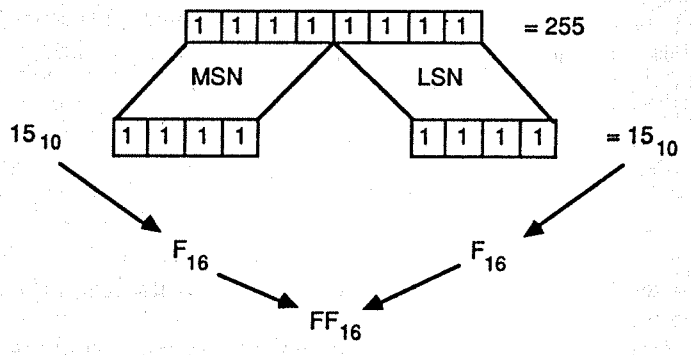
Nach einigem Nachdenken werden Sie rasch auf die richtige Antwort kommen. Denken Sie nochmals an das Dezimalsystem mit der Basis 10. Es besitzt 10 verschiedene Ziffern von 0 bis 9. Beim dualen System benötigen wir entsprechend 2 Ziffern (0 und 1). Ein System zur Basis 256 würde also 256 verschiedene Ziffern erfordern!

Hexadezimale Zahlendarstellung

Einen Kompromiß stellt ein Zahlensystem dar, das mit der Basis 16 arbeitet. Es wird Hexadezimalsystem genannt. Jeweils vier Bits eines Datenwortes können so zu einer Hexadezimalzahl zusammengefaßt werden. Neben den Ziffern 0 bis 9 sind noch zusätzlich 6 weitere Zahlensymbole notwendig, um 16 verschiedene Ziffern darstellen zu können. Es werden hierfür die Buchstaben A bis F verwendet. Die folgende Abbildung zeigt zur Erläuterung eine Gegenüberstellung in dezimaler, dualer und hexadezimaler Notation.

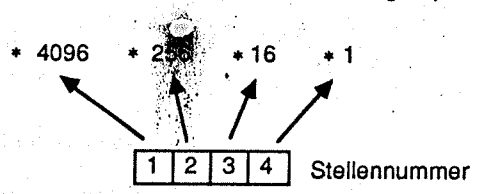
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

In hexadezimaler Schreibweise können 8-bit-Datenworte durch zwei Hexadezimalziffern dargestellt werden. Eine repräsentiert den höherwertigen, die andere den niederwertigen Teil eines Bytes. Diese Halbbytes werden auch als Nibbles bezeichnet. Ein Byte besteht somit aus einem höherwertigen (MSN) und einem niederwertigen (LSN) Nibble, wie dies in der folgenden Abbildung gezeigt ist.

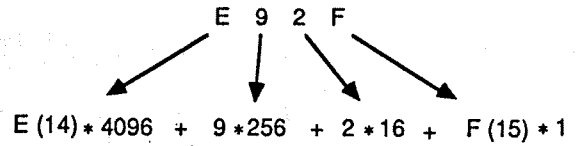


Der Hauptvorteil der Zahlendarstellung im Hexadezimalsystem ist die Kompatibilität mit dem Dualsystem. Jede 8-bit-Dualzahl kann in Form von zwei Hexadezimalziffern dargestellt werden.

Alle eingegebenen Zahlen werden in dem Feld rechts oben auf dem Bildschirm in hexadezimaler Notation angezeigt. Um sich mit dem Verhalten hexadezimaler Ziffern vertraut zu machen, sollten Sie einfach auf die nunmehr bereits bekannte Weise einen Aufwärtszählvorgang durchführen. Bis zur Ziffer 9 stimmen die hexadezimalen Ziffern mit den dezimalen überein. Darüber hinaus entsprechen die hexadezimalen Ziffern A bis F den dezimalen Zahlen 10 bis 15. Über 15 hinausgehende Zahlen werden durch stellengerechte Aneinanderreihung der hexadezimalen Ziffern gebildet. So entspricht hexadezimal FF der dezimalen Zahl 255. Die Basis dieses Stellenwertsystems ist natürlich die Zahl 16 (sehen Sie sich dazu bitte auch die folgende Abbildung an).



Nun zeigen wir die Umwandlung der Hexadezimalzahl E92F in die entsprechende Dezimalzahl.



Da Sie sich jetzt auch mit der hexadezimalen Zahlendarstellung auskennen, sollten Sie die nachfolgend angegebenen Übungsaufgaben lösen.

Übungsaufgabe AE.3

Wandeln Sie die folgenden Hexadezimalzahlen in die entsprechenden Dezimalzahlen um:

a) 09_{16}
b) 13_{16}
c) $A5_{16}$
d) AE_{16}
e) $0E_{16}$
f) $1A_{16}$
g) EA_{16}

Führen Sie die Berechnungen zunächst auf dem Papier aus. Sie sollten erst anschließend die Lösungen mit dem Programm überprüfen. Es nimmt neben Dezimal- bzw. Dualzahlen auch Hexadezimalzahlen bei der Eingabe an. Sie müssen eindeutig durch Voranstellen des Zeichens & gekennzeichnet sein. Wenn Sie also &FF eingeben, wird das dezimale Äquivalent 255 angezeigt.

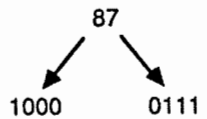
Lassen Sie das Programm noch etwas im Speicher. Sie werden es noch einmal benötigen.

BCD-Zahlen

In der Computertechnik wird neben der dezimalen, der dualen und der hexadezimalen Zahlendarstellung noch von einer weiteren Schreibweise Gebrauch gemacht, die unter dem Namen BCD-Schreibweise bekannt ist. BCD ist eine Abkürzung für "binary coded decimal". Wie der Name schon vermuten läßt, handelt es sich um eine gemischte Art der Zahlendarstellung. Üblicherweise wird sie bei der Darstellung von digitalen Informationen in einer benutzerfreundlichen Form (Digitaluhr) oder bei der Verarbeitung von Zahlen mit höchster Stellengenauigkeit verwendet.

Für die Darstellung einer Dezimalzahl in BCD-Form wird für jede Ziffer ein Datenwort mit vier bit Wortbreite (also ein Nibble) benutzt.

Die Zahl 87_{10} beispielsweise wird dann so verschlüsselt:



Die zugehörige BCD-Zahl lautet somit 1000 0111 oder - in ein Byte zusammengepackt - 10000111. Um eine BCD-Zahl von einer Dualzahl zu unterscheiden, wird der Zahl häufig das Attribut (BCD) nachgestellt: 10000111 (BCD).

Da die höchste Ziffer im dezimalen Zahlensystem die 9 ist, werden vier Bits für die Verschlüsselung benötigt. Die sechs über 1001 hinausgehenden Codeworte (1010 bis 1111) werden nicht benutzt. Sie werden als Pseudotetraden bezeichnet. Vom Standpunkt des Speicherbedarfs her gesehen ist die BCD-Darstellung also nicht ökonomisch.

Nachfolgend noch einmal das typische Bildungsgesetz anhand einer Neunerüberschreitung:

8_{10} = 0000 1000 (BCD)
9_{10} = 0000 1001 (BCD)
10_{10} = 0001 0000 (BCD)
11_{10} = 0001 0001 (BCD)

Die eingegebenen Zahlen werden im Feld BCD in der zuvor erläuterten Form dargestellt. Betätigen Sie die Taste E, geben Sie dann eine 1 ein, und zählen Sie anschließend aufwärts. Bis zur Ziffer 9 stimmen die Angaben mit denen im dualen Feld überein. Anschließend jedoch erfolgt bei der BCD-Darstellung der Übertrag zum nächsten Nibble.

Wenn der Wert der eingegebenen Zahl wächst, wird die wenig ökonomische Natur der BCD-Darstellung immer deutlicher. Wenn Sie nun beispielsweise als neuen Startwert 95 eingeben und von da aus aufwärtszählen, werden Sie sehen, daß beim Übergang von 99 zu 100 (dezimal) ein Problem auftritt. Der Akkumulator weist einen falschen Wert auf. Das Programm berücksichtigt diese Tatsache, indem es eine Fehlermeldung ausgibt. Wenn Sie nun auf 99 (dezimal) zurückzählen, wird zwar wieder der richtige BCD-Wert angezeigt, das Übertragsbit jedoch bleibt gesetzt. (Wenn Sie wollen, können Sie beide Übertragsbits durch Betätigen der Taste R zurücksetzen. Sobald jedoch ein Wert über 99 (dezimal) erreicht wird, wird das Übertragsbit im BCD-Feld wieder gesetzt.)

Wie bereits an anderer Stelle erwähnt, dient das Übertragsbit nur vorübergehend als Zwischenspeicher. Es muß unmittelbar weiterverarbeitet werden, sonst ist die Information verloren. Ein Übertrag wird bei der BCD-Darstellung bei Überschreiten von 99, bei der dezimalen Darstellung natürlich bei Überschreiten von 999, bei der dualen Darstellung bei der Überschreitung von 1111 1111 und bei der hexadezimalen Darstellung bei Überschreitung von FF auftreten.

Festigen Sie Ihr bisher erworbenes Wissen durch die Bearbeitung folgender Übungsaufgaben:

Übungsaufgabe AE.4

Wandeln Sie die nachfolgend angegebenen Dezimalzahlen in BCD-Zahlen um:

a) 4
b) 53
c) 10
d) 102
e) 77
f) 953
g) 97
h) 2579

Übungsaufgabe AE.5

Wandeln Sie die folgenden BCD-Zahlen in Dezimalzahlen um:

a) 0000 0001
b) 0000 1001
c) 0001 0101
d) 0010 0000
e) 0100 1001
f) 0010 0011
g) 1001 0111
h) 1000 1000

Leider können in das Programm keine BCD-Zahlen unmittelbar eingegeben werden. Eine Ausweichmöglichkeit besteht darin, jedes Nibble getrennt einzugeben.

Nach den bisherigen Erläuterungen sollten Sie nun in der Lage sein, mit Zahlen in den unterschiedlichen Schreibweisen umzugehen. Das Ende des Anhangs E ist damit erreicht.

Anhang F

Der Assembler

Dieser Teil des Anhangs beschreibt sowohl den Assembler als auch den Editor. Sie sollten sich die nachfolgenden Ausführungen gut durchlesen und während der Durcharbeitung des Buchs immer wieder hier nachschlagen.

Nach dem Start des Programms ASSEMBLER wird folgendes Menü auf dem Bildschirm ausgegeben:

```
CPC 464 ASSEMBLERKURS      ©      1985 Sybex GmbH
-------------------------------------------------

   <A>   Assemblieren
   <B>   Zahlenumrechnung
   <C>   Programm ausfuehren
   <D>   Zeile loeschen
   <F>   Zeilen listen
   <I>   Zeile einfuegen
   <L>   Datei laden
   <N>   Renumerieren
   <O>   Arithmetische Umrechnung
   <P>   Datei drucken
   <Q>   Neues Programm eingeben
   <R>   Zeile ersetzen
   <S>   Datei speichern
   <X>   Zurueck zu Basic

-------------------------------------------------

Ihre Wahl>
```

Die Kommandos werden nachfolgend im einzelnen erläutert.

Der Einfügemodus (Kommando I)

Das Kommando I aktiviert den Einfügemodus. Er dient dazu, um einen Programmtext zu erstellen oder neuen Text in bereits vorhandenen einzufügen. Sie können sowohl die Startzeilennummer als auch die Schrittweite nach der Meldung

Startzeilennummer & Schrittweite

angeben. Die zuerst eingegebene Zahl wird als Startzeilennummer, die zweite als Schrittweite angesehen. Die beiden Zahlen müssen durch einen Bindestrich (-) voneinander getrennt werden. Die folgende Eingabesequenz führt zu einer Startzeilennummer 10 und einer Zeilennummer-Erhöhung in Schritten von 10. Die zweite Zeilennummer lautet also 20.

Beispiel:

Ihre Wahl> I
Startzeilennummer & Schrittweite: 10-10 [ENTER]

Bitte beachten Sie, daß Sie das eingegebene I nicht sehen. In der Sequenz zuvor wurde es nur gezeigt, um dessen Eingabe anzudeuten. Im weiteren wollen wir außerdem [ENTER] verwenden, um anzudeuten, daß die Eingabe durch Betätigen der ENTER-Taste abzuschließen ist.

Wenn keine Schrittweite angegeben wird, setzt der Editor diese auf den Wert 10. Um den Einfügemodus zu verlassen, können Sie entweder vor der Eingabe der Zeilennummer die Taste M oder nachher die Taste mit dem "Klammeraffen" @ als erstes Zeichen einer Zeile betätigen. Der Editor wird dann in den Kommandomodus zurückkehren und wieder das Menü zeigen. Wenn beispielsweise der Programmtext den Assemblerbefehl RET in Zeile 120 enthält, geben Sie als erstes in Zeile 130 ein @ ein, um den Einfügemodus zu verlassen.

Das sieht dann so aus

120 RET [ENTER]
130 @ [ENTER]

Ausgabe des Programmlistings

Mit dem Befehl "Zeilen listen" (F-Kommando) können Sie sich das ganze Assemblerprogramm oder auch nur einen Teil davon auf dem Bildschirm ausgeben lassen. Nach Betätigen der Taste F erscheint die Meldung:

Start-Zeilennummer:

Wenn Sie das ganze Programm sehen wollen, brauchen Sie nur die ENTER-Taste zu betätigen. Die Ausgabe hält immer dann an, wenn das Ausgabefenster gefüllt ist. Drücken Sie irgendeine Taste, um die Ausgabe fortzuführen. Wenn Sie die Ausgabe erst ab einer speziellen Zeilennummer wünschen, geben Sie diese unmittelbar auf die oben angegebene Meldung hin ein. Während der Ausgabe können Sie das Listing zu jeder Zeit mit der Leertaste anhalten. Betätigen irgendeiner beliebigen Taste führt dann zur Fortsetzung der Ausgabe. Drücken

von M während der Ausgabe bricht den gesamten Ausgabevorgang ab. Auf die Meldung

<<<Taste druecken >>>

hin führt die Betätigung einer beliebigen Taste zur Rückkehr ins Menü und zum Kommandomodus.

Zeile ersetzen

Nach Betätigen der Taste R im Kommandomodus können Sie gezielt Zeilen eines bereits erstellten Programmtextes ersetzen. Nehmen Sie einmal an, im Speicher stünde das folgende Programm

```
10  LD   A,65
20  IN   C A
30  CALL &BB5A
40  RET
```

In der Zeile 20 sollte eigentlich INC A stehen.

Geben Sie zur Änderung ein:

Ihre Wahl> R
Zeilennummer eingeben: 20
20 INC A [ENTER]

Nach der Eingabe kehrt der Editor wieder zum Hauptmenü zurück. Bitte beachten Sie, daß es nicht möglich ist, mehr als eine Zeile auf einmal zu ersetzen.

Zeile löschen

Mit dem Kommando D können Sie eine oder auch mehrere aufeinanderfolgende Zeilen löschen. Die Syntax entspricht genau derjenigen für den Einfügebefehl.

Beispiel:

Ihre Wahl> D
Zeilennummer eingeben: 20 [ENTER]

Mit dieser Eingabesequenz wird die Zeile 20 gelöscht. Die folgende Eingabe

Ihre Wahl> D
Zeilennummer eingeben: 20-200

dagegen löscht die Zeilen 20 bis 200 einschließlich.

Wenn Sie das ganze Programm löschen wollen, geben Sie Zeilennummern ein, die außerhalb des verwendeten Bereichs liegen. Wenn ein Programm beispielsweise die Zeilennummern 10 bis 320 benutzt, können Sie es durch

Zeilennummer eingeben: 1-400 [ENTER]

vollständig löschen.

Programm assemblieren

Wenn Sie den Assemblerprogrammtext erstellt haben, müssen Sie diesen Quellcode in den Maschinencode übersetzen. Hierzu muß der Assembler aufgerufen werden. Dies geschieht durch das A-Kommando.

Auf dem Bildschirm wird anschließend die Meldung:

Moeglichkeiten:

1. Keine Ausgabe.
2. Ausgabe auf den Bildschirm.
3. Ausgabe auf den Drucker.
4. Ausgabe auf beides.

Ihre Wahl:

ausgegeben.

1. Bei Wahl der Ziffer 1 erfolgt keine Ausgabe des Assemblerlistings auf dem Bildschirm. Dies ist die schnellste Methode der Übersetzung. Falls Fehler im Programm enthalten sind, werden diese allerdings auf dem Bildschirm angezeigt.

2. Bei Wahl der Ziffer 2 wird ein Assemblerlisting auf dem Bildschirm ausgegeben.

3. Bei Wahl der Ziffer 3 wird das Listing auf einem angeschlossenen Drucker ausgegeben.

4. Bei Wahl der Ziffer 4 erfolgt die Ausgabe sowohl auf dem Drucker als auch auf dem Bildschirm.

Wenn während des Übersetzungsvorgangs ein Fehler entdeckt wird, erfolgt eine Fehlermeldung und die Rückkehr zum Editor.

Rückkehr zum BASIC-Interpreter

Mit dem X-Kommando kehrt der Assembler/Editor wieder in den normalen Interpretermodus zurück. Der Bildschirm wird mit einem MODE 1-Kommando gelöscht.

Der Assembler/Editor ist eine Mischung aus einem BASIC- und einem Maschinenprogramm. Das kleine BASIC-"Treiberprogramm" benutzt die BASIC-Zeilen ab 63000 aufwärts. Um von der BASIC-Ebene wieder zum Editor/Assembler zurückzukehren, betätigen Sie bitte die Taste mit dem Dezimalpunkt innerhalb des numerischen Tastenblocks.

Der Aufruf eines Maschinenprogramms

Mit dem C-Kommando können Sie Maschinenprogramme, die unter Benutzung des ENT-Befehls assembliert wurden, unmittelbar von der Ebene des Assembler/Editors aus aufrufen. Jeder Aufruf muß die Startadresse des jeweiligen Maschinenprogramms enthalten. Wenn Sie nicht mit dem ENT-Befehl arbeiten, d.h. den Programmbeginn mit dem ORG-Befehl vereinbaren, müssen Sie einen Aufruf von der BASIC-Ebene aus mittels des CALL-Befehls durchführen. Das C-Kommando setzt also voraus, daß die erste Programmzeile des Assemblerprogramms aus einem ENT-Befehl besteht. Nach dem C-Kommando wird der Bildschirm gelöscht und das Maschinenprogramm gestartet.

Angenommen, die erste Zeile des Assemblerprogramms lautet

10 ORG 30000

dann wird das erste Byte des Maschinenprogramms unter dieser Adresse (30000) im Speicher abgelegt. Um das Programm starten zu können, verlassen Sie bitte den Editor/Assembler mit dem X-Kommando, und geben Sie anschließend den Befehl

CALL 30000 [ENTER]

ein. Tragen Sie bitte dafür Sorge, daß das Maschinenprogramm fehlerfrei ist. Es könnte sonst zu einem Systemabsturz kommen.

Ein bereits auf einem Datenträger (Kassette oder Diskette) abgespeichertes Maschinenprogramm kann jederzeit von der BASIC-Ebene aus in den Speicher geladen und anschließend mit einem CALL-Befehl gestartet werden. Der Ladebefehl ist genau der gleiche wie für BASIC-Programme. Zusätzlich ist nur diejenige Adresse anzugeben, ab der das Programm im Speicher abgelegt werden soll.

Um beispielsweise das Maschinenprogramm TEST, das mit dem Namen TEST-B abgespeichert wurde, ab der Adresse 30000 in den Speicher zu laden, geben Sie einfach

LOAD "Test-B",30000 [ENTER]

ein. Anschließend starten Sie das Programm mit

CALL 30000 [ENTER]

Assemblerprogramme, also Textdateien, können ebenfalls von der Ebene des Editor/Assemblers aus auf Massenspeichern gesichert oder von diesen wieder in den Textspeicher geladen werden. Dies gilt auch für assemblierte Programme. Dies erweist sich als sehr nützlich, da Maschinenprogramme auf diese Weise völlig unabhängig vom Editor/Assembler aufgerufen und gestartet werden können.

Sicherung von Programmen

Mit dem S-Befehl können sowohl Text- als auch Binärdateien auf einem externen Massenspeicher gesichert werden.

Abspeichern einer Textdatei

Wie bei BASIC-Programmen auch wird einer Textdatei zunächst ein Name zugeordnet. Sie werden bei dem Aufruf des Sicherungsbefehls automatisch nach dem Namen gefragt:

Ihre Wahl> S
Name: TEST [ENTER]

Wenn Sie mit einem Kassettenrecorder arbeiten, erscheint anschließend die bekannte Meldung:

Press REC and PLAY then any key:

Bei Diskettenbetrieb erfolgt der Sicherungsvorgang unmittelbar nach Bestätigung der Namenseingabe durch die ENTER-Taste. Verfahren Sie in beiden Fällen genauso, wie Sie es von der Sicherung von BASIC-Programmen her gewohnt sind.

Abspeichern von Binärdateien

Sie können auch eine Binärdatei auf einem Massenspeicher sichern, die Sie gerade bei einem Übersetzungsvorgang erzeugt haben.

Binärdateien müssen Sie durch eine spezielle Kennung im Namen kennzeichnen. Diese Kennung besteht aus einem B, das durch einen Bindestrich mit dem Namen verknüpft wird. Wenn Sie beispielsweise eine Binärdatei mit dem Namen TEST abspeichern wollen, dann lautet die Befehlssequenz:

Ihre Wahl>
Name: TEST-B [ENTER]

Laden von Textdateien

Mit dem L-Kommando (Load) können Sie von der Kommandoebene aus eine Textdatei von einem Massenspeicher in den Speicher laden. Binärdateien können auf diese Weise nicht geladen werden. Die Befehlssequenz lautet:

Ihre Wahl> L
Name: TEST [ENTER]

Bei Kassettenbetrieb wird dann die obligatorische Meldung:

Press PLAY then any key:

ausgegeben. Bei Diskettenbetrieb wird der Ladevorgang sofort gestartet.

Ausgabe auf dem Drucker

Mit dem P-Kommando kann ein im Textspeicher enthaltenes Assemblerprogramm auf einem angeschlossenen Drucker ausgegeben werden. Die Syntax entspricht der des L-Kommandos.

Zeilen umnumerieren

Mit dem N-Kommando können die Zeilennummern eines Assemblerprogrammtextes geändert werden. Die Schrittweite kann angegeben werden.

Zahlenkonvertierung

Während der Programmerstellungsphase ist es oft ganz nützlich, wenn man Zahlenwerte von einem Zahlensystem in ein anderes umsetzen kann. Der Editor/Assembler ruft nach Eingabe des B-Befehls ein entsprechendes Unterprogramm auf. Es akzeptiert Zahleneingaben zwischen 0 und 65535. Es können Dezimalzahlen, Dualzahlen und Hexadezimalzahlen konvertiert werden.

Zur Unterscheidung zwischen ihnen werden die Standardattribute verwendet:

Basis	Attribut
dezimal	keins
hexadezimal	&
dual	&X

Jede eingegebene Zahl wird in die anderen Schreibweisen überführt. Die Ausgabe erfolgt in einem eigenen Textfenster. Um beispielsweise die Dezimalzahl 275 in dualer und hexadezimaler Notation zu ermitteln, geben Sie nach der Meldung:

Zu konvertierende Nummer: 275 [ENTER]

ein. Es wird die Zahl anschließend in allen drei Notationen ausgegeben.

Arithmetische Berechnungen

Nach dem O-Kommando wird ein Unterprogramm zu Durchführung arithmetischer Operationen aufgerufen. Es können Zahlenwerte in allen Schreibweisen addiert oder voneinander subtrahiert werden.

Beispiel:

Ihre Wahl> O
Erste Nummer? 200 [ENTER]
Zweite Nummer? &FF [ENTER]
A: Additieren S: Subtrahieren? A [ENTER]

Die Ausgabe lautet dann:

Binaer = &X11100011
Dezimal = 445
Hexadezimal = &1C7

Pseudobefehle des Assemblers

Neben dem vollständigen Befehlssatz des Z80 akzeptiert der Assembler noch spezielle Pseudobefehle. Im Gegensatz zu einem Maschinenbefehl wendet sich ein Pseudobefehl ausschließlich an den Assembler und nicht an den Mikroprozessor.

Pseudobefehle dienen der Erhöhung der Lesbarkeit von Programmen. Sie erleichtern darüber hinaus den Programmentwurf maßgeblich.

Der erste Pseudobefehl sollte Ihnen zwischenzeitlich bereits sehr vertraut sein:

ENT sorgt dafür, daß der Objektcode unmittelbar hinter dem Quellcode abgelegt wird, so daß das Maschinenprogramm von der Editor/Assemblerebene aus aufgerufen werden kann.

Bei der Entwicklung von Programmen, die einen größeren Speicherraum für die Ablage von Daten erfordern, als durch die Register oder den Stapel zur Verfügung gestellt wird, müssen Speicherplätze für die Datenaufnahme reserviert werden. Im Idealfall sollte dieser Speicherbereich in der Nähe des Programms liegen. Da im allgemeinen die exakte Länge eines Programms von vornherein nicht bekannt ist, gestaltet sich die Angabe einer Startadresse für einen an das Programm angehängten Datenblock als recht schwierig. Am einfachsten geht das mit dem Pseudobefehl

DEFS n Reserviere n Bytes ab der aktuellen Lage des Programmzählers.

Um auf diesen Bereich einfach zurückgreifen zu können, wird dem Befehl ein Label zugeordnet. Zur Einrichtung eines reservierten Speicherbereichs von 4 byte Länge mit dem Label STORE: sollte folgendes vereinbart werden:

```
STORE:  DEFS 4
```

Wenn der Assembler bei der Übersetzung auf diesen Pseudobefehl stößt, wird er einen 4 byte langen Pufferspeicher einrichten, dessen Startadresse bei STORE: liegt.

Beispiel:

Nehmen Sie einmal an, daß eine 8*8-bit-Multiplikation ein Ergebnis mit 16 bit Wortlänge in HL liefert. Wenn Sie dieses Ergebnis im Programm weiter verwenden wollen, müssen Sie es irgendwo dauerhaft zwischenspeichern. Falls Sie die folgende Programmstruktur verwenden, können Sie das Ergebnis in einem bei STORE beginnenden Puffer ablegen.

```
        ENT
STORE:  DEFS  2
          .
          .
Multiplikationsprogramm
          .
          .
        LD      (STORE:),HL
```

Der Inhalt der durch STORE: näher bezeichneten Adresse enthält dann das Ergebnis der Multiplikation. Natürlich gibt es Pseudobefehle auch noch für andere Aufgaben.

Numerische Werte können beispielsweise durch den Pseudobefehl

EQU Ordne den Wert nn dem vorangestellten Label zu.

zugeordnet werden.

Verwendet wird dieser Befehl in der Regel zur Definition von konstanten Werten.

Beispiel:

```
LOOP:   EQU   20
        LD    A,LOOP:
```

Dies führt dazu, daß A mit dem Wert 20 geladen wird.

Bisher wurden die meisten Programme mit der ENT-Anweisung eingeleitet. Das ist zwar ganz nützlich für die Entwicklung und den anschließenden Test von Programmen. Wenn ein Maschinenprogramm dagegen von der BASIC-Ebene aus aufgerufen werden soll, muß es an einer Stelle im Speicher abgelegt werden, an der es nicht überschrieben werden kann. Der für BASIC verfügbare Speicherbereich wird durch den MEMORY-Befehl festgelegt. Wenn Sie beispielsweise MEMORY auf 39999 setzen, kann der Interpreter Speicherplätze oberhalb dieses Wertes nicht für die Ablage von Daten verwenden. Sie können so einen Speicherbereich für Maschinenprogramme schützen.

Wenn Sie nicht dem Assembler selbst die Entscheidung über die Startadresse eines Maschinenprogramms überlassen wollen, dann müssen Sie von dem Pseudobefehl

ORG nn Setze den Programmzähler auf den Wert nn.

Gebrauch machen. Hierdurch können Sie den Assembler anweisen, das übersetzte Programm ab der unter nn vereinbarten Adresse im Speicher abzulegen. Achtung: Der Wert nn muß größer sein als die Adresse der letzten vom Editor/Assembler verwendeten Speicherstelle. Wenn Sie einen unerlaubten Wert eingeben, wird Sie der Assembler davon mit einer entsprechenden Fehlermeldung unterrichten.

In vielen Programmen müssen Speicherbereiche mit Daten geladen werden. Hierfür stehen drei Pseudobefehle zur Verfügung. Die ersten beiden laden numerische Daten in den Speicher.

DEFB n Speichere den Wert n (1 Byte) an der durch den Programmzähler bestimmten Stelle.

DEFW nn Lade das niederwertige Byte eines 16 bit langen Wertes nn unter der aktuellen Programmzähleradresse, das nächsthöhere Byte unter der nächsthöheren Adresse.

Schließlich gibt es noch einen Pseudobefehl, mit dessen Hilfe ASCII-Codewerte einer Zeichenkette in den Speicher geladen werden können. Von dieser Möglichkeit wird beispielsweise Gebrauch gemacht, um Meldungen auf dem Bildschirm auszugeben. Der Befehl lautet:

DEFM "s" Lade die Speicherzellen beginnend bei der aktuellen Adresse des Programmzählers mit den unter "s" angegebenen Elementen einer Zeichenkette.

Probieren Sie einmal das folgende Programm aus:

```
        ENT
MESS:   DEFM  "EINE NACHRICHT!"
        LD    B,15
        LD    HL,MESS:
LOOP:   LD    A,(HL)
        CALL  &BB5A
        INC   HL
        DJNZ  LOOP:
        RET
```

Addition von Operanden & direkte Abspeicherung von ASCII-Zeichen

Wenn Sie das Buch bis zu dieser Stelle durchgearbeitet haben, werden Sie sich vielleicht daran erinnern, daß in Kapitel 10 einmal die Zahl &80 zu einem ASCII-Zeichen addiert wurde, das in einem DEFB-Kommando enthalten war. Mit dieser Befehlskombination soll die aktuelle Speicheradresse, auf die der Programmzähler verweist, mit dem in Anführungsstrichen gesetzten ASCII-Zeichen geladen werden, nachdem zuvor der Wert &80 hinzugezählt wurde. Die Wirkung ist die, daß das Bit 7 gesetzt wird. Dies war deshalb notwendig, weil beim letzten Zeichen eines externen Befehls das Bit 7 gesetzt werden muß.

Um beispielsweise den ASCII-Code für den Buchstaben A in den Akkumulator zu laden, können Sie den Befehl

```
LD A,"A"
```

verwenden. A wird dann mit 65 geladen. Um den Akkumulator mit dem ASCII-Wert des Buchstabens B zu laden, können Sie auch den Befehl

```
LD A,"A"+1
```

verwenden. Diese Art der Programmierung ist manchmal recht nützlich, wenn Datenfelder eingerichtet werden müssen.

Kommentare

Sie können im Assemblerlisting Kommentare vorsehen. Diese müssen eindeutig durch Voranstellen eines Semikolons kenntlich gemacht werden.

Beispiel:

```
10 ENT
20 ;Lade A mit 65
30 LD A,65
40 ;Gib A auf dem Bildschirm aus
50 CALL &BB5A
60 RET
```

Beachten Sie bitte, daß die Kommentare in eigenen Zeilen vorgesehen werden müssen. Sie können nicht mit Befehlen zusammen in einer Zeile stehen.

Anhang G

Lösungen zu den Übungen

Kapitel 1

Übungsaufgabe 1.1

```
ENT
LD    A,65
CALL  47962
RET
```

Übungsaufgabe 1.2

```
ENT
LD    A,70
CALL  47962
LD    A,82
CALL  47962
LD    A,69
CALL  47962
LD    A,68
CALL  47962
RET
```

Kapitel 2

Übungsaufgabe 2.1

```
ORG   30000
LD    B,10
LD    A,65
CALL  47962
DEC   B
JR    NZ,30004
RET
```

Übungsaufgabe 2.2

Der JR-Befehl wurde benutzt, da die Sprungadresse innerhalb des Bereichs von +129 bis -126 Bytes von der Adresse des Befehls liegt. Der relative Sprungbefehl braucht gegenüber dem absoluten ein Byte weniger Speicherplatz.

Übungsaufgabe 2.3

```
      ORG   30000
      LD    C,26
      LD    A,65
NXT:  CALL  47962
      INC   A
      DEC   C
      JR    NZ,NXT:
      RET
```

Kapitel 3

Übungsaufgabe 3.1

Adresse	Inhalt
:	:
:	:
200	E
201	D
202	L
203	H
:	:
:	:

Übungsaufgabe 3.2

```
ENT
LD    DE,100
LD    (35000),DE
LD    HL,400
LD    (35002),HL
LD    DE,(35000)
LD    HL,(35002)
CALL  48118
RET
```


Wenn Sie die Maschinenroutine bei 48016 benutzen, sieht das Programm folgendermaßen aus:

```
ENT
LD    DE,100
LD    (35000),DE
LD    HL,400
LD    (35002),HL
LD    DE,0
LD    HL,0
CALL  48106
LD    DE,(35000)
LD    HL,(35002)
CALL  48118
RET
```

Übungsaufgabe 3.3

```
ENT
LD    DE,200
LD    HL,300
CALL  48118
LD    DE,400
LD    HL,200
CALL  48118
LD    DE,0
LD    HL,0
CALL  48118
RET
```

Beachten Sie: Es gibt keinen Grund, die Koordinaten zuerst in den Speicher zu schreiben und anschließend wieder auszulesen. Wir haben das hier nur deshalb so gemacht, um diese Speicher-Technik einmal vorzuführen.

Übungsaufgabe 3.4

```
      ENT
      LD    BC,35000
      LD    A,65
      LD    E,3
NXT:  LD    (BC),A
      INC   BC
      DEC   E
      JR    NZ,NXT:
```

```
        LD    E,3          Anzahl der Schleifendurchläufe
        LD    BC,35000     Startadresse der A´s
PUT:    LD    A,(BC)       A in Akkumulator laden
        INC   BC           Eine 1 zur Datenadresse addieren
        CALL  47962        Ein A ausgeben
        DEC   E            Schleifenzähler erniedrigen
        JR    NZ,PUT:      Wenn Schleifenzähler <> 0, Sprung zu PUT:
        RET                Rücksprung
```

Übungsaufgabe 3.5

```
        ENT
        LD    BC,35000
        LD    A,65
        LD    E,26         26 Schleifendurchläufe
NXT:    LD    (BC),A
        INC   BC
        INC   A            Nächstes ASCII-Zeichen
        DEC   E
        JR    NZ,NXT:
        LD    BC,35000
        LD    E,26         26 Schleifendurchläufe
PUT:    LD    A,(BC)
        INC   BC
        CALL  47962
        DEC   E
        JR    NZ,PUT:
        RET
```

Übungsaufgabe 3.6

```
        ENT
        LD    IX,35000
        LD    A,83
        LD    (IX+0),A
        LD    (IX+2),A
        INC   A
        INC   A
        LD    (IX+1),A
        LD    A,65
        LD    (IX+3),A
        LD    A,78
        LD    (IX+4),A
        LD    A,(IX+2)
```



```
        CALL  47962
        LD    A,(IX+3)
        CALL  47962
        RET
```

Kapitel 4

Übungsaufgabe 4.1

```
        ENT
        LD    A,200
        ADD   A,48
        CALL  47962
        RET
```

Es wird ein kleines Männchen auf dem Bildschirm ausgegeben.

Übungsaufgabe 4.2

```
        ENT
        LD    A,&41
        ADD   A,&10
        CALL  &BB5A
        RET
```

Dieses Programm schreibt ein Q auf den Bildschirm.

Übungsaufgabe 4.3

```
        ENT
        LD    C,&FA        LSB von 250
        LD    A,58         LSB von 600
        AND   A
        ADD   A,C
        PUSH  AF           Flags speichern (interessant ist hier nur C)
        ADD   A,65
        LD    (&7000),A
        POP   AF           Flags zurückholen
        LD    C,&00        MSB von 250
        LD    A,&2         MSB von 600
        ADC   A,C
        ADD   A,65
        LD    (&7001),A
```

```
LD    A,(&7000)
CALL  &BB5A
LD    A,(&7001)
CALL  &BB5A
RET
```

Übungsaufgabe 4.4

```
ENT
LD    C,9
LD    A,233
SUB   C
CALL  &BB5A
RET
```

Es ist nicht nötig, zu dem Ergebnis 224 einen Offset zu addieren oder zu subtrahieren, da CHR$(224) ein sichtbares Zeichen (lachendes Gesicht) ergibt.

Übungsaufgabe 4.5

```
ENT
LD    A,126
ADD   A,97
LD    C,153
SUB   C
CALL  &BB5A
RET
```

Dieses Programm gibt ein F auf dem Bildschirm aus.

Übungsaufgabe 4.6

```
ENT
LD    DE,4008
LD    HL,4248
AND   A
SBC   HL,DE
LD    A,L
CALL  &BB5A
RET
```

Das Programm gibt einen nach oben gerichteten Pfeil auf dem Bildschirm aus.

Übungsaufgabe 4.7

```
ENT
LD    HL,35000
LD    A,10
LD    (HL),A
INC   HL
LD    A,20
LD    (HL),A
LD    A,65
ADD   A,(HL)
LD    (HL),A
DEC   HL
LD    A,65
ADD   A,(HL)
LD    (HL),A
CALL  &BB5A
INC   HL
LD    A,(HL)
CALL  &BB5A
RET
```

Die beiden Buchstaben KU werden ausgegeben.

Übungsaufgabe 4.8

```
ENT
LD    DE,100
LD    HL,50
CALL  48118
LD    D,0
LD    E,100
LD    A,75
ADD   A,E
LD    E,A
LD    H,0
LD    L,50
LD    A,75
ADD   A,L
LD    L,A
CALL  48118
RET
```

Kapitel 5

Übungsaufgabe 5.1

```
ENT
LD    A,7
ADD   A,5
DAA
ADD   A,65
CALL  &BB5A
RET
```

Dieses Programm gibt ein S auf dem Bildschirm aus.

Übungsaufgabe 5.2

```
ENT
LD    C,&12
LD    A,&35
SUB   C
DAA
ADD   A,65
CALL  &BB5A
RET
```

Dieses Programm gibt ein d auf dem Bildschirm aus.

Übungsaufgabe 5.3

1. C=0
2. C=1
3. C=0

Übungsaufgabe 5.4

Die benötigte Maske ist 00000011.

Übungsaufgabe 5.5

253 UND 75 = 73 (1001001)

Übungsaufgabe 5.6

1. 1001 ODER 1101 = 1101
2. 250 ODER 25 = 251
3. (209 ODER 20) UND 27 = 17

Übungsaufgabe 5.7

1. 1011 Exklusiv-ODER 1110100 = 121

```
ENT
LD    C,11
LD    A,116
XOR   C
CALL  &BB5A
RET
```

Dieses Programm gibt ein Grafikzeichen ähnlich einem Schachbrett auf dem Bildschirm aus.

2. 77 Exklusiv-ODER 200 = 133

```
ENT
LD    C,77
LD    A,200
XOR   C
CALL  &BB5A
RET
```

Das Programm gibt einen vertikalen Balken auf dem Bildschirm aus.

3. (25 ODER 255) UND 200 = 200

```
ENT
LD    C,25
LD    A,200
OR    C
LD    C,200
AND   C
CALL  &BB5A
RET
```

Dieses Programm gibt zwei diagonale Linien auf dem Bildschirm aus.

Übungsaufgabe 5.8

1. 0100
2. 0100010
3. 0001

Übungsaufgabe 5.9

1.	1010	= 10
	1101	= -3
	0111	7
2.	1111	= -1
	0111	= 7
	0110	6
3.	0110	= -10
	1000	= 8
	1110	-2

Übungsaufgabe 5.10

```
ENT
LD    A,20
CPL
INC   A
ADD   A,98
CALL  &BB5A
RET
```

Dieses Programm gibt den Buchstaben N auf dem Bildschirm aus.

Kapitel 6

Übungsaufgabe 6.1

Die Lösung entspricht Programm 6.1 bis auf die folgenden Zeilen:

```
LD    C,10
LD    E,9
```

Das Programm gibt das Zeichen mit dem Code 155 auf dem Bildschirm aus. Das ist eine Art umgedrehtes T.

Übungsaufgabe 6.2

```
      ENT
      LD    C,124
      LD    E,146
      LD    D,0
      LD    B,8
      LD    HL,0
NXTB: SRL   C
      JR    NC,NOADD:
      ADD   HL,DE
NOADD: SLA  E
      RL    D
      DEC   B
      JR    NZ,NXTB:
      LD    A,H
      CALL  &BB5A
      LD    A,L
      CALL  &BB5A
      RET
```

Das Programm erzeugt die Ausgabe F und Pi.

Übungsaufgabe 6.3

Am einfachsten löst man die Aufgabe, indem man DEC B löscht und die Zeile JR NZ,NXTB: durch DJNZ NXTB: ersetzt.

Übungsaufgabe 6.4

1.
```
      ENT
      LD    A,5
      RLCA
      RLCA
      RLCA
      RLCA
      RLCA
      CALL  &BB5A
      RET
```

Das Programm gibt das Zeichen mit dem Code 160 aus, eine Pfeilspitze nach oben.

```
2.      ENT
        RRCA
        CALL  &BB5A
        RET
```

Dieses Programm gibt das Zeichen mit dem Code 127 aus, eine Art Schachbrett.

Übungsaufgabe 6.5

```
        ENT
        LD    A,255
        CALL  &BB5A
        RES   4,A
        CALL  &BB5A
        SET   4,A
        RES   3,A
        CALL  &BB5A
        RET
```

Das Programm gibt einen Pfeil, eine Rakete und eine auf der Seite liegende Pyramide auf dem Bildschirm aus.

Kapitel 7

Übungsaufgabe 7.1

```
        ENT
        LD    DE,0
        LD    HL,0        Grafikcursor auf (0,0) setzen
        CALL  &BBC0
        LD    DE,100      DE mit 100 laden
        LD    HL,200      HL mit 200 laden
        PUSH  DE          DE auf Stapel schreiben
        PUSH  HL          HL auf Stapel schreiben
        CALL  &BBF6       Linie nach (100,200) zeichnen
        LD    DE,0
        LD    HL,0        Cursor auf (0,0) setzen
        CALL  &BBC0
        POP   DE          Alten Inhalt von HL in DE laden
        POP   HL          Alten Inhalt von DE in HL laden
        CALL  &BBF6       Linie nach (200,100) zeichnen
        RET
```


Kapitel 8

Übungsaufgabe 8.1

```
              ENT
              LD    HL,&B100
              LD    DE,&C000
              LD    BC,&3FFF
LOOP:         LDI
              JP    PO,ENDE:
              JP    LOOP:
ENDE:         RET
```

Anhang E

Übungsaufgabe AE.1

a. 3
b. 4
c. 128
d. 131
e. 183
f. 115

Übungsaufgabe AE.2

a. 31
b. 41
c. 189
d. 136

Übungsaufgabe AE.3

a. 9
b. 19
c. 165
d. 174
e. 14
f. 26
g. 234

Übungsaufgabe AE.4

a. 0000 0100
b. 0101 0011
c. 0001 0000
d. zu groß für 2-Byte-BCD-Darstellung
e. 0111 0111
f. zu groß für 2-Byte-BCD-Darstellung
g. 1001 0111
h. zu groß für 2-Byte-BCD-Darstellung

Übungsaufgabe AE.5

a. 1
b. 9
c. 15
d. 20
e. 49
f. 23
g. 97
h. 88

Anhang H

Anleitung zum Konvertieren des Assemblerkurses von Kassette auf Diskette

1. Die Programme

   LADER
   ASSEMBLER
   BIN/HEX
   GRAPHIK-DEMO

   lassen sich durch einfaches Laden von Kassette und anschließendes Abspeichern auf Diskette konvertieren. Sie müssen dafür mittels der Befehle |TAPE und |DISC zwischen Kassetten- und Diskettenbetrieb hin- und herschalten.

2. Das Programm ASSDAT wird wie folgt konvertiert:

```
|TAPE
MEMORY 12499
LOAD "ASSDAT"
|DISC
SAVE "ASSDAT",B,12500,10000
```

3. Für das Programm GRAPHIK-B gibt es zwei Möglichkeiten. Sie können mit dem Assembler den Quellcode GRAFEXT laden (der dann auch gleich konvertiert werden kann) und dann assemblieren. Danach kann der Maschinencode mit SAVE "GRAPHIK-B" ("-B" ist wichtig) auf Diskette abgespeichert werden. Damit dies geschehen kann, muß man zwischendurch ins BASIC zurück und dort den Befehl |DISC eingeben. Danach können Sie in den Assembler zurück und die Datei abspeichern.

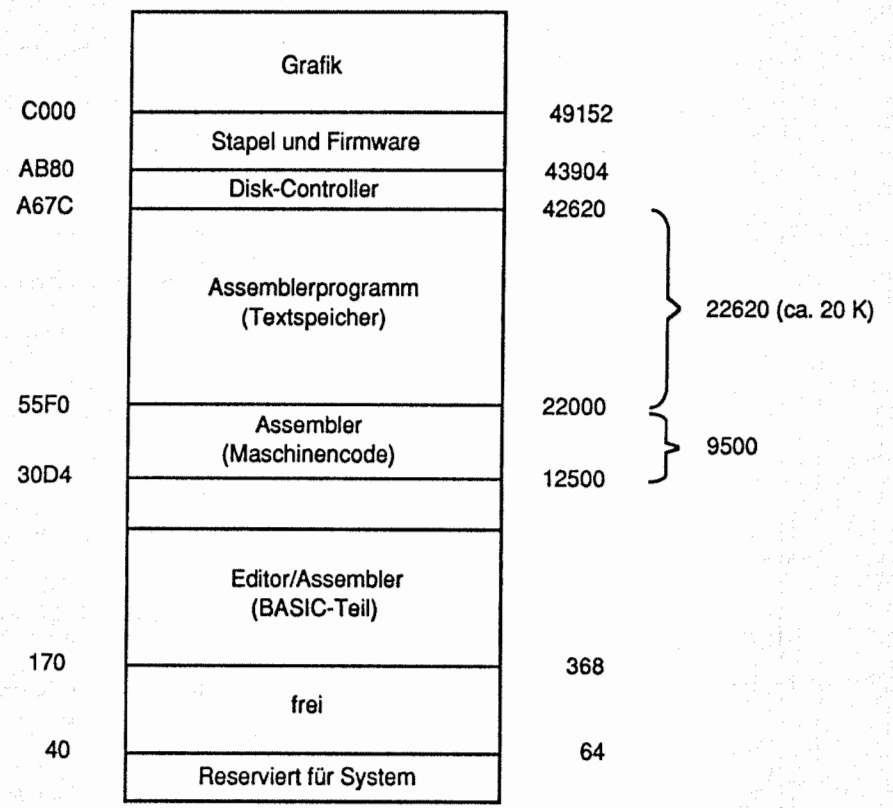
   Nun zur zweiten Möglichkeit. Sie können die folgende Befehlssequenz benutzen:

```
|TAPE
MEMORY 39999
LOAD "GRAPHIK-B"
|DISC
SAVE "GRAPHIK-B",B,40000,1000
```

4. Das Programm GRAFEXT wird ähnlich wie GRAPHIK-B (erste Methode) konvertiert. Zuerst wird der Assembler geladen. Verlassen Sie den Assembler und schalten Sie im BASIC mit |TAPE auf Kassettenbetrieb. Nun gehen Sie zurück in den Assembler (Punkt-Taste im Zehnerblock drücken). Hier können Sie jetzt mit L das Programm laden. Gehen Sie wieder ins BASIC, schalten Sie mit |DISC auf Diskettenbetrieb um, und kehren Sie zurück in den Assembler, von wo aus Sie das Programm dann abspeichern (mit dem Befehl S).

Anhang I

Speicherbelegung



Stichwortverzeichnis